Formal Verification of Sequential Circuits Using Reasoning Techniques

Dissertation
zur Erlangung des Doktorgrades
der Naturwissenschaften

vorgelegt beim Fachbereich Informatik
der Johann Wolfgang Goethe – Universität
in Frankfurt am Main

von
Dominik Stoffel
geb. in Xenia, Ohio, USA

Frankfurt am Main, 1999
DF 1

vom Fachbereich Informatik der Johann Wolfgang Goethe-Universität als Dissertation angenommen.

Dekan:	Prof. Dr. Wolfgang Kunz
Gutachter:	Prof. Dr. Wolfgang Kunz, Universität Frankfurt Prof. Dr. Prem Menon, University of Amherst, MA, USA Prof. Dr. Joachim Wunderlich, Universität Stuttgart
Datum der Disputation:	20.12.1999

Contents

Acknowledgements

This thesis is the result of a several years lasting project at various locations and institutions. The major part was conducted while I was with the Max-Planck Fault-Tolerant Computing Group at the University of Potsdam. During a short but intensive stay at Mentor Graphics Boston Research Group in Billerica, MA, some of the core ideas for sequential equivalence checking were developed. During the last one and a half years I have been with the Electronic Design Automation Group at the University of Frankfurt, finishing this work. At all these locations there are many people who contributed to this research in various ways and to whom I am very grateful.

First of all, I am greatly indebted to my supervisor, Prof. Dr. Wolfgang Kunz, who is the head of the Design Automation Group in Frankfurt. His constant support, encouragement and advice have been very imporant in the course and for the outcome of this work. Many ideas described in this thesis were born during the long, intensive and inspiring discussions with Wolfgang.

I am also very grateful to Prof. Dr. Michael Gössel, head of the Max-Planck Fault Tolerant Computing Group, who has generously supported me in many ways and given me important advice and suggestions. I have to thank him for the opportunity to work within the excellent research environment he has provided. Exchanging ideas with the research group and the many visitors from international research institutions has provided me with important experiences and input to my work.

A partial funding of this research came from Mentor Graphics' Boston Research Group in Billerica, MA, for which I am very grateful. I owe special thanks to Reilly Jacoby and Henry Cox, for making it possible to visit the group for a few months. Thanks to the great people I have met there, this visit was both, very productive for my work and also a lot of fun.

The work on AND/OR reasoning graphs is based on earlier research by Wolfgang Kunz and Prem Menon. I would like to thank Professor Menon for the helpful discussions and feedback he gave on this subject.

Furthermore, I would like to thank all my colleagues in Potsdam and Frankfurt, especially Stefan Gerber, Martin Cobernuss, Hendrik Hartje, Andrej Morosov, Mitrajit Chatterjee, Petra Vogel, Manuela Zeitner, Ingmar Neumann and Kolja Sulimma, for the interesting discussions, fruitful collaborations and the fun we had.

Last, but not least, I am indebted to my parents for their continuous support of my work, especially to my father for much helpful advice and the time he spent proof-reading this thesis.

Dominik Stoffel Frankfurt am Main, September 9th, 1999

Chapter 1

Introduction

Designing state-of-the art digital electronic circuits is nearly impossible without the use of Computer-Aided Design (CAD) tools. Today's design tasks involve millions of transistors and only the – at least partial – automation of the design process has made it possible to successfully cope with the complexity of modern VLSI (Very Large Scale Integration) circuits. CAD tools help to specify, synthesize and re-use designs, allow for a modular and hierarchical design architecture making the design process systematic, and provide means to communicate data between individuals in large design teams. Especially, the process of optimizing the quality of a design in terms of performance, power consumption, testability and manufacturing yield is far too time-consuming and error-prone to be accomplished manually by a human designer. Highly sophisticated automatic synthesis methods have been developed making this step simple. CAD tools help to reduce the time-to-market of a new product. Due to continuous improvements in circuit technology the period during which a product is competitive in the market keeps shrinking. It is therefore extremely important for a manufacturer to develop a product fast in order to regain the costs of design and fabrication and to make profit. This holds true especially for *Application Specific Integrated Circuits (ASICs)* where design costs make up a larger part of the total product costs than for a generic high-volume production IC (Integrated Circuit).

In addition to a short design time, it is very important that the design is free of errors. Firstly, it is nearly impossible to repair a chip or a production mask. Secondly, after the mask layout for a chip has been sent to the silicon foundry, the turnaround time for chip production is several weeks. If a design error is detected in the fabricated IC, a re-design and re-fabrication becomes necessary, involving additional cost and time delays. With the given tight cost and time constraints, such an iteration is not acceptable. Therefore, a major part of the design resources must be used for *design validation*.

The purpose of design validation is to ascertain that the designed circuit functions correctly and has the properties intended by the designer. There are basically three approaches to design validation: simulation, emulation and formal verification. Simulation is the most widely employed technique today. Designers apply input patterns (stimuli) to a software simulation model of the design and check whether it produces the intended outputs. The patterns may be chosen specifically to expose a certain behaviour or possible logical errors of the design, or may be created randomly. Large numbers of patterns are simulated to increase the confidence in the correctness of the design.

Emulation consists of building a prototype of the circuit for example using programmable logic such as *Field Programmable Gate Arrays (FPGAs)*. Input stimuli are applied to the prototype at high speeds and the produced output patterns are checked for correctness. Emulation accelerates simulation so that many more patterns can be evaluated in a given amount of time. However, even then, the set of simulated patterns usually represents only a small fraction of the exhaustive set of patterns, and it is always possible to overlook a design error. A well-known example for this to happen is the bug in Intel's Pentium processor making world-wide headlines a few years ago.

Formal verification is an alternative to simulation-based methods. Formally verifying the correctness of a design means determining by mathematical proof that the design has the desired function and fulfills the required properties *for all possible input patterns*. By conducting a formal proof also those errors ("bugs") can be found that occur with low probability and which are hard to detect by simulation. It is common to distinguish between two main application areas of formal verification techniques, *design verification* and *implementation verification*.

1.1 Design Verification

The first phase of a design process is to create an initial circuit model representing the specification of the desired functionality. In this conceptual phase, design verification is used to make sure that the initial model fulfills all requirements and has all the properties intended by the designer. Design verification cannot only aid in finding bugs in the specification, it can also increase the designers understanding of the design issues. The designer formulates the properties of the design to be checked in an appropriate language or formalism. Typical formalisms used are certain logics such as propositional logic, temporal logic, first-order predicate logic, and also finite-state automata operating on finite or infinite words. The properties expressed in such a formalism are checked by an automated method of proof suited for this formalism.

The most successful methods for design verification are *model checking* (see e.g., [66]) and *language containment* (see e.g., [56]). For model checking, the properties to be checked are expressed in a simplified form of temporal logic defined by Clarke and Emerson [22] called *Computation Tree Logic (CTL)*. The system to be checked is represented as a so-called *Kripke model* which is similar to a finite state machine. Properties are checked by propagating formulas in the state transition graph of this model until a fixed point is reached. Language containment methods are based on the theory of ω-regular automata. Both the design to be checked and the properties to be proven are represented as finite automata. The property is verified by proving that the language of the design automaton is contained in the language of the property automaton.

Although the formalism used for expressing properties is different, the two approaches of model checking and language containment are not too far apart. The core algorithms employed are similar and are based on fixed point iterations on state transition graphs. Well-known tools for model checking are SMV[66] which was developed at Carnegie Mellon University and VIS[9] from the University of California at Berkeley.

A serious problem in model checking (or language containment) limiting its applicability in practice is known as the *state explosion problem*. For example, take the case where some property is to be checked for all states that a system can possibly assume. Hence, to prove this property we need to determine the set of all states that can be reached from the set of initial states. This task is called *reachability analysis*. It plays a central role in formal verification and involves the traversal of the complete state transition graph of the system. However, the number of states of a sequential system, i.e., a system containing memory elements like latches and flip-flops, can be exponential in the number of these memory elements. Since a representation of the relevant state sets is needed when performing the fixed point iterations mentioned above, these methods fail when the memory required for the state set representation is too large. Researchers [26, 66] have successfully alleviated this problem by introducing implicit representations of state sets using *binary decision diagrams, (BDDs)* [14]. BDDs are efficient data structures for representing Boolean functions in a canonical form. Large sets of states can be represented by constructing the BDD of the characteristic function of a state set. However, in some cases also BDDs exhibit their worst-case size behaviour which is also exponential in the number of variables. Therefore, even with these improvements many industrial-size circuits remain too complex for state-of-the-art model checking techniques.

1.2 Implementation Verification

While design verification is used to check the technical soundness of a specification, implementation verification has the task to verify the correctness of the various design steps transforming the specification into the final implementation. The initial specification is usually given in a hardware description language such as *Verilog* or *VHDL* at the *register transfer level (RTL)*. Synthesis tools transform this specification into a gate-level model which is then optimized for performance, power consumption, area, testability etc. After technology mapping and a physical design phase the layout of the final chip is obtained. For each of these design steps, it must be verified that the transformations of the circuit model have not changed the specified function. There are a number of possible scenarios how a design can become incorrect. Firstly, the programs used for synthesis and optimization may have caused errors. Although the core algorithms of the CAD tools are correct by construction, the software implementation of these algorithms may be erroneous. Secondly, errors may be introduced not only by the actual CAD programs but also by interface software such as data format converters or other software developed by the user. Thirdly, very often designers make manual changes to the design, especially in the optimization phase, to improve or re-design certain critical parts of the circuitry generated by the CAD tool.

Implementation verification is performed by checking the functional equivalence of the specification and the implementation after all modifications. This is the task of an *equivalence checking* tool. Checking the equivalence of two models A and B means to verify that the output behaviour of A and B is the same for all possible inputs (of interest). Equivalence checking is often repeated several times in order to verify the correctness of circuit modifications along the design process.

If a design does not exhibit sequential behaviour, i.e., if it does not contain any memory elements such as registers and latches, the logic functions to be verified are purely combinational.

In this case, a traditional approach is to represent both circuit models in a canonical (= unique) form, e.g., using binary decision diagrams and then to check the isomorphism of both representations. In many cases, however, we encounter that the logic functions to be checked for equivalence have BDD representations which are exponential in the number of input variables. Such BDD blow-up is particularly common in the data path of a design containing barrel shifters, multipliers or related functions.

Sequential equivalence checking is even more difficult. Since the behaviour of the system is also dependent on its current state, it is not sufficient to only consider all input stimuli of the circuits. Additionally, *all possible states* of the system have to be considered, too. Therefore, it is necessary to perform a *reachability analysis* like in model checking in order to obtain the set of reachable states. Consequently, sequential equivalence checking methods face the same state-explosion problem as model checking and are applicable only to small and mid-size circuits.

1.3 Motivation and Thesis Overview

State-of-the-art formal methods for hardware verification rely heavily on Boolean techniques to represent and manipulate Boolean functions. The invention of binary decision diagrams [2, 14] and related graph representations of Boolean functions has played an important role in achieving major progress both in model checking and equivalence checking. Boolean function representations are not only useful to check combinational equivalence of two circuits but are also the basic instrument to represent and manipulate large state sets when traversing the state space of a *finite state machine (FSM)*. In particular, there is one property of BDDs which is very helpful in formal verification. BDDs can represent Boolean functions in a canonical form that is also compact in many cases. As mentioned, this is immediately useful in combinational equivalence checking where the equivalence check can be reduced to checking whether or not the BDDs for the two circuits are identical. Less obviously, canonicity is also a key property when performing fixed point iterations in model checking. As will be elaborated in later chapters the fixed point of these iterations is detected by noting that a certain state set being considered does not change from one iteration to next. Since the state set is represented as a Boolean function the detection of the fixed point is only guaranteed if the Boolean function is represented in a canonical form.

Unfortunately, there is a high price to be paid for the property of canonicity. Even when using BDDs, for many Boolean functions the canonical representation grows exponentially with the number of variables causing the complete failure of the formal verification method. Therefore, it is tempting to investigate how formal verification methods can operate without the use of canonical representations of Boolean functions. Dropping the requirement of canonicity, unfortunately, makes other aspects of formal verification algorithms more complex. So far, only in the domain of combinational equivalence checking there has been some notable success in developing verification techniques that do not require any canonical circuit representations [51, 8]. These methods employ Boolean reasoning techniques originally developed for the purpose of *automatic test pattern generation (ATPG)* and operate directly on the structural gate netlist of the circuit. Since they are capable of making efficient use of structural design properties they are often referred to as *structural* techniques. These techniques and their further developments (e.g., [76, 44, 62, 50]) have made combinational equivalence checking feasible for circuits with up to one million gates. Unfortunately, techniques of this kind are only in its infancy or do not exist at all for the important tasks of sequential equivalence checking or model checking.

Therefore, this thesis creates a theoretical framework and develops a set of practical techniques for a structural approach to sequential circuit verification.

Chapter 2 reviews some fundamentals of switching theory and introduces the reader into some terminology being basic to the subject of this thesis.

Chapter 3 introduces the Boolean techniques that form the basic reasoning apparatus for the formal verification algorithms developed in later chapters. The main contribution of this chapter is to introduce the notion of an implicant in multi-level combinational circuits and to present a method how such implicants can be determined. This makes it possible to generalize classical notions and algorithms for two-level circuits to multi-level circuits. The developed reasoning techniques are first discussed with circuit optimization as background leading to a unified view on two-level and multi-level circuit optimization. In the last part of this chapter, these concepts and algorithms are applied to the problem of set representation as needed in sequential circuit verification.

Chapter 4 develops a new approach to exploring the state space of a finite state machine called structural FSM traversal. A new approach is necessary since the Boolean techniques developed in Chapter 3 are not well-suited for being used in combination with traditional FSM traversal methods. In particular, in this chapter a structural fixed point iteration for performing reachability analysis is presented. The convergence behaviour of this fixed point iteration is theoretically examined by relating it to certain topological properties of the FSM's *state transition graph*.

Chapter 5 demonstrates the practical value of the developed theory and algorithms. It introduces an approximation to the exact structural FSM traversal of Chapter 4 and develops an algorithm for equivalence checking of sequential circuits. It is shown how the drawback of dealing with non-canonical circuit representations can be overcome by a controlled step-by-step circuit decomposition algorithm leading to a practical equivalence checking algorithm called *record_and_play()*. Experiments have been conducted to demonstrate the attractiveness of this approach.

Chapter 6 describes some future directions. It is discussed how the performance of the proposed Boolean reasoning techniques can be further increased and how the structural verification approach proposed in this thesis can be applied to model checking.

Chapter 7 concludes the thesis with a summary of the main results.

For reasons of better readability the proofs of the theorems in this thesis have been moved to the appendix.

Chapter 2

Boolean Algebra and Two-Level Circuit Theory

This chapter briefly reviews the basic concepts of switching theory. This summary is not complete; only such topics are considered which are of relevance for the understanding of later chapters. For a more detailed introduction into the theory of switching functions the reader may refer to a standard text book, e.g., [48], [64] and [33]. A second objective of this chapter is to familiarize the reader with the symbolic notations used in later chapters of this thesis.

2.1 Relations

In the following, we revisit some basic notions about relations which will be useful throughout this thesis. Standard notations are used to describe sets and operations on sets.

An ordered pair (a, b) is a pair of two elements with an order associated with them. The Cartesian product of two sets A and B, denoted $A \times B$, is the set of all ordered pairs (a, b), such that $a \in A$ and $b \in B$. A subset R of $A \times B$ is called a *binary relation* between A and B and we denote $a\, R\, b$ to express that element a is related to b by R. Since only binary relations are considered here, for reasons of brevity we simply speak of relations. In the sequel, we consider relations between elements of a single set, S, so $R \subseteq S \times S$.

The following properties of relations are important: A relation R between between the elements of a set S is called

reflexive	if $(a, a) \in R$ for all $a \in S$
symmetric	if $(a, b) \in R \longrightarrow (b, a) \in R$ for all $a, b \in S$
antisymmetric	if $(a, b) \in R$ and $(b, a) \in R \longrightarrow a = b$ for all $a, b \in S$
transitive	if $(a, b) \in R$ and $(b, c) \in R \longrightarrow (a, c) \in R$ for all $a, b \in S$

A relation R among the elements of a set S that is transitive, reflexive and symmetric is an *equivalence relation*. It *partitions* the set S. A *partition* is a set of disjoint subsets of S whose union is S. The subsets are called *blocks* of the partition or *equivalence classes*.

A relation R among the elements of a set S that is transitive, reflexive and antisymmetric is called a *partial order*. A set S in combination with a partial order is called a *partially ordered* *set*. As an example, the reader may think of relation R as the "is less than or equal to" relation to better understand the following notions. If and only if *(iff)* $a\,R\,b$ for every element $b \in S$, then a is called the *least element* of S. Similarly, a is said to be the *greatest element* of S, iff $b\,R\,a$ for all $b \in S$. Further let P be a subset of S. An element $s \in S$ is called an *upper bound* of P iff, for every $p \in P$, $p\,R\,s$. It is called a *lower bound* of P iff, for every $p \in P$, $s\,R\,p$. An upper bound s of P is defined to be the *least upper bound* iff sRs' for all upper bounds s' of P. Similarly, a lower bound s of P is called the *greatest lower bound* iff $s'\,R\,s$ for all lower bounds s' of P.

The notions of upper and lower bounds become intuitively clear if a partially ordered set is represented by a *Hasse diagram* as shown in standard text books. In the Hasse diagram, a partially ordered set for which a unique least upper bound and a unique greatest lower bound exists for every pair of elements looks like a grid or lattice.

Definition 2.1 (Lattice) *A* lattice *is a partially ordered set where every pair of elements has a unique greatest lower bound and a unique lowest upper bound.*

It immediately follows from this definition that a lattice has both, a least and a greatest element. We denote the least element by 0 and the greatest element by 1. Determining the least upper bound and the greatest lower bound can be viewed as two operations on the elements of the lattice. The operations consist in assigning the unique lowest upper bound or the unique greatest upper bound to each ordered pair of elements. These operations, in the following, are called *sum* or *product*, respectively, and are denoted:

$a + b$ = lowest upper bound (a, b)
$a \cdot b$ = greatest lower bound (a, b)

2.2 Boolean Algebra

Based on the operations of lowest upper bound (+) and greatest lower bound $(\cdot)$ introduced in the previous section, a Boolean algebra can be defined as follows.

Definition 2.2 (Boolean Algebra) *A lattice is called a* Boolean algebra *iff it is* complemented *and* distributive*, i.e., if the lattice fulfills the following two conditions:*

distributivity: $a \cdot (b + c) = a \cdot b + a \cdot c$ *and* $a + (b \cdot c) = (a + b) \cdot (a + c)$
complement: *for each element* a *in the lattice there exists a unique element* $\overline{a}$ *such that* $a \cdot \overline{a} = 0$ *and* $a + \overline{a} = 1$

Since the complement is unique, determining the complement can be considered a third operation on the elements of the lattice.

We have introduced a Boolean algebra as a special lattice. Another common way to define a Boolean algebra is to postulate certain properties for a set S and two binary operations $(\cdot)$ and $(+)$. These properties are known as *Huntington's postulates*. They include the existence of a complement and distributivity as given in Definition 2.2. Further postulates are *idempotency*, *commutativity*, *absorption*, *associativity* and the existence of *universal bounds*. Note that defining Boolean algebra by Huntington's postulates is equivalent to defining it as a complemented and distributive lattice. In fact, it can be proved that a set S in combination with two operations $(+)$ and $(\cdot)$ is a lattice iff the following laws are fulfilled:

idempotency:	$a \cdot a = a + a = a$
commutativity:	$a \cdot b = b \cdot a$ and $a + b = b + a$
absorption:	$a + a \cdot b = a$ and $a \cdot (a + b) = a$
associativity:	$a \cdot (b \cdot c) = (a \cdot b) \cdot c$ and $a + (b + c) = (a + b) + c$
universal bounds:	$a + 0 = a$ and $a \cdot 0 = 0$ and $a \cdot 1 = a$ and $a + 1 = 1$

Boolean algebra is the fundamental basis for the analysis of digital electronic circuits. In the 1930s, Shannon [85] showed that the behaviour of switching circuits can be described by a two-valued Boolean algebra. Such a switching algebra is obtained if we consider a lattice defined by the set $B = \{0, 1\}$ and the operations of conjunction (AND), disjunction (OR) and complementation (NOT). The operation of conjunction is also called product, disjunction can also be referred to as sum and complementation is often called negation. These operations are defined in Table 2.1.

a	b	$a \cdot b$
0	0	0
0	1	0
1	0	0
1	1	1

a	b	$a + b$
0	0	0
0	1	1
1	0	1
1	1	1

a	$\overline{a}$
0	1
1	0

Table 2.1: Definitions of conjunction (AND), disjunction (OR) and complementation (NOT)

The switching algebra defined by the operations in Table 2.1 is isomorphic to a two-valued Boolean algebra given by Definition 2.2. Therefore, in the sequel, we will use the terms Boolean algebra and switching algebra interchangeably.

2.3 Graphs

A graph $G(V, E)$ is a pair (V, E). V is a set and E is an incidence mapping on V. The elements of V are called vertices or nodes of the graph and the elements of E are called edges. Each edge $e \in E$ is associated with two nodes $v_i, v_j \in V$ being connected by this edge. If the nodes belonging to an edge are ordered the graph is called *directed*, otherwise it is called *undirected*. In the following we concentrate on directed graphs. In a directed graph, the node v_i of an edge (v_i, v_j) is the *immediate predecessor* of v_j, the node v_j is called *immediate successor* of v_i.

A graph $G' = (V', E')$ is called a subgraph of $G = (V, E)$ iff $V' \subseteq V$ and $E' \subseteq E$. The number of edges incident to a node v_i is called the *degree* of v_i. In a directed graph, the *outdegree* or *fanout* of a node refers to the number of its immediate successors. Similarly, the *indegree* or *fanin* is given by the number of its immediate predecessors. A node with indegree 0 is sometimes called a *source*, a node with outdegree 0 is called *sink* of the graph. A *loop* or *self-edge* is an edge with two identical end-points. A *walk* is an alternating sequence of vertices and edges. A *trail* is a walk with distinct edges, and a *path* is a trail with distinct vertices. A *cycle* is a closed walk (i.e., such that the end-point vertices coincide). A graph with no cycles is called an *acyclic* graph. A directed graph without cycles is called a *directed acyclic graph (DAG)*. DAGs represent partially ordered sets. If a DAG has one distinguished node, called *root*, which does not have any predecessors, it is called *rooted*. A *directed rooted tree* is a DAG such that all nodes other than the root have exactly one immediate predecessor. The immediate predecessor of a node v in a directed rooted tree is sometimes called *parent* of v, the immediate successors of v are called *children* of v. A node v_j that has the same parent as a node v_i is called a *sibling* of v_i. The nodes without successors are called *leaves* of the tree. For simplicity, in this thesis we will refer to directed rooted trees simply as trees.

In a DAG, a node v_j is called *successor* of a node v_i and v_i is called a *predecessor* of v_j iff there is a path from v_i to v_j. The set of all successors for a given node, in some literature, is also referred to as the *transitive fanout* of the node. Similarly, the set of all predecessors of a node are called its *transitive fanin*. Directed acyclic graphs are often used to describe structural and functional properties of switching circuits. The functional representations of Sections 2.4 and 3.3 associate Boolean functions with DAGs.

2.4 Boolean Functions and their Representations

Consider a Boolean algebra on a set B and a set of variables $x_1, x_2, \ldots, x_n$ such that each can be assigned independently an element of B. We say that the n variables $x_1, x_2, \ldots x_n$ form an n-dimensional Boolean space B^n where each vertex in the space is defined by one of the $|B^n|$ combinations of assignments. A mapping f, $f : B^n \mapsto B$, which uniquely associates every point of B^n with an element of B is called a Boolean *function*. In most applications it is $B = \{0, 1\}$. The vertices of B^n being mapped to 1 are called the *ON-set* of f and those being mapped to 0 are called the *OFF-set* of f. Often, Boolean functions are incompletely specified, i.e., for some vertices in B^n we "do not care" to what element of B they are mapped. These vertices form the so called *don't-care set* of f. This is usually denoted by introducing a third element for B, denoted X, so that $B = \{0, 1, X\}$. Further, we speak of an m-ary Boolean function if each point in the space is mapped to m elements of B, i.e., we consider a mapping f, $f : B^n \mapsto B^m$.

Boolean functions are the basis to describe the behaviour of switching circuits. Any method to optimize or analyze switching circuits relies on manipulating Boolean functions. Therefore, the efficiency of algorithms for specific problems in computer-aided design (CAD) of circuits depends highly on an appropriate representation of Boolean functions. In fact, the algorithmic solution of a given problem and the representation of the involved Boolean functions are intimately related and cannot be considered independently of each other. For simplicity, we speak of a *Boolean representation* when we mean the representation of a switching circuit as Boolean function.

Most commonly, Boolean functions are represented by *Boolean expressions* or by sets of Boolean expressions. General Boolean expressions, as opposed to sum-of-product forms (to be defined) are also called *factored forms*.

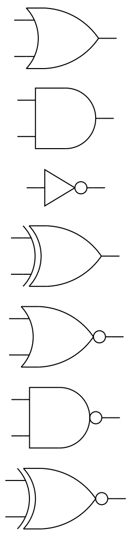
Definition 2.3 (Boolean expression) *A Boolean expression with the operations of disjunction, conjunction and negation is recursively defined by:*

1. *a variable is a Boolean expression,*
2. *the constants 0 and 1 are Boolean expressions,*
3. *the complement of a Boolean expression is a Boolean expression,*
4. *the disjunction of two Boolean expressions is a Boolean expression,*
5. *the conjunction of two Boolean expressions is a Boolean expression.*

A variable of a Boolean function in complemented or uncomplemented form is called a *literal*. For example, a and $\overline{a}$ are expressions containing the same variable, however, they are two different literals.

2.4.1 Implementing Boolean Functions as Switching Circuit

Switching circuits are constructed by interconnections of electronic gates implementing elementary Boolean functions like disjunction, conjunction or complementation. Table 2.2 depicts some standard logic gate types. Gates are electronic devices that produce voltage levels at their outputs



OR	$y = a + b$
AND	$y = a \cdot b$
Inverter (NOT)	$y = \overline{a}$
XOR	$y = a \cdot \overline{b} + \overline{a} \cdot b = a \oplus b$
NOR	$y = \overline{a + b}$
NAND	$y = \overline{a \cdot b}$
XNOR	$y = a \cdot b + \overline{a} \cdot \overline{b} = \overline{a \oplus b}$

Table 2.2: Standard digital electronic gates and their symbols

as a function of the voltage levels received at the inputs. These voltage levels are restricted to two ranges "high" and "low" and are usually associated with the logic values 1 and 0. Boolean functions can be implemented by an interconnection of electronic gates. If *all* Boolean functions can be implemented using a given set of gates then the set of gates is called *functionally* *complete*. For example {AND, OR, NOT} is functionally complete. Actually, either AND or OR can be removed from this set without making it incomplete, i.e., either {NAND} or {NOR}, each represents a functionally complete set of gates. Boolean functions can also be expressed exclusively by AND and XOR if the logic value 1 is available as constant input for the logic gates. Therefore, {1, AND, XOR} is a functionally complete set.

The implementation of a Boolean function by a switching circuit can be derived directly from the mathematical representation of the function. However, not every Boolean representation is equally suitable as a basis for a low cost implementation.

2.4.2 Disjunctive Forms

Let B denote a set forming a Boolean algebra. In some literature, the set B is referred to as *logic alphabet* and its elements are called *logic values*. The most naive way to represent a Boolean function is to list all vertices of the Boolean space as rows in a table and to associate with each row a logic value. An example of such a truth table is shown in Table 2.3.

x_1	x_2	x_3	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Table 2.3: Example of a tabular Boolean representation (truth table)

Since the truth table contains 2^n rows for a function with n variables, Boolean representation by truth tables is not practical except for very small examples. The following terminology is useful to describe other representations of Boolean functions. A *product term* is one of the following forms: the constant 1, a literal, or a conjunction of literals where no variable appears more than once. A product term that contains a literal for every variable of a Boolean function f is called *minterm* of f. Similarly, a *sum term* is one of the following forms: the constant 0, a literal, or a disjunction of literals where no variable appears more than once. A sum term that contains a literal for every variable in f it is called a *maxterm*. Minterms and maxterms can be associated with the rows of a truth table. If a minterm is associated with a row where f is 1 it is called *ON-set minterm*, if f is 0 it is called *OFF-set minterm*. Similarly, we can distinguish *ON-set maxterms* and *OFF-set maxterms*. For the above example, Table 2.4 shows the ON-set minterms and OFF-set maxterms.

A Boolean function is completely described by a disjunction of all its ON-set minterms or by a conjunction of all its OFF-set maxterms. In the former case we speak of the *disjunctive normal*

x_1	x_2	x_3	f	terms
0	0	0	0	$M_1 = x_1 + x_2 + x_3$
0	0	1	1	$m_1 = \overline{x_1} \cdot \overline{x_2} \cdot x_3$
0	1	0	0	$M_2 = x_1 + \overline{x_2} + x_3$
0	1	1	1	$m_2 = \overline{x_1} \cdot x_2 \cdot x_3$
1	0	0	0	$M_3 = \overline{x_1} + x_2 + x_3$
1	0	1	0	$M_4 = \overline{x_1} + x_2 + \overline{x_3}$
1	1	0	1	$m_3 = x_1 \cdot x_2 \cdot \overline{x_3}$
1	1	1	0	$M_5 = \overline{x_1} + \overline{x_2} + \overline{x_3}$

Table 2.4: Minterms and maxterms for a Boolean function

form (DNF) and in the latter we speak of the *conjunctive normal form (CNF)*. In the example of Table 2.4 the normal forms are:

$$y = \overline{x_1} \cdot \overline{x_2} \cdot x_3 + \overline{x_1} \cdot x_2 \cdot x_3 + x_1 \cdot x_2 \cdot \overline{x_3} \quad \text{(DNF)}$$
$$y = (x_1+x_2+x_3)+(x_1+\overline{x_2}+x_3)+(\overline{x_1}+x_2+x_3)+(\overline{x_1}+x_2+\overline{x_3})+(\overline{x_1}+\overline{x_2}+\overline{x_3}) \quad \text{(CNF)}$$

These normal forms are unique representations of Boolean functions, i.e., two Boolean functions are equivalent if and only if their normal forms contain the same minterms or maxterms, respectively. Unique representations of Boolean functions are also called *canonical*. Although conjunctive and disjunctive forms are equally important it is usually sufficient to only refer to one of them when describing notions and algorithms of switching theory. The standard literature gives preference to the disjunctive form. Therefore, the following considerations will be restricted to the disjunctive form.

A general Boolean expression can be transformed into its DNF by repeatedly applying Shannon's expansion theorem, given by

$$y(x_1, \ldots, x_n) = x_i \cdot y(x_1, \ldots, x_i = 1, \ldots, x_n) + \overline{x_i} \cdot y(x_1, \ldots, x_i = 0, \ldots, x_n)$$

In short form we write:

$$y = x \cdot y|_{x=1} + \overline{x} \cdot y|_{x=0}$$

The terms $y|_{x=1}$ and $y|_{x=0}$ are often called the *positive* and *negative cofactors* of y with respect to x.

Further, the notion of an *implicant* is very important in the theory of optimizing switching functions. An implicant for a function f is a product term p such that $p = 1 \longrightarrow f = 1$. A set of implicants is said to *cover* (to be a *cover* for) a function f iff for every minterm m of f, the set contains an implicant p with $m = 1 \longrightarrow p = 1$. An implicant p is called *prime* if the product term p is not an implicant anymore if any literal is removed, i.e., there exists no implicant that covers p and has less literals. A prime implicant of a function f is called *essential* if it is an element of every cover of f.

It is common to represent Boolean functions by a disjunction of implicants referred to as *sum of products (SOP)*. The dual representation is called *product-of-sums (POS)* which is a conjunction of *implicates*. An implicate q of a function f is a sum of literals such that $q = 0$ implies $f = 0$.

The sum of all prime implicants and the product of all prime implicates of a Boolean function f are both canonical representations of a Boolean function f. The sum of all prime implicants of f is sometimes called *complete sum* or *Blake normal form*. In contrast, a general Boolean expression according to Definition 2.3 which is neither a SOP nor a POS is often called a *factored form*.

It is often desirable to make a SOP as small as possible. If a SOP only contains prime implicants the SOP is called *prime*. If the function is not covered anymore if any of the implicants is removed, the SOP is called *irredundant*. Note that a prime and irredundant SOP is not a canonical representation of a Boolean function. There usually exist many possibilities to cover a Boolean function by a selection of prime implicants. Finding the right selection, i.e., determining a set of prime implicants that covers the function and leads to minimal cost when implementing the SOP as an electronic circuit, is the classical problem of two-level minimization. The first exact solution to this problem has been given by Quine [73] and McCluskey [63] and is known as the Quine-McCluskey method. For large circuits an exact solution may not be viable. Therefore, heuristic minimization techniques have been presented as, e.g., in ESPRESSO [11]. These heuristic methods heavily rely on exploiting properties of unate functions or sub-functions. A SOP-expression is called unate if each variable appears only in its complemented or uncomplemented form, but not both. For example, $y = \overline{a}b + bc$ is unate but $y = ab + \overline{b}c$ is not. A Boolean function is called unate, iff there exists a unate SOP-expression for it.

2.4.3 Boolean Networks

A circuit implementation of the forms described in Section 2.4.2 results in two-level circuits, i.e., any path from the inputs to the outputs of the circuit traverses at most two gates. In general, smaller circuits can be obtained if no restrictions are made on the number of levels in the circuit. Such multi-level circuits are usually described by *Boolean networks*.

Definition 2.4 (Boolean network) *A* Boolean network *is a triple (V, E, F) with the directed acyclic graph (V, E) and a set of Boolean functions, F.*

1. *The set $I \subseteq V$ is a set of nodes without predecessors, called* primary inputs, *of the Boolean network. The set $O \subseteq V$ is the set of nodes without successors called* primary outputs. *The edges leaving the nodes of I are associated with the input variables of the Boolean network.*

2. *The nodes $g_i \in V \setminus I$ are associated with Boolean functions $f_i \in F$ and the edges correspond to variables s_i denoting the functions f_i of the nodes which they are leaving. For every function f_j in the network there is an edge from g_i to g_j if f_j depends on s_i.*

Boolean networks are a technology-independent description of combinational circuits. In this thesis we consider circuits being described by *gate netlists* . The gates are elements of some library which depends on the available technology and the specific application. A gate netlist description is a special case of a Boolean network where each gate is associated with a node and the function of the node is defined by the gate function. In this work we generally assume that the library consists only of the gate types shown in Table 2.2. AND, OR, NOR, NAND can have an arbitrary number of inputs, and XOR, XNOR must have exactly two inputs. Note that this choice of the library corresponds to the usual assumptions in literature and contains the elements of a typical gate netlist description. Such a restricted Boolean network, in the sequel, will be referred to as *combinational network* or *combinational circuit* or *circuit netlist*, interchangeably. Further, avoiding formalism we often denote a node, function and variable in a combinational network with the same symbol and speak of nodes, functions, variables or signals interchangeably.

Two combinational networks C_1 and C_2 are said to have the same structure (denoted by $C_1 \equiv C_2$) iff the underlying circuit graphs (V_1, E_1) and (V_2, E_2) are isomorphic.

2.4.4 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) are graph representations of Boolean functions. The motivation to represent Boolean functions by binary decision diagrams is twofold: firstly, certain types of binary decision diagrams are canonical representations of Boolean functions. This is important in many applications, especially in formal verification. Secondly, certain algorithmic problems and manipulations of Boolean functions which have exponential worst case complexity for conventional representations as described in Sections 2.4.2 or 2.4.3, only have polynomial complexity for BDDs. In particular, this is true for the important *satisfiability* problem for a Boolean function, i.e., the problem of determining whether or not a Boolean function can evaluate to the logic value 1. This problem is *NP-complete* if the function is represented as a product of sums (conjunctive form), it is a constant time operation if the function is represented by a BDD.

Binary decision diagrams are special types of branching programs. For more information on branching programs see, e.g., [94]. It was first suggested by Akers [3] to use binary decision diagrams as a Boolean representation for solving problems in design automation and particularly in test generation. This became practical by a refinement of the model and the introduction of *reduced ordered binary decision diagrams (ROBDDs)* by Bryant [14]. Bryant also developed the basic algorithms for Boolean manipulations using ROBDDs.

Ordered binary decision diagrams (OBDDs) as proposed by Bryant [14] can be defined as follows:

Definition 2.5 (OBDD) *An* OBDD *is a rooted DAG with vertex set V. Each non-leaf vertex has as attributes a pointer* $\mathrm{index}(v) \in \{1, 2, \ldots, n\}$ *to an input variable in the set* $\{x_1, x_2, \ldots, x_n\}$*, and two children* $\mathrm{low}(v), \mathrm{high}(v) \in V$*. A leaf vertex v has as an attribute a value* $\mathrm{value}(v) \in B$.

For any vertex pair $(v, c_v), c_v \in \{\mathrm{low}(v), \mathrm{high}(v)\}$*, such that no vertex is a leaf,* $\mathrm{index}(v) < \mathrm{index}(c_v)$.

Roughly speaking, an OBDD represents a Boolean function if we associate the steps of a Shannon expansion with the nodes of the OBDD. This can be defined more precisely as follows:

Definition 2.6 *An OBDD with root v denotes a function f^v such that:*

1. *if v is a leaf with* $\text{value}(v) = 1$, *then* $f^v = 1$,
2. *if v is a leaf with* $\text{value}(v) = 0$, *then* $f^v = 0$,
3. *if v is not a leaf and* $\text{index}(v) = i$, *then* $f^v = \overline{x_i} \cdot f^{\text{low}(v)} + x_i \cdot f^{\text{high}(v)}$.

Example 2.1 As an example we construct an OBDD for function $f = a \cdot b + c$. Figure 2.1 shows the corresponding OBDD. As a convention, the child *low*(v) is always attached to the edge leaving v towards the left. The child *high*(v) is attached to the edge on the right. We build the OBDD by repeatedly performing Shannon's expansion. This illustrates the construction rule of OBDDs as given in Definition 2.6. It should be mentioned however that this is *not* quite how OBDDs are actually constructed for a given circuit description in practical applications.

The root node can be associated with function f. Performing a Shannon expansion for variable a we obtain the two cofactors c and $b + c$. These correspond to the left and right children of the root node. We now decompose these cofactors by further applications of Shannon's expansion and this is continued until constant values 0 and 1 are obtained as cofactors. In an ordered BDD the variables are always picked in the same order. For function f we have assumed a variable order (a, b, c), i.e., according to Definition 2.5 we choose the indices as: *index*$(a) = 1$, *index*$(b) = 2$, *index*$(c) = 3$.

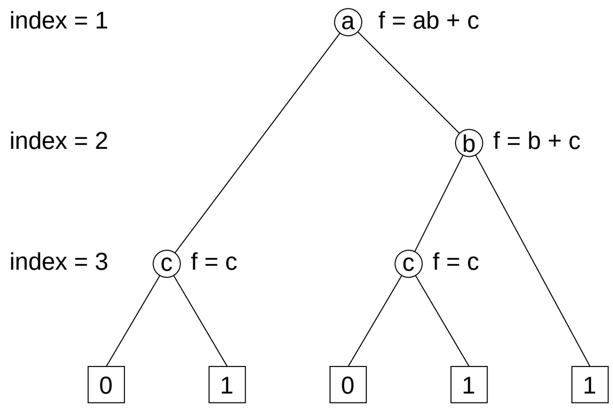


Figure 2.1: OBDD for function $f = a \cdot b + c$

The OBDD in Figure 2.1 can still be *reduced*. Only a reduced OBDD (ROBDD) is canonical. An OBDD is reduced by checking whether the graph contains isomorphic subgraphs.

Definition 2.7 *Two OBDDs $F = (V, E)$ and $F' = (V', E')$ are called isomorphic if and only if there exists a bijective function $m : V \mapsto V'$ such that for all $v \in V$, $m(v) \in V'$:*

1. *v and $m(v)$ are leaf vertices with the same value,* $\text{value}(v) = \text{value}(m(v))$, *or*
2. *v and $m(v)$ are non-leaf vertices with the same index,* $\text{index}(v) = \text{index}(m(v))$.
   Further it is $m(\text{low}(v)) = \text{low}(m(v))$ *and* $m(\text{high}(v)) = \text{high}(m(v))$.

In other words, two OBDDs are isomorphic if there is a one-to-one mapping between all nodes of the same type such that the children of a node in one OBDD are mapped onto the children of the corresponding node in the other OBDD.

> **Example 2.1 (continued)** Consider the OBDD in Figure 2.1. It contains isomorphic subgraphs, namely the subgraphs rooted in node c. By sharing these isomorphic subgraphs we obtain the reduced OBDD shown in Figure 2.2.

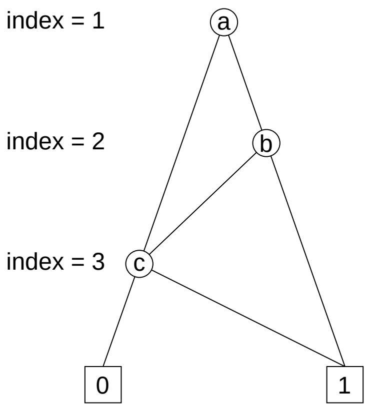


Figure 2.2: Reduced OBDD of Figure 2.1

Definition 2.8 (ROBDD) *An OBDD is called* reduced OBDD (ROBDD) *if it contains no vertex v with* $\text{low}(v) = \text{high}(v)$, *nor any nodes $v, v' \in V$ such that the subgraphs rooted in v and v' are isomorphic.*

In an OBDD, every path from the root to one of the leaves corresponds to a combination of value assignments for the variables of the represented function. If the path leaves a node through the left edge this means assigning 0 to the corresponding variable, the right edge corresponds to assigning a 1. If the path terminates in a leaf with value 1 then the set of value assignments makes the function evaluate to 1, otherwise, for the leaf with value 0 the function is 0. If the function is not satisfiable then there only exists a leaf with value 0 and the ROBDD consists only of this node.

ROBDDs are an important Boolean representation facilitating the solution of many problems in circuit design. However, in order to benefit from this representation it is essential to develop efficient algorithms to perform operations such as

- *apply*: given two OBDDs for functions f_1 and f_2, determine an OBDD for the function that results if a two-input Boolean function like conjunction (AND), disjunction (OR), antivalence (XOR), etc., is applied to f_1 and f_2.
- *compose*: given two OBDDs for functions f_1 and f_2, determine an OBDD for function f_1 if one variable x_i of f_1 is replaced by function f_2.

Table 2.5 shows the complexity of some basic operations on binary decision diagrams as proposed by Bryant [14]. For a function f its OBDD is denoted by F and the number of vertices in F is $|F|$. The implementation of these operations is based on the *if-then-else-operator (ITE-operator)* introduced by Brace [7]. All operations in Table 2.5 can also be expressed in terms of the ITE-operator. For a more detailed description of these operations see [7].

	Operation	Complexity
reduce	makes OBDD canonical	$O(\|F\|)$
apply	$f' = f(f_1, f_2)$	$O(\|F1\| \cdot \|F2\|)$
restrict	$f' = f(x_1, \dots, x_i = V, \dots, x_n), V \in \{0, 1\}$	$O(\|F\|)$
compose	$f'_1 = f_1(x_1, \dots, x_i = f_2, \dots, x_n)$	$O(\|F1\|^2 \cdot \|F2\|)$

Table 2.5: Operations on OBDDs

If an OBDD is to be built for a multi-level combinational network this can be accomplished using the *apply* operation. Starting at the primary inputs of the circuit as a first step OBDDs are built for the circuit nodes adjacent to the primary inputs. Moving towards the primary outputs using the *apply* operation, OBDDs are built step by step for the internal circuit nodes until the primary outputs are reached. *Hashing* plays an important role in this procedure aiming at building new OBDDs making maximum use out of previously computed OBDDs. Alternatively, the *compose* operation could be used to build an OBDD for a multi-level circuit. In this case, we have to move backwards from the primary outputs to inputs. Since *compose* is more complex than *apply*, as shown in Table 2.5, the *apply* operation generally would be preferred.

Variable Ordering is important in making OBDD-based methods efficient. For many practical functions the ROBDD size is highly sensitive to the variable ordering. Therefore, many heuristics have been developed to determine and change the variable orderings for OBDDs [61], [36], [43], [80]. Unfortunately, for some functions like integer multiplication the size of the ROBDD is exponential in the number of input variables no matter what variable ordering is chosen [14]. Therefore, some circuits like multipliers are not amenable to BDD-based techniques.

Chapter 3

Reasoning in Multi-Level Networks

This chapter develops the Boolean techniques forming the basis for solving the formal verification tasks considered in later chapters. We will introduce the concept of implicants in multi-level networks and describe network transformations that can be obtained based on this concept. A major part of the chapter is devoted to so-called AND/OR reasoning techniques that allow us to derive such implicants. It will be described how implicants can be used for network optimization and for Boolean representations of sets which is important for sequential verification.

3.1 Introduction

Consider the problem of checking the equivalence of two combinational circuits. This gives us a first impression of some of the difficulties also encountered with sequential equivalence checking. Studying this problem also demonstrates the need for exploiting structural circuit properties in formal verification algorithms.

Two combinational circuits A and B are called equivalent if they produce the same output vector for every possible input vector. The traditional approach to prove the equivalence of two circuit descriptions is to express the function of each circuit in a canonical form and then to verify that both representations are identical. Graph representations of Boolean functions such as *Reduced Ordered Binary Decision Diagrams (ROBDDs)* [14] are presently the most popular canonical circuit representation. If we have succeeded in representing each output function of both circuits by an ROBDD equivalence checking reduces to the problem of checking whether or not the ROBDDs of corresponding output functions in circuit A and B are isomorphic. Examples for approaches of this kind can be found in [35, 61].

The advantage of using such canonical forms is that the equivalence check itself is simple. The difficulty of verifying the equivalence of two designs is replaced by the difficulty of constructing the canonical forms. In many cases, this task can be accomplished very efficiently, because much research effort has already been put into the development of powerful methods for constructing and manipulating these representations (e.g., [7, 80]). Specialized data structures and algorithms have been developed to make verification feasible for certain classes of circuits. *Ordered functional decision diagrams (OFDDs)* [46] and *ordered Kronecker functional decision* *diagrams (OKFDDs)* [31] are an attempt towards more compact canonical representations by allowing also other decomposition types (Reed-Muller, positive and negative Davio).

The disadvantage of using canonical forms, however, is that in certain cases their size complexity exhibits its worst-case exponential behaviour. For a large class of circuits, the size of an ROBDD is heavily dependent on the order in which the input variables are evaluated. For a certain variable order, the BDD may be small, for another its size may be exponential in the number of input variables. Even worse, for another class of circuits (e.g., multipliers), *every* variable order yields a BDD of exponential size. This "BDD blow-up" makes it impossible to represent practical circuits of this category. By designing specialized data structures for certain kinds of circuits the problem can sometimes be avoided. For example, the verification of arithmetic circuits which is difficult for BDDs has been effectively approached by the introduction of *word-level decision diagrams* such as *multi-terminal binary decision diagrams (MTBDDs)* [23, 5], *binary moment diagrams (BMDs)* [13] and related forms such as EVBDDs [57], *BMDs [13], K*BMDs [30]. For general-type circuitry, however, it is always possible that constructing a canonical representation of a circuit function fails due to lack of memory.

In order to overcome these limitations, verification methods [51, 8] were developed using structural techniques originating in the field of automatic test pattern generation (ATPG, see, e.g., [1]). These methods operate directly on a gate netlist description of the circuits, which is a *non-canonical* representation of its functionality. The equivalence checking problem is expressed by using a special construction combining the designs under comparison (Fig. 3.1) which was called *miter* by Brand [8].

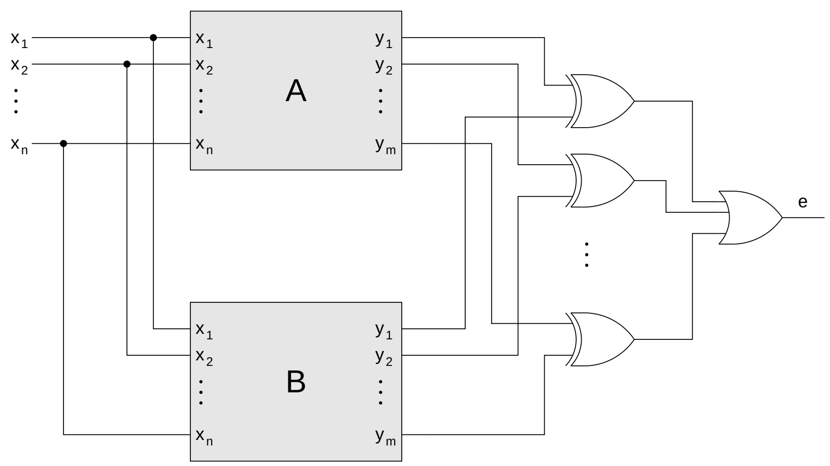


Figure 3.1: Miter

Corresponding inputs of both circuit descriptions are connected. Corresponding output pairs each feed an XOR (exclusive or) gate. All XOR gates feed a common OR gate which produces the (only) output function e of the miter. The designs are equivalent if there exists no combination of value assignments at the primary inputs which produces a logic 1 at the output e. If there is such a combination of value assignments, it is called a *distinguishing vector* or *(combinational)* *counter example* for the two designs. In other words, checking equivalence of the designs means checking *satisfiability* of the Boolean function implemented by gate e in Fig. 3.1.

Solving satisfiability is a very complex problem. If the circuits are indeed equivalent, the satisfiability solver must (implicitly) enumerate all possible combinations of value assignments at the primary inputs in search of a distinguishing vector. This is usually infeasible even for small designs.

However, the use of a structural representation of the designs allows us to simplify the verification task by using problem-specific knowledge. In many practical equivalence checking scenarios, the two designs to be compared have a high degree of structural similarity. The reason for this is that the two circuit descriptions have been generated in a similar way, or that one has incrementally evolved from the other. For example, often, a designer introduces manual changes to the design (called ECs – engineering changes), and the verification task is to check whether the logic function of the circuit has not changed. Other applications of an equivalence checker are to verify that the logic functionality of a circuit has been preserved by an incremental transformation as performed by a synthesis tool. These incremental steps usually maintain the overall circuit structure by preserving many internal subfunctions of the design.

Most modern combinational equivalence checking tools make use of the fact that often two circuits to be compared are structurally similar. In [51] this is done by identifying implications between signals in circuit A and B. The implications are stored at the respective nodes and are used in subsequent reasoning steps. In [8] signals in subcircuit A are identified which can be substituted for signals in subcircuit B, while taking into account *observability don't-cares* [67]. Storing implications or making physical connections between the two subcircuits install "reasoning short cuts" for the satisfiability solver. This makes it possible to break the verification procedure down into smaller steps, so that the equivalence of subfunctions of the two designs at a certain level of logic is proved before proceeding to the next level.

Recent works combine OBDD techniques with the use of structural information. The miter circuit is partitioned based on network cuts at equivalent or potentially equivalent internal signals. The subcircuits of the partitions are represented by *local* BDDs. Equivalence is proved by verifying the equivalence of the subcircuits. Examples for approaches of this kind are [76, 44, 62, 50].

For many problems of realistic size the use of structural information is the only way to avoid the exponential worst-case behaviour of the satisfiability solving algorithms underlying the verification techniques. By using structural similarities we can make use of the fact that the Boolean function e of the miter circuit is independent of the Boolean functions of the individual designs A and B. In fact, output e implements the characteristic function of the set of distinguishing vectors of both subcircuits. In the case of combinational equivalence this set is empty so that e is equal to constant 0 and the miter circuit is a very complex implementation of this very simple logic function. It is obvious that the complexity of checking equivalence is more related to the structure of the representation of the verification problem than to the complexity of the logic functions implemented by the original designs under comparison. Looking at the miter from a synthesis perspective, clearly, the goal must be to *optimize* the miter prior to invoking the satisfiability solver [87, 55]. Performing substitutions as in [8] can be seen as a simple optimization process in this sense. Common subexpressions of designs A and B are detected and shared. Provided the circuits are equivalent eventually every output function of circuit B can be replaced by the corresponding output of A, making the satisfiability check at e trivial.

Substituting equivalent signals, however, provides only a restricted means for optimizing the miter. In principle, any modern synthesis method can be used for this purpose. In this work, we consider circuit transformations which are based on the concept of *implicants* [86] in multi-level combinational networks. This approach will be described in detail in the following sections. Since it can exploit structural similarites between designs and finds more general relationships than mere signal equivalences, it can be seen as a natural extension of the approaches in [51] and [8]. Furthermore, it naturally permits exploiting observability don't-care conditions. The following issues will be adressed in the sequel:

1. The concept of an implicant is basic to two-level circuit theory. How can it be generalized to have a meaning in *multi-level circuits* (section 3.2.1)?

2. How can such implicants be used for transforming a multi-level circuit (section 3.2.2)?

3. How can these implicants be calculated (section 3.3)?

3.2 Implicant-Based Network Transformations

This section introduces the concept of an *implicant* in a multi-level combinational network and shows how circuit transformations can be performed using this concept. In the context of verification, implicant-based network transformations can be used for the optimization of the miter as described in the previous section. It will be shown that arbitrary circuit transformations are possible.

The conventional notion of an *implicant* is closely associated with the theory of two-level logic minimization. Two-level minimization techniques have an impact in all areas where two-level forms are used as a basic means of representing a logic function. Often, two-level forms are directly implemented as *programmable logic arrays (PLAs)* which are used also in macro-cell VLSI design styles or in most types of *programmable logic devices (PLDs)*.

For the optimization of *multi-level* circuits, however, the concept of an implicant to-date has not played a significant role. Instead, in the past a variety of methods have been developed that view the problem in a different way. The earliest systematic approach is known as functional decomposition [4, 27, 79]. It has recently found new applications in the synthesis of *look-up table (LUT)* based field programmable gate arrays (FPGAs). Among today's approaches to multi-level optimization, the techniques pioneered by Brayton et al. [10] have had the greatest impact. They are based on modeling a multi-level circuit by a graph called a *Boolean network* as defined in Definition 2.4 (page 14). The nodes of this graph are either primary inputs and outputs or arbitrary logic functions expressed by variables associated with the direct predecessors of a node. The optimization operations can be categorized into *Boolean* and *algebraic* techniques. The latter manipulate the local Boolean functions according to the rules of polynomial algebra, neglecting some properties of Boolean algebra. This sacrifices some optimization quality to the benefit of deriving a certain set of network transformations very quickly. The functions associated with the internal nodes of a Boolean network are typically represented in two-level form. Note however, that although the Boolean techniques of [10] apply sophisticated two-level minimization algorithms to individual nodes of a Boolean network, these techniques cannot be understood as a generalization of the Quine-McCluskey scheme to multi-level circuits.

In the following section we propose a new definition of an implicant that contains the conventional two-level concepts as a special case but generalizes to multi-level circuits so that also *multi-level* optimization and verification algorithms can be formulated based on implicants.

3.2.1 Implicants in Multi-Level Combinational Networks

In order to obtain a notion of an implicant being useful in multi-level networks, we only need a small extension to the classical concept for two-level circuits. We drop the restriction that the literals of a product term must belong to primary input variables. Instead, we define a literal to be an arbitrary variable or its complement in a *multi-level* combinational network (see page 15) and consider product terms (as defined in Section 2.4.2) being composed out of such literals.

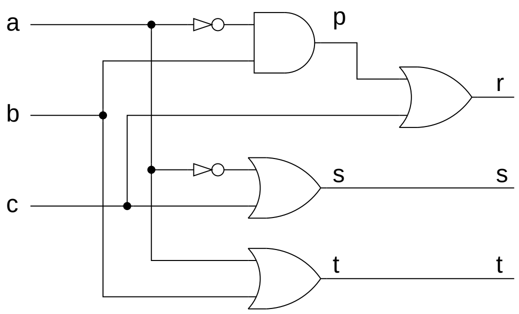
We propose a unified treatment of SOP- and POS-representations. In a multi-level circuit, which is composed of subsequent stages of sums and products of intermediate network variables, it is necessary that methods for manipulating intermediate functions give no preference to either SOP- or POS-type representations. Note that implicants and implicates are closely related concepts. A sum term, e.g., $(a + c)$, in a POS-type representation of a function y is called an implicate of y. This sum term can be converted into a (negated) product term p using DeMorgan's law: $a + c = \overline{\overline{a} \cdot \overline{c}} = \overline{p}$. Any input vector producing $p = 1$ implies $y = 0$. We therefore call p a *0-implicant* of function y. This allows us to treat both implicants and implicates in the same way and leads us to the following definition:

Definition 3.1 (Implicant) *A* 1-implicant (0-implicant) *for a given function f in a combinational network C is a product term p such that f assumes the value $1(0)$ for every set of value assignments at the primary inputs of C for which p assumes the value 1. An* implicant *for a function f is a product term which is either a 1-implicant or a 0-implicant of f.*

> **Example 3.1** Consider the multi-level circuit in Figure 3.2. function r is implemented in a classical two-level sum-of-products (SOP) form. It is immediately obvious that $\overline{a}b$ is a prime implicant of r. Input c is another prime implicant of r consisting of only a single literal.
>
> If we allow that the literals of the implicants do not have to belong exclusively to a primary input but can belong to arbitrary nodes of the network, additional prime implicants can be determined. Note by examining the truth table in Fig. 3.2 that all combinations of value assignments at the primary inputs a, b and c which produce $s = 1$ and $t = 1$ simultaneously, also cause $r = 1$. Therefore, product st is also a prime 1-implicant of r.
>
> Function s can be viewed as a classical SOP. It is the sum of two (single-literal) 1-implicants, $\overline{a}$ and c. It can also be viewed as a product of sums (POS). In this case,

it is a degenerated one-term "product" of a two-literal sum term, $(\overline{a} + c)$. Using DeMorgan's law, we can rewrite this term as $\overline{(a\overline{c})}$. The product term $a\overline{c}$ is a prime 0-implicant of function s.

Dropping the restriction that literals may only belong to primary inputs, we can again find more prime implicants. Another prime 1-implicant of s is the single-literal product term r. Any combination of value assignments at the primary inputs producing $r = 1$ also produces $s = 1$.



a	b	c	p	r	s	t	$s \cdot t$
0	0	0	0	0	1	0	0
0	0	1	0	1	1	0	0
0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	1
1	0	0	0	0	0	1	0
1	0	1	0	1	1	1	1
1	1	0	0	0	0	1	0
1	1	1	0	1	1	1	1

Figure 3.2: Circuit example for multi-level implicants and associated truth table

When minimizing a two-level circuit, the Boolean functions we are concerned with are those implemented by the primary outputs. Logic functions of internal nodes are not considered individually. These nodes make up the inverters, product and sum terms of which the output functions are composed.

When dealing with multi-level circuits, on the other hand, it makes sense to also associate a logic function with an *internal* network node. Whereas an internal node in a two-level circuit can only implement a simple product or sum term, it can implement a complex sub-function in a multi-level network. An internal node in a multi-level network is commonly viewed as being embedded in a local environment consisting of the nodes feeding its input variables and the nodes that have its output function as input.

Depending on its functionality, the environment is not always able to produce all possible patterns at the inputs of a node. For the set of impossible patterns, the output values produced by the node are irrelevant. This set is called *controllability don't-care set (CDC)* (see, e.g., [67]).

Also, it is possible that the environment does not always make use of the output values produced by an internal node y. For some input patterns, the output values of y may be irrelevant for the values of the logic functions at the primary circuit outputs, i.e., for these input patterns none of the circuit outputs would change its value if we inserted an inverter at y. In this case, y is said to be *unobservable* for these inputs. The set of input patterns for which a node y is unobservable is called *observability don't care set (ODC(y))* [67].

The logic function f implemented by an internal node can be replaced by a different function g, as long as this replacement is not observable at any primary output of the circuit. This is the case if f and g produce different values only for the input patterns in the observability don't-care set. A function g that fulfills this requirement is called a *permissible* function according to Muroga [68].

Definition 3.2 (Permissible function) *In a combinational network a function g is called* permissible *at a node with function f, if the function $C(x)$ of the combinational network does not change when f is replaced by g.*

Since we are extending the notion of implicants from two-level circuits to multi-level circuits it seems wise to also take into account the concepts of observability don't-cares and permissible functions. This can be done in the following way. In addition to the implicants of a function f at a node we can identify implicants of permissible replacement functions g. We require that a product term p is an implicant of f for only those patterns for which f is observable at a primary output. For the non-observable input patterns we "don't care" about the value of the product term. Such an implicant is called a *permissible* implicant.

Definition 3.3 (Permissible implicant) *For some node f in a combinational network C, a product term p of some node variables of C is called a* permissible 1-implicant *for f, if and only if the following condition holds:*

If $p = 1$ then $f = 1$ or f is not observable at any primary output of C.

Similarly, p is called a permissible 0-implicant *of f, if and only if the following condition holds:*

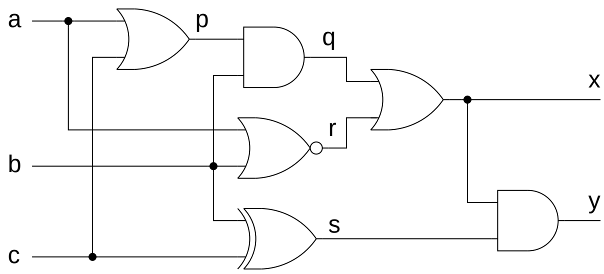
If $p = 1$ then $f = 0$ or f is not observable at any primary output of C.

A permissible implicant is called prime *if there is no permissible implicant that covers it and has less literals.*

Example 3.2 Figure 3.3 shows a multi-level circuit with three inputs and two outputs. Consider the internal logic function implemented by node s. It is $s = \overline{b}c + b\overline{c}$. The product term $t = p\overline{b}$ can be expressed in terms of primary input variables as $t = \overline{b}a + \overline{b}c$. Obviously, t is not an implicant of function s. However, it is a *permissible* implicant of s. This can be easily verified by examining the truth table for this circuit. Note that for s to be observable at the output of AND gate y, signal x must be set to 1. In all cases where $x = 1$ and $p\overline{b} = 1$, it is also $s = 1$.

3.2.2 Network Optimization

This section addresses the issue of how implicants as defined above can be used for transforming a multi-level circuit with the goal of optimizing it. Since we have generalized the concept of an implicant from two-level to multi-level circuits in the last section, it seems promising to investigate whether it is also possible to generalize some algorithmic approaches for two-level minimization to the multi-level case.



a	b	c	p	q	r	x	s	$t = p \cdot \overline{b}$
0	0	0	0	0	1	1	0	0
0	0	1	1	0	1	1	1	1
0	1	0	0	0	0	0	1	0
0	1	1	1	1	0	1	0	0
1	0	0	1	0	0	0	0	1
1	0	1	1	0	0	0	1	1
1	1	0	1	1	0	1	1	0
1	1	1	1	1	0	1	0	0

Figure 3.3: Circuit example for permissible implicants and associated truth table

The first method for minimizing a two-level circuit reported in literature is the Quine-McCluskey procedure [73, 63]. It is an exact method, and it has been under improvement until today. Recent advances in exact two-level minimization can be found in [28, 81, 24, 25, 65]. Except for [65], all exact methods are based on calculating a complete set of prime implicants of the circuit function and then solving a unate covering problem to find a minimal set of prime implicants covering the function.

Despite its theoretical importance, an exact minimization approach is not feasible for many practical circuits, because the number of prime implicants can grow exponentially with the number of input variables, yielding a huge prime implicant table and making the covering problem intractable. Heuristic procedures, such as the UC Berkeley tool *ESPRESSO* [11] avoid calculating all prime implicants. Instead, they start with an initial set of implicants representing a cover of the function to be minimized. Then, (in a simplified view), the following steps are iteratively repeated until no further improvement can be achieved:

1. New (not necessarily prime) implicants of the function are calculated and added to the cover. (This introduces redundancy to the cover, because the minterms covered by an added implicant are already covered by other existing implicants).

2. Redundant implicants and literals are removed such that, according to the underlying cost criterion, an improvement over the previous situation is achieved. (The cover is made prime and irredundant again).

Example 3.3 Figure 3.4 shows the Karnaugh map for the logic function $f = ab + a\overline{c} + \overline{a}\overline{b} + \overline{a}c$. If we add the implicant $p = bc$ to the cover of the function, the implicants ab and $\overline{a}c$ become redundant (see Fig. 3.5), so that we can remove them and obtain the simpler representation $f = \overline{a}\overline{b} + bc + a\overline{c}$.

The general idea in heuristic two-level minimization is to "stir up" the set of implicants representing a function by "throwing in" appropriate new implicants and removing "old" implicants that become redundant.

How can we use these ideas for heuristic multi-level circuit optimization? The approach to be proposed here follows the same iterative two-step methodology as in the two-level case. We

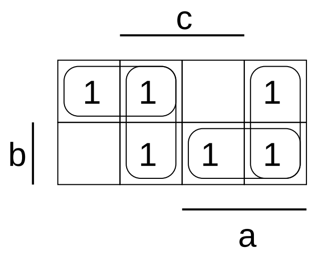


Figure 3.4: Initial cover

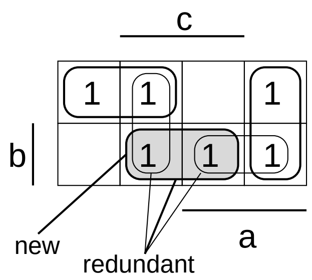


Figure 3.5: Cover with added redundant implicant $p = bc$

introduce new implicants to the cover of a network node and then make it prime and irredundant again.

How is an implicant p added to the cover of a function f? In the two-level case of a sum-of-products form adding an implicant means to form the disjunction of the existing cover f with the implicant: $f' = f + p$. The new cover is functionally equivalent to the old cover. Analogously, for a product-of-sums expression, an implicate q is added by forming the conjunction of the existing cover f with the implicate: $f' = f \cdot q$. Again, the new cover f' is equivalent to the old cover. We can easily extend this to our notion of multi-level network implicants. Distinguishing between 0- and 1-implicants, we obtain the following lemmas:

Lemma 3.1 *Let f be a function associated with a node in a multi-level combinational network. The disjunction*

$$f' = f + p$$

is equivalent to f if, and only if, p is a 1-implicant of function f according to Def. 3.1.

Proof: *obvious*

This is easy to see by noting that $f' = f + p \neq f$ iff $p = 1$ and $f = 0$. This case is impossible because $p = 1$ implies $f = 1$. An analogous lemma holds for 0-implicants:

Lemma 3.2 *Let f be a function associated with a node in a multi-level combinational network. The conjunction*

$$f' = f \cdot \overline{p}$$

is equivalent to f if, and only if, p is a 0-implicant of function f according to Def. 3.1.

Proof: *obvious*

The above lemmas provide us with a way of modifying the cover of a function implemented by a node in a multi-level network. They are our basic instrument in the first step of the heuristic optimization approach outlined above.

For the second step of removing redundant implicants and literals in a multi-level network function we make use of the standard technique of ATPG-based *redundancy elimination* [1]. Redundant signals in a multi-level combinational circuit correspond to *untestable stuck-at faults* [1].

The circuit structure can be simplified by replacing a signal y which has an untestable fault, y stuck-at-v, $v \in \{0, 1\}$, by the constant signal v. Note that ATPG-based redundancy elimination, when applied to a two-level circuit, is a method to make a cover prime and irredundant.

When removing redundancies we have to be careful about which signals are removed: since any implicant which is added to the cover of a function is redundant by construction we have to exclude it from the target fault list. Otherwise, redundancy elimination simply reverses the addition of the implicants to the cover we have just achieved.

These two steps – adding implicants to the cover of a network node function and removing redundancies using ATPG – allow us to transform a combinational network. Both steps alter the network structure, not the function. They produce networks which are structurally different but functionally equivalent to the original network. An important question coming to mind is what kinds of transformations can be performed with this methodology. The following theorem answers this question.

Theorem 3.3 *Let f_i be a node in a combinational network C_i. Further, let p_i be a multi-level network implicant according to Def. 3.1, such that*

1. *the transformation of node f_i into f_{i+1} given by*
   (a) $f_{i+1} = f_i + p_i$ *if p_i is a 1-implicant of f_i*
   (b) $f_{i+1} = f_i \cdot \overline{p_i}$ *if p_i is a 0-implicant of f_i*

   followed by
2. *redundancy removal (with appropriate fault list)*

generates a combinational network C_{i+1}. For an arbitrary pair of equivalent combinational networks C and C' there exists a sequence of equivalent combinational networks $(C_1, C_2, \ldots, C_k)$ such that $C \equiv C_1$ and $C' \equiv C_k$.

Proof: *see Appendix A, page 133*

Theorem 3.3 is based on Theorem 3.1 of [55]. It states that using implicant-based transformations as described, theoretically *all* networks implementing a given function can be derived. Beginning with an arbitrary network structure C implementing the circuit function, there exists a sequence of transformations composed of the two substeps discussed above, such that a functionally equivalent network C' with different structure is obtained. For instance, network C may be a given combinational network and C' a functionally equivalent network which is optimal with respect to a given cost function. This means that any circuit transformation as performed by traditional logic synthesis methods such as functional decomposition, kerneling, division, transduction, etc., can also be described in terms of transformations based on multi-level network implicants. This illustrates the general nature of the proposed approach and the value of the proposed notion of implicants in multi-level circuits.

The set of transformations to be performed at an internal network node can be further extended. To guarantee equivalence of the transformed network to the original network, it is not necessary to perform only equivalence transformations of internal network nodes. By allowing the replacement of an internal function f_i by a *permissible function* g_i we can exploit additional degrees of freedom given by *observability don't cares* as explained in the previous section. Replacing a node function f_i by a permissible function g_i which is not equivalent to f_i has no effect on the logic functions of the primary outputs, i.e., equivalence of the networks before and after the transformation is guaranteed. Permissible functions can be created by adding permissible implicants to the cover of a function as stated in the following two lemmas.

Lemma 3.4 *Let f be a function associated with a node in a multi-level combinational network. The disjunction*

$$g = f + p$$

is a permissible function for f if, and only if, p is a permissible 1-implicant of function f according to Def. 3.3.

Proof: *obvious*

Lemma 3.5 *Let f be a function associated with a node in a multi-level combinational network. The conjunction*

$$g = f \cdot \overline{p}$$

is a permissible function for f if, and only if, p is a permissible 0-implicant of function f according to Def. 3.3.

Proof: *obvious*

According to Theorem 3.3, arbitrary network modifications can already be obtained without permissible implicants. However, with permissible implicants less transformation steps may be necessary because these steps can be "bigger", hence improving performance and quality of the optimization method.

Note that Theorem 3.3 only states the existence of a sequence of implicant-based transformations yielding the desired circuit structure for a given function. It does not, however, tell how we can find these implicants and which of these implicants have to be used. It is the task of appropriate heuristics to select those implicants which help our goal of optimizing the circuit. Up to this point, we have only examined how multi-level network implicants can be used to transform a circuit. The following sections will introduce a method to calculate multi-level network implicants and will also give some hints on the selection of implicants in order to optimize a circuit.

3.3 AND/OR Reasoning Graphs

This section introduces *AND/OR reasoning graphs* for determining prime implicants in multi-level networks [86]. These graphs are a representation of a special kind of search called *AND/OR enumeration* which is of very different nature compared to the search techniques conventionally employed in logic synthesis. This motivates some basic considerations before we return to the problem of identifying implicants in multi-level networks.

3.3.1 OR search versus AND/OR search

Every search process can be viewed as a traversal of a directed graph. Standard literature (e.g., [77]) distinguishes between two basic types of search graphs. In the simpler case to be considered, the graph is a so-called *OR graph*. A node in the OR graph represents a possible move or decision that can be made at the current state of the search process. A solution is found by traversing the graph following certain strategies guided by some heuristics exploiting problem-specific knowledge. Typical strategies are known as *depth-first* search, *breadth-first* search or *best-first* search.

For some problems, however, it is useful to allow graphs with *two* types of nodes, AND nodes and OR nodes, that represent a different type of search process. If at a given state of the search a certain move is made this may lead to *several* new problems that *all* have to be solved. Such AND/OR graphs are the basis for many search methods employed in the field of automatic theorem proving with predicate logic and are used in proof-by-refutation strategies. For a description of general problem-solving techniques in computer science and for more information on the concepts of OR graphs and AND/OR graphs, the reader my refer to a standard text book, e.g., [77].

In our context we are interested in search techniques for determining implicants of Boolean functions implemented by multi-level combinational networks. Determining a 1-implicant p for a Boolean function f means searching for a conjunction of literals (product) such that the function takes the value 1 if the product is 1. The implicant represents a logical condition for f being satisfiable. In general, any method that determines implications or implicants is also a method that solves satisfiability.

In the field of design automation, many methods that solve satisfiability have already been developed. Most methods rely on exploring the finite Boolean space defined by the set of all combinations of value assignments at the input variables by some sort of enumeration of this set.

For example, a common search scheme to solve satisfiability or related problems like test generation is decision tree-based backtracking. In search of a satisfying input vector, decisions regarding value assignments at input variables are made. If subsequent steps prove that a decision was wrong, it is reverted and an alternative is chosen. The decision tree keeps track of decisions and alternatives. At a decision point in the search, any one of the alternatives which leads to a solution of the problem is sufficient. Therefore, decision tree-based backtracking is an OR type search and the decision tree itself is an OR graph. The way in which this method explores the search space is sometimes called *implicit enumeration* as opposed to *explicit enumeration*, because not all possible combinations of value assignments need to be explored.

An example for an explicit enumeration of the search space is Shannon's expansion of a Boolean formula. If a function is expanded with respect to every input variable, every combination of value assignments at the inputs is examined. Representing this exhaustive simulation as a graph yields a Shannon tree. This tree can be reduced by sharing isomorphic subtrees so that we obtain a binary decision diagram (BDD). As a method to solve satisfiability, explicit enumeration is, again, an OR type search. The corresponding graph (Shannon tree, BDD) is an OR graph. In this case, the graph is also a representation of the Boolean formula for which it was derived.

All these concepts to solve satisfiability and related problems have in common that they can be interpreted as OR trees and not as AND/OR trees. To better understand this important point the difference between OR trees and AND/OR trees is illustrated by the following example.

Example 3.4 Let us consider Robinson Crusoe's [29] situation after he was shipwrecked and washed ashore a small island. Robinson analyzes his situation and starts thinking how he can leave the island again. We will now follow his reasoning and show how this leads to the AND/OR tree depicted in Fig. 3.6.

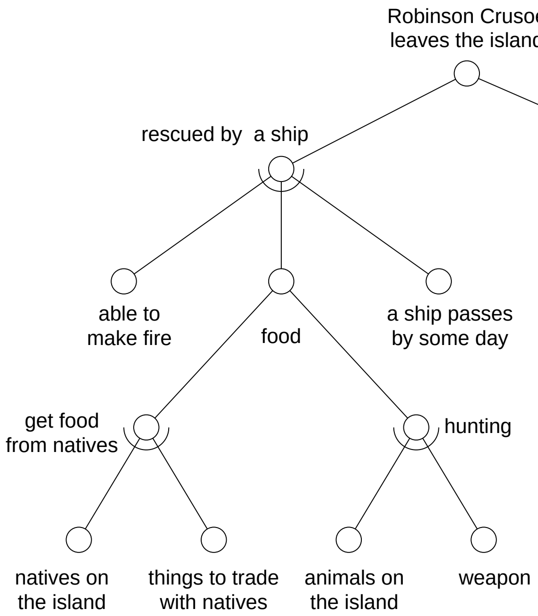


Figure 3.6: Example of AND/OR tree

Robinson can see only two possibilities to leave the island: either he waits for a ship to pass by and pick him up or he builds a boat himself. From the root node representing the assumption that Robinson will leave the island there are two edges each leading to a node representing one of the two possibilities. Since one of the two possible outcomes is sufficient for him to come back home, the root node is an OR node. If he decides to build a boat he will need some wood and appropriate tools. Either prerequisite is not sufficient by itself for making a vessel, therefore, the node "build a boat" in the AND/OR tree is an AND node with two sucessors, "wood on the island" and "tools". (AND nodes are commonly marked by an arc). If, on the other hand, he waits for a ship to come by some day, he will need food for the wait and some means for attracting attention when the ship arrives, for example a fire to send smoke signals. In the AND/OR tree, these necessary conditions are depicted as successors to an AND node labelled "rescued by a ship". Also, finding food on the island is a problem of its own. He sees two possibilities: trade with natives or go hunting, and one of them must be realized in order to save him from starvation. Node "food" therefore is an OR node in the AND/OR tree.

Robinson has analyzed his situation by determining the necessary conditions for leaving the island. As can be seen, whether he will actually succeed depends on a number of “variables”. In principle, each of these variables can assume the value “yes” or “no”, depending on how lucky he is and what kind of island he is on. Robinson has determined what “values” of these “variables” would be necessary for different scenarios of leaving the island. Since these requirements do not contradict each other the situation is not hopeless and he has a chance of leaving.

Of course, in order to decide whether or not he can leave the island it is also possible to check the “variables” one after the other. This leads to representing the problem as an OR tree. In Figure 3.7 the corresponding OR tree is depicted. There are several ways to build an OR tree for the AND/OR tree of Fig. 3.6. The tree shown in Figure 3.7 is built in analogy to building OBDDs for Boolean functions.

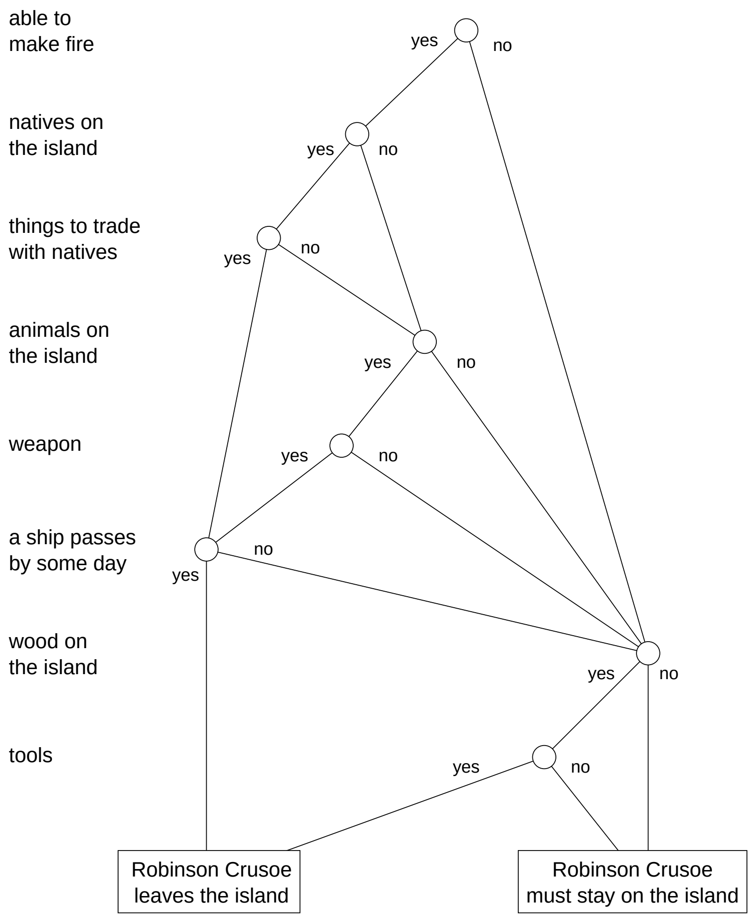


Figure 3.7: Example of an OR tree for the AND/OR tree of Fig. 3.6

Note that the OR nodes in the OR tree of Figure 3.7 correspond to the OR leaves of the AND/OR tree of Figure 3.6. The order in which the “variables” appear has a strong influence on the size of the tree. It is interesting to observe that, at least in this example, the structure of the AND/OR tree suggests a good order. The OR nodes in the OR tree follow the same order in which the leaves of the AND/OR tree appear from the left to the right. This suggests that analyzing AND/OR trees for Boolean functions may also lead to useful methods for OBDD variable ordering. However, in this thesis, this aspect is not further considered.

There is another important reason why AND/OR trees are of interest to us in the field of design automation. They turn out to be more suitable for *systematic reasoning* than OR trees. For example, from the AND/OR tree of Figure 3.6 the following implication can be derived in a simple way:

(no natives on the island) AND (no weapon) AND (no tools)
$\longrightarrow$ (must stay on the island)

In this section it will be explained in detail how such implications can be extracted from an AND/OR tree. In principle, this implication can also be derived from the OR tree in Figure 3.7. Note that all paths from the root node to one of the terminal nodes that pass through the “no” branches of “natives on the island”, “weapon” and “tools” lead to the terminal node “Robinson Crusoe must stay on the island”. This may be easy to see in this small example but for larger trees such an analysis becomes intractable. The problem is that we cannot exclusively consider these three nodes. We also have to go through nodes “able to make fire”, “animals on the island” and “wood on the island”. In fact, it does not matter whether there are animals on the island if Robinson has no weapon. This information is obvious in the AND/OR tree, however, moving from the top to the bottom in the OR tree we cannot skip the node “animals on the island” although this node does not have any influence on the result if Robinson does not have a weapon.

Note that any Boolean expression can be understood as an AND/OR tree. However, such a general AND/OR tree does not decide whether the implemented Boolean function is satisfiable. As mentioned, this problem is usually solved by resorting to an OR tree-based enumeration. The AND/OR trees we present here are produced by a special type of AND/OR enumeration that decides satisfiability. Solving satisfiability using this type of search requires a totally different way of stepping through the circuit and its variables compared to conventional backtracking methods.

The differences between the two searching schemes are of great practical interest in the field of design automation. As illustrated in the above example, OR search techniques are hard to use for *systematic reasoning*. Specifically, for some Boolean statement A we would like to derive some statement B that is true if A is true, i.e., $A \rightarrow B$. Previous representations of Boolean functions are not well suited for this kind of task. For example, given a statement A, a BDD-based approach cannot *derive* or *imply* statement B, it can only *check* if $A \rightarrow B$ is true when both A and B are given. By way of contrast, as will be shown in section 3.4 , AND/OR reasoning techniques can *determine* implications and implicants in multi-level networks.

3.3.2 AND/OR enumeration in multi-level circuits

We now formally introduce the algorithm for AND/OR enumeration in multi-level combinational networks. Determining multi-level network implicants based on AND/OR enumeration as described here is a generalization of *recursive learning* [53], which is an algorithm for determining all necessary assignments for single stuck-at fault detection. Besides being of great interest in the field of automatic test generation, recursive learning has also lead to substantial contributions in logic synthesis and verification [52, 51, 55].

Both, the method for determining general multi-level network implicants described in this thesis and the recursive learning procedure are based on the same AND/OR enumeration technique. Both methods extract information by *monitoring* this search. Recursive learning finds assignments which are necessary for exciting a fault and propagating the faulty signal to the circuit outputs. The method described here extracts implicants for a given node in a multi-level network from the AND/OR tree corresponding to the search. Actually, the necessary assignments or *implications* determined by recursive learning can be seen as special cases of network implicants: they correspond to "product terms" consisting of a *single* literal. Recursive learning can therefore be seen as a special case of AND/OR graph-based implicant calculation: it can find all *single-literal* implicants of the function implemented by a given network node. This is commonly referred to as "performing implications" in the testing literature.

AND/OR enumeration is performed by injecting and reversing signal values in a combinational network and by evaluating their logical consequences using event-driven implication techniques. This process is very time-critical and must be implemented very efficiently, for example using the sophisticated data structures proposed in [90]. Depending on the problem to be solved (simulation, satisfiability, test generation etc.), implications are based on different logic alphabets. The most simple alphabet is $B_2 = \{0, 1\}$ which is, for example, used for functional simulation of fault-free circuits. In order to describe the faulty behaviour of a circuit, Roth's D-calculus [78] has become widely accepted. In Roth's notation a signal is assigned the logic value D if it assumes 1 in the fault-free and 0 in the faulty circuit. In the opposite case, if the signal is 0 in the fault-free and 1 in the faulty circuit, it is denoted by $\overline{D}$. For logic values being equal in the fault-free and faulty cases, namely 0 or 1, the signal value is denoted 0 or 1, respectively. With these notations we obtain the logic alphabet $B_4 = \{0, 1, D, \overline{D}\}$ and it can be verified that B_4 together with the operations of disjunction, conjunction and negation forms a Boolean algebra. For the algorithmic description of making implications, it is of advantage to introduce a fifth logic value, X, describing the case where no unique logic value has been assigned. This value is usually referred to as the *don't care* or *unknown* value. Including X in B_4 results in the five-valued logic alphabet B_5 which to-date forms the basis for many modern ATPG algorithms. With the above definitions of logic values 0, 1, D and $\overline{D}$ and the operations of conjunction, disjunction and negation we obtain the truth values for this five-valued logic as shown in Table 3.1.

It is interesting to observe that B_5 and the shown operations do not form a Boolean algebra, because the associative law is violated. For example, $D \cdot (\overline{D} \cdot X) \neq (D \cdot \overline{D}) \cdot X$. The reason is the insufficient resolution of the don't care value.

Several logic alphabets with larger sets of values have been developed (e.g., [2, 69]), mostly

AND	0	1	X	D	$\overline{D}$
0	0	0	0	0	0
1	0	1	X	D	$\overline{D}$
X	0	X	X	X	X
D	0	D	X	D	0
$\overline{D}$	0	$\overline{D}$	X	0	$\overline{D}$

OR	0	1	X	D	$\overline{D}$
0	0	1	X	D	$\overline{D}$
1	1	1	1	1	1
X	X	1	X	X	X
D	D	1	X	D	1
$\overline{D}$	$\overline{D}$	1	X	1	$\overline{D}$

NOT	
0	1
1	0
X	X
D	$\overline{D}$
$\overline{D}$	D

Table 3.1: AND-, OR- and NOT-operation in 5-valued logic (*D-calculus*)

for application in test generation. In this thesis, we only consider the logic systems $B_2 = \{0, 1\}$, $B_3 = \{0, 1, X\}$ and $B_5 = \{0, 1, X, D, \overline{D}\}$, which are the most popular alphabets. The methods developed in this chapter, however, can also be applied to other logic alphabets. Logic alphabet B_5 will be used for so-called *D-AND/OR enumeration*, to be described in Section 3.5, which allows to determine permissible implicants according to Definition 3.3.

Making implications in a combinational network means analyzing each gate in order to find input or output signals whose values can be uniquely determined according to the function of the gate. In this process, *unspecified* signals become *specified*. These terms are defined as follows:

Definition 3.4 *For the logic alphabets B_2, B_3, B_5 the values 0, 1, D and $\overline{D}$ are called* fixed *or* specified*. The logic value X is called* unspecified*.*

Starting from an initial set of specified signals, an implication procedure iteratively analyzes the gates in a circuit to make other signals specified. (This is best implemented in an event-driven fashion.) Making *direct implications* according to [53] means to carry out all such implications until no further value assignments can be found by locally analyzing each gate. Figure 3.8 shows value assignments in a circuit. After direct implications have been performed, the value assignments have changed as shown in Fig. 3.9.

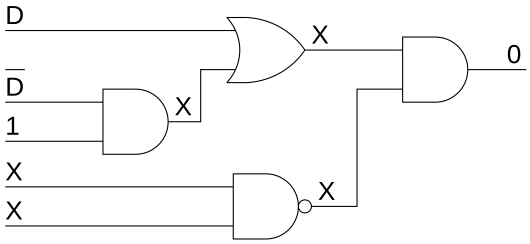


Figure 3.8: Signal values before direct implications

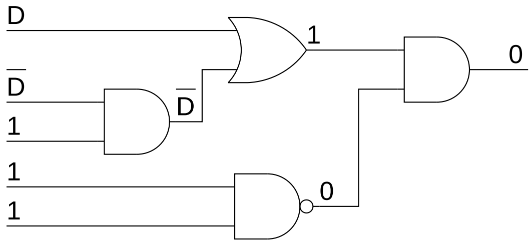


Figure 3.9: Signal values after direct implications

Note that the set of direct implications is a subset of all implications that are possible for a given set of initial value assignments. Implications which cannot be derived by local gate function evalation have been termed *indirect implications* [53]. Figures 3.10 and 3.11 show an example of an indirect implication proposed by Schulz et al. [83].

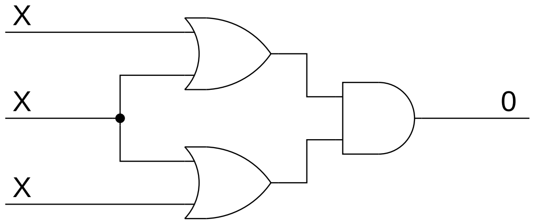


Figure 3.10: No direct implication possible

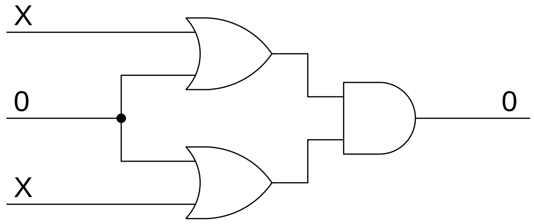


Figure 3.11: Signal values after *indirect* implications

For the formulation of the algorithm *and_or_enumerate()*, we need the following definitions from [53].

Definition 3.5 (Unjustified gate) *Given a gate g that has at least one specified input or output signal and the values at g are logically consistent. Gate g is called* unjustified, *if there are one or more unspecified input or output signals of g for which there exists a combination of value assignments that is logically inconsistent at g. Otherwise, g is called* justified.

Note that the notion of unjustified or justified gates only applies to gates with at least one specified input or output signal. If no signal is specified the gate is termed *unspecified*. The special case of an unjustified gate where the gate output is a specified signal is commonly referred to as *unjustified line* in test generation literature [1].

Definition 3.6 (Justification) *Let $f_1, f_2, \ldots, f_n$ be some specified input or output signals of an unjustified gate g and let $V_1, V_2, \ldots, V_n$ be logic values which specify them. The set of signal assignments, $J = \{f_1 = V_1, f_2 = V_2, \ldots, f_n = V_n\}$, is called* justification *for g if the combination of value assignments in J makes g justified.*

Fig. 3.12 depicts three examples of unjustified or justified gates in 3- and 5-valued logic. For the examples of unjustified gates, all possible justifications are given. Justifications for a gate are sets of value assignments which can be subsets of one another. This motivates to formulate the following definition:

Definition 3.7 (Complete set of justifications) *Let C_g be a set of m justifications for an unjustified gate g: $C_g = \{J_1, J_2, \ldots, J_m\}$. If there is at least one justification $J_i \in C_g$ for any possible justification J^* of g such that $J_i \subseteq J^*$, then set C_g is called a* complete set of justifications *for gate g.*

For a given unjustified gate, the derivation of a complete set of justifications is straightforward. In the worst case this set consists of all consistent combinations of signal value assignments representing a justification of the considered gate. Often though, the set can be smaller, as for the following example.

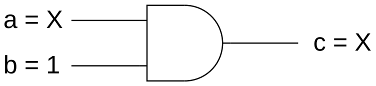
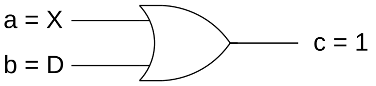
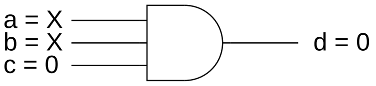
unjustified / justified gates	justifications
 unjustified, 3-valued logic	$J_1 = \{a = 1, c = 1\}$ $J_2 = \{a = 0, c = 0\}$
 unjustified, 5-valued logic	$J_1 = \{a = \overline{D}\}$ $J_2 = \{a = 1\}$
 justified, 3-valued logic	justified

Figure 3.12: Unjustified gates and justifications

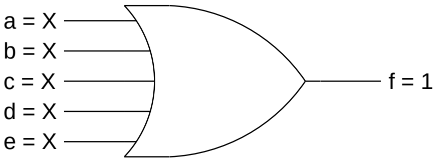


Figure 3.13: Example for complete set of justifications

Example 3.5 The following represents a complete set of justifications for the unjustified gate in Figure 3.13: $C = \{J_1, J_2, J_3, J_4, J_5\}$ with $J_1 = \{a = 1\}$, $J_2 = \{b = 1\}$, $J_3 = \{c = 1\}$, $J_4 = \{d = 1\}$, $J_5 = \{e = 1\}$. Note that for example the justification $J^* = \{a = 1,\, b = 0\}$ does not have to be in C because all assignments in J_1 are contained in J^*.

A complete set of justifications represents a minimal set of all possible "cases" of how an unjustified gate g can become justified. In order to explore all these possibilities, one only needs to enumerate the justifications in the complete set C_g. Every other justification is covered by at least one justification in C_g. This enumeration of justifications at an unjustified gate is an important part of the AND/OR search. In the AND/OR graph associated with the search, an unjustified gate corresponds to an OR node. The successors of this node are AND nodes each representing a justification for the unjustified gate.

For the enumeration process, it is necessary to keep track of value assignments that lead to unjustified gates. For this purpose an *event list* is defined as follows:

Definition 3.8 (event list) *Let $R(S)$ be the set of value assignments $f_i = V_i$ for those variables f_i in a combinational network whose values have been changed by making implications for a given set of value assignments S. Further, let $U(R)$ be the set of variable assignments at the outputs of those unjustified gates which have an input with a variable assignment contained in R. The set $E(S) = R(S) \cup U(R)$ is called the event list E for S.*

In other words, when performing implications for a given set of value assignments S, the event list E contains all variables whose values have been changed. This includes the output signals of new unjustified gates. Furthermore, the output signals of old unjustified gates are included if their status has changed, i.e., if one of their inputs has assumed a different value.

Now we have all concepts necessary to formulate routine *and_or_enumerate()*, shown in Table 3.2. In recursive learning, this algorithm serves as the basic search mechanism. The "learning" steps which extract the necessary assignments or indirect implications are performed at the position indicated by "statements for monitoring the search". If the search is not evaluated, the shown "barebone" procedure produces only a single binary-valued result: it determines whether the given set of value assignments in a combinational network is satisfiable or not.

Theorem 3.6 *Routine and_or_enumerate() is exact.*

Exactness means that the consistency or inconsistency of an initial set of value assignments in a circuit is determined correctly by the algorithm, provided the aborting condition given by a maximum recursion level r_{max} is disabled (e.g., by setting $r_{max} = -1$ in Table 3.2). The proof for this theorem is the same as the proof for the completeness of recursive learning, because learning of implications is not needed for proving that an initial set of value assignments is inconsistent. The proof for the completeness of recursive learning can be found in, e.g., [53] or [54].

While *and_or_enumerate()* provides an interesting algorithmic solution to the satisfiability problem, its main application is to gain important information about the problem by monitoring the search process. For this purpose, r_{max} can be set to a finite positive value, allowing for the adjustment of the invested computation effort and the amount of information gained.

```
/* this procedure operates on a global data structure representing the gate netlist of
the circuit with possibly pre-set value assignments at some of the nodes. S is a new
set of value assignments in the circuit, r is the current level of recursion, initially,
r = 0. r_max is a user-defined aborting criterion */

and_or_enumerate(S, r, r_max)
{
  /* determine OR nodes of AND/OR tree */
  make all direct implications for S in circuit and
  set up a list U^r of unjustified gates in event list E(S);

  if (value assignments are logically inconsistent)
    return INCONSISTENT;

  /* determine AND nodes of AND/OR tree */
  if (r < r_max)
  {
    for (each unjustified gate g in U^r)
    {
      /* try justifications */
      determine set of justifications C_g^r
      for (each justification J_i ∈ C_g^r)
        consistent_i := and_or_enumerate(J_i, r + 1, r_max);

      [ ... statements for monitoring the search ... ]

      /* check logic consistency */
      if (consistent_i = INCONSISTENT for all i)
        return INCONSISTENT;
    }
  }
  return CONSISTENT;
}
```

Table 3.2: Routine *and_or_enumerate()*

3.3.3 AND/OR Reasoning Trees

Just as branch-and-bound enumeration is visualized by a decision tree which is an OR-tree, AND/OR enumeration as given by Table 3.2 can be represented by an AND/OR tree. In this tree, unjustified gates (Def. 3.5) and implied signal values correspond to OR nodes. Justifications (Def. 3.6) represent the AND nodes in the AND/OR tree. In general terms, AND nodes represent some requirements that are either given by the initial set of value assignments or correspond to injected justifications. OR nodes are the logical consequences that result from these requirements.

Let us look at an example to gain some understanding of how algorithm *and_or_enumerate()* works and how it traverses the corresponding AND/OR tree.

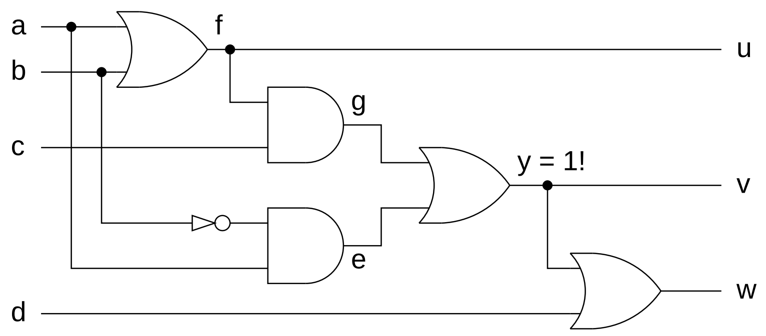


Figure 3.14: Example of a combinational circuit with value assignment ($y = 1$)

Example 3.6 Consider the circuit in Fig. 3.14. We apply *and_or_enumerate()* for an initial situation of value assignments $S = \{y = 1\}$. The initial event list is $E = \{y = 1\}$. Node y in the circuit of Fig. 3.14 becomes an unjustified gate (this is indicated by an exclamation mark) and the complete set of justifications for this gate is $C_y = \{\{g = 1\}, \{e = 1\}\}$. This corresponds to the two AND nodes in level 1 of the AND/OR tree of Figure 3.15. For each justification direct implications imply logic signal values and produce new unjustified gates. Every value assignment forms an OR node in the tree. For $g = 1$ we imply $c = 1$, $f = 1$ and $u = 1$, where node f becomes a new unjustified gate. This requires new justifications and the technique continues to enumerate the AND/OR tree as shown in Fig. 3.15 in a depth-first manner.

More precisely, an AND/OR tree and its construction by the algorithm *and_or_enumerate()* are described by the following definitions:

Definition 3.9 *An* AND/OR tree *is a bipartite rooted directed tree with two disjoint vertex sets,* V_{AND} *and* V_{OR}*. The root node* v_r *is an element of* V_{AND}*. The terminal node (*leaves*) of the tree are elements of* V_{OR}*. Adjacent nodes belong to different vertex sets. Each node* $v_{OR} \in V_{OR}$ *has as attribute a variable assignment* $f = V$*, where* f *is an element of a set of variables*

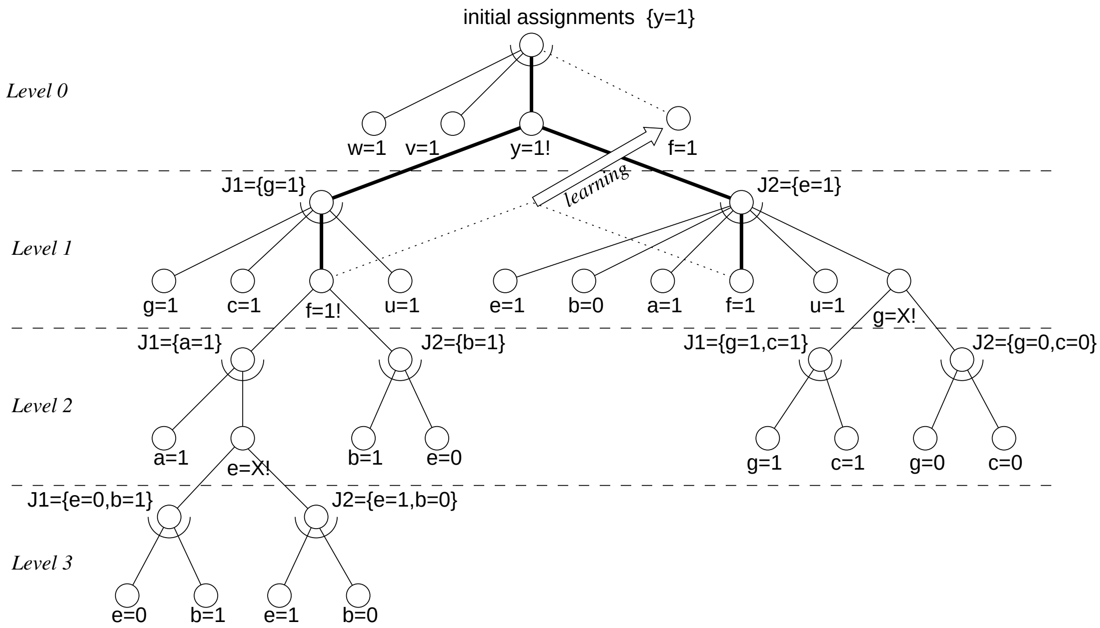


Figure 3.15: AND/OR tree for assignment $y = 1$ in the circuit of Fig. 3.14

$\{f_1, f_2, \dots, f_n\}$ and V is an element of a set of values B. Each node $v_{\text{AND}} \in V_{\text{AND}}$ has as attribute a set of variable assignments $S = \{f_1 = V_1,\ f_2 = V_2, \dots,\ f_k = V_k\}$. Furthermore, each vertex v has as attribute an integer $l(v)$ such that

1. *The root node v_r has*
$$l(v_r) = 0$$

2. *OR nodes v_{OR} have the same l-values as their immediate (AND) predecessors v_{pred}:*
$$l(v_{\text{OR}}) = l(v_{\text{pred}})$$

3. *AND nodes v_{AND} with their immediate (OR) predecessors v_{pred} have*
$$l(v_{\text{AND}}) = l(v_{\text{pred}}) + 1$$

Definition 3.10 *An AND/OR tree with root node v_r can be associated with the AND/OR enumeration* and_or_enumerate() *of Tab. 3.2 as follows:*

1. *each AND node v_{AND} belongs to a set $S = \{f_1 = V_1,\ f_2 = V_2,\ \dots,\ f_k = V_k\}$ of variable assignments at nodes in the combinational network, where this set is given either by the initial set of variable assignments if $v_{\text{AND}} = v_r$ (root node), or by justifications for unjustified gates if $v_{\text{AND}} \neq v_r$ (intermediate nodes). If a set S turns out to be logically inconsistent, the corresponding AND node and all its successors are removed from the tree.*

2. *each OR node v_{OR} belongs to a variable assignment $f = V$ at a node in the combinational network which is required for the logic consistency of the set S associated with the parent AND node of v_{OR}, i.e., an OR node belongs to a variable assignment in the event list $E(S)$. If $f = V$ is at the output of an unjustified gate g then v_{OR} has m AND successors, each belonging to a justification $J \in C_g$, with $m = |C_g|$. If $f = V$ is at the output of a justified gate then v_{OR} is a leaf of the tree.*

Such a tree is called the AND/OR reasoning tree *for the initial set of value assignments S and the given combinational network.*

As an application of AND/OR enumeration, let us take a look at the recursive learning algorithm of [53]. It consists of the algorithm *and_or_enumerate()* plus some statements which evaluate the logic value assignments made for each justification of an unjustified gate. In particular, value assignments which are part of *every* justification of an unjustified gate are "learned" to be necessary for the combination of value assignments to which the unjustified gate belongs. These value assignments are called indirect implications. Viewed in an AND/OR tree, indirect implications can be identified by finding common OR node successors in all AND sucessors of an unjustified gate assignment.

> **Example 3.7** Consider again the circuit of Fig. 3.14 and its corresponding AND/OR tree depicted in Fig. 3.15. The initial assignment $y = 1$ makes gate y unjustified. Two justifications are tried, $\{g = 1\}$ and $\{e = 1\}$, both of which have the assignment $f = 1$ as a logical consequence. In the AND/OR tree, the AND nodes corresponding to the justifications both have an OR node labelled $f = 1$ as a successor. "Learning" corresponds to identifying this assignment as necessary for $y = 1$ and attaching the OR node to the predecessor of $y = 1$.

Learning of indirect implications can occur in any recursion level and the value assignments resulting in the previous level can change the course of subsequent enumeration so that more logical consequences can be examined faster.

The following important observations about AND/OR enumeration and AND/OR reasoning graphs can be made examining the above example. The main purpose of AND/OR enumeration in CAD is not always to prove or disprove satisfiability. Often, like in the application described in Section 3.4, the goal is to extract valuable information from the enumeration process. This may include information about the function as well as the structure of the considered circuitry. A combination of both can lead to effective heuristics for multi-level circuit optimization, as discussed in Section 3.6. It is important to note that in order to obtain this information, the AND/OR tree does not need to be traversed exhaustively. In the above example, one level of recursion is sufficient to identify the indirect implication.

As the example of recursive learning shows, it is possible to evaluate AND/OR graphs with respect to certain information without actually representing them as data structures in computer memory. Recursive learning therefore has memory requirements which grow linearily with the size of the examined circuitry. Of course, it is possible to actually construct the graphs. In doing so, we can find efficient trade-offs between time and memory.

While the goal is to apply AND/OR enumeration to multi-level circuits, it is nevertheless illuminating to examine the behaviour of routine *and_or_enumerate()* in two-level circuits. Consider the two-level circuit in Figure 3.16. Figure 3.17 shows the AND/OR graph if the value 0 is assigned to the output y. AND/OR enumeration for the value 0 at the output of a two-level SOP-type circuit performs a *tautology test*. The SOP is a tautology if and only if a conflict is produced by *and_or_enumerate()*. As can be noted, the AND/OR tree for a *unate* SOP is very simple and has the same structure as the two-level circuit. The root AND node in the AND/OR tree corresponds to the OR gate in the circuit and the succeeding OR nodes correspond to the AND gates in the circuit. Obviously, this is because the AND gates represent implicants of function y and therefore the value assignment $y = 0$ implies OR nodes labelled $h = 0$, $i = 0$ and $j = 0$ which correspond to these implicants. Since the circuit implements a unate function the AND/OR tree is finished in the next level. All AND nodes have only one succeeding OR node, representing a leaf of the tree. This reflects the well-known fact that tautology checking in unate functions is of polynomial complexity. Node that it is only in this special case that the AND nodes and OR nodes of the AND/OR tree have direct relationships with the OR *gates* and AND *gates* of the circuit.

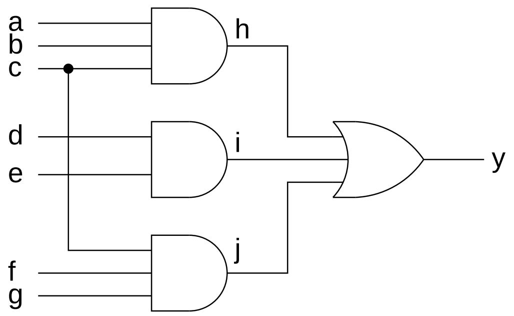


Figure 3.16: A two-level circuit for a unate function

Let the AND/OR tree be levelized according to the recursion depth r in *and_or_enumerate()*, then each level consists of a set of AND nodes with their OR successors. The following theorem holds:

Theorem 3.7 *Let y be the output signal of a two-level combinational circuit in SOP form. The AND/OR tree for the assignment $y = 0$ (tautology test) has only two levels if the SOP expression is unate.*

Proof: *see Appendix A, page 135*

The fact that the AND/OR tree for a unate SOP has only two levels is also related to the well-known result that all prime implicants in a unate SOP are essential, i.e., the unate SOP is a syllogistic formula [12]. If the circuit is not unate, the AND/OR tree has to be continued after level 1 in order to explore the logic consequences which are not covered by the implicants being included in the SOP.

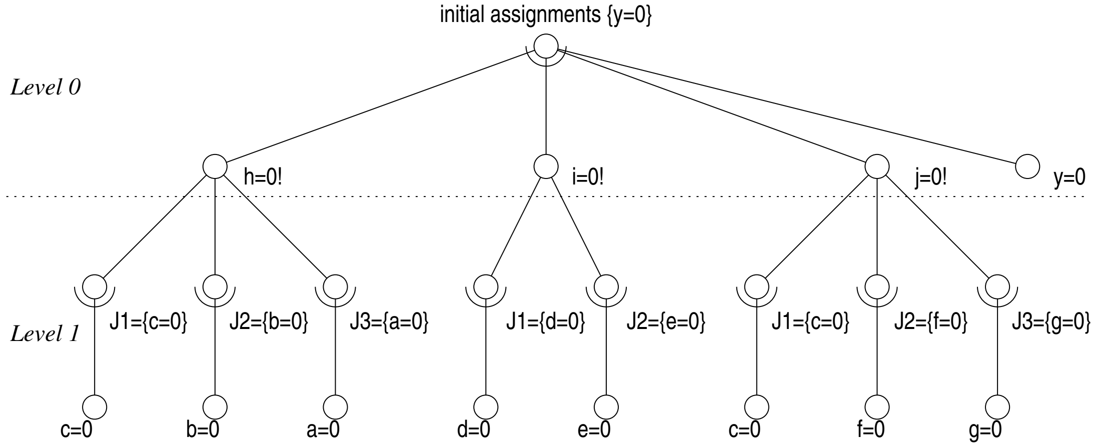


Figure 3.17: AND/OR tree for the unate circuit of Figure 3.16

The situation for the non-unate case is illustrated in Figure 3.18 and Figure 3.19, where the circuit of Figure 3.17 is modified such that it becomes non-unate in variable c.

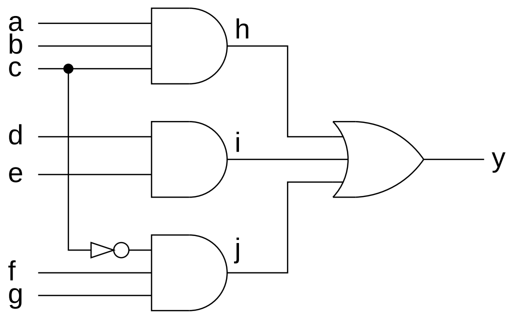


Figure 3.18: Non-unate circuit

Also in the case of a non-unate circuit, level 0 of the AND/OR tree reflects the implicants in the SOP. If the circuit is not unate, however, the AND/OR tree continues after level 1. This is because the justifications at some unjustified line, e.g., $h = 0$ in Figure 3.18, produce events at other unjustified lines without justifying them. The justification $c = 0$ at gate h produces a logic 1 at the output of gate j. This changes the status at gate j and represents an event so that the unjustified line $j = 0$ is added to the list of unjustified gates for the next recursion level. As can be noted, destroying the unateness of variable c by inserting an inverter as shown in Figure 3.18 leads to an AND/OR tree with three levels as shown in Figure 3.19.

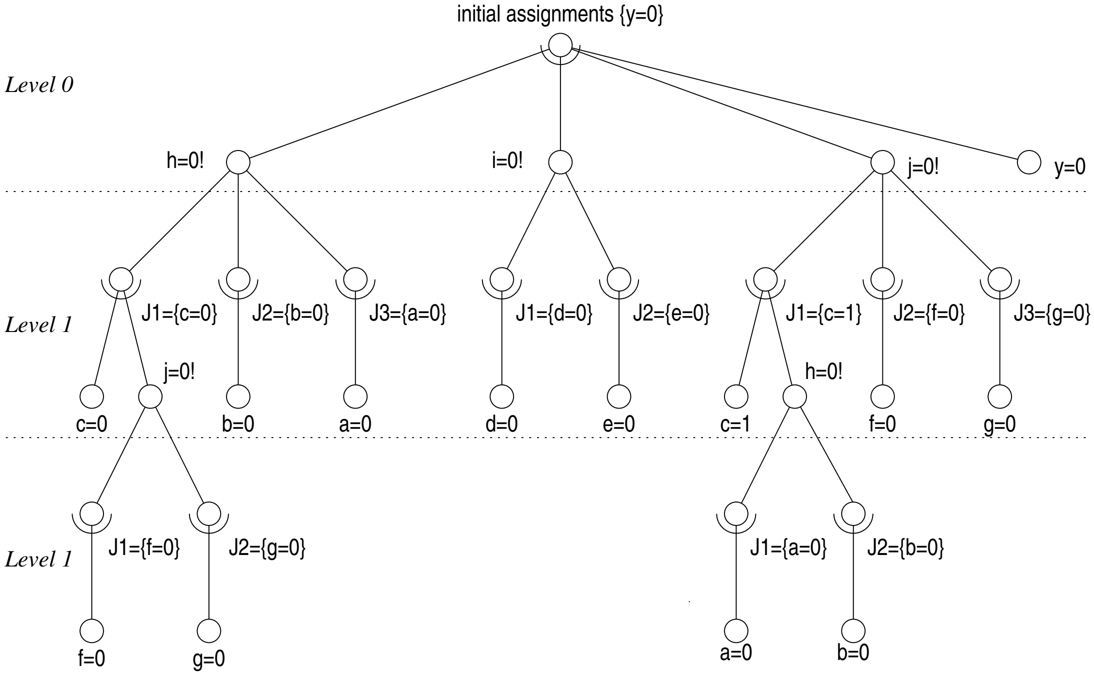


Figure 3.19: AND/OR tree for non-unate circuit of Figure 3.18

3.4 Determining Prime Implicants

In the last section it was shown how AND/OR reasoning graphs for a function in a combinational network are constructed. The main purpose of such a graph, besides solving satisfiability, is to extract important information about the logic function for which it is built. This section will show how we can determine from the graphs multi-level network implicants as introduced in Section 3.2. As explained there, the literals of an implicant of some node y in a multi-level combinational network can belong to arbitrary other nodes in the network, including those which are not in the transitive fanin of y.

We extract the implicants from the AND/OR graph by identifying subtrees with certain properties. The idea we use is the following. Consider a product term $p = l_1 \cdot l_2 \cdot \ldots \cdot l_k$ of k literals which are each associated with the function of a network node or its complement. If p is a 1-implicant of node y then the following implication is valid (by Definition 3.1):

$$(l_1 = 1) \wedge (l_2 = 1) \wedge \ldots \wedge (l_k = 1) \longrightarrow (y = 1)$$

Using contraposition and DeMorgan's law we can restate this as:

$$(y = 0) \longrightarrow (l_1 = 0) \vee (l_2 = 0) \vee \ldots \vee (l_k = 0) \tag{3.1}$$

The AND/OR graph constructed for the initial set of assignments $S = \{y = 0\}$ using routine *and_or_enumerate()* contains subgraphs which can be associated with OR-type expressions like the right-hand side of the Implication 3.1. These expressions are implied by the initial set of value assignments so that the above implication holds. The subtrees are called *implication subtrees* and are formally defined as follows:

Definition 3.11 (Implication Subtree) *An* implication subtree (IST) *is an AND/OR tree with the following properties:*

1. *It is a subtree of an AND/OR reasoning tree.*
2. *The enumeration tree and its subtree have the same root node.*
3. *For each AND node included in the subtree, all its siblings in the AND/OR reasoning tree are also included in the subtree.*

An implication subtree has a set of OR nodes as leaves. These leaves correspond to value assignments which can be interpreted as the right hand side of Implication 3.1. The following theorem allows to relate the leaves of an IST to an implicant in a multi-level network.

Theorem 3.8 *Let y be an arbitrary node in a combinational network and T be the AND/OR enumeration tree for an initial set of value assignments $S = \{y = 0\}$. Consider a product term $p = l_1 \cdot l_2 \cdot \ldots \cdot l_k$ where l_i is a literal corresponding to a variable f_i or its complement in the combinational network. Further, consider an IST of T with a set of leaves, L.*

If there is a one-to-one mapping between the literals l_i of p and the elements $(f_i = V_i)$ of L such that $V_i = 0$ if l_i represents the uncomplemented variable f_i and $V_i = 1$ if l_i represents the complemented variable $\overline{f_i}$, then p is a 1-implicant of y. Analogously, p is a 0-implicant of y if the IST is a subtree of the enumeration tree with the initial assignment $S = y = 1$.

Proof: *see Appendix A, page 135*

Theorem 3.8 states the rule for deriving implicants from an AND/OR tree. An implicant is formed by the conjunction of variables belonging to the leaves of an IST. If a variable at a leaf of the IST is assigned to 0 then we have to take the uncomplemented variable, if it is assigned to 1 we have to take the complemented variable as a literal in the implicant.

Example 3.8 As an example for a multi-literal implicant derived from an AND/OR tree consider the circuit for function $f = a \cdot \overline{b} + b \cdot c$ in Fig. 3.20. The AND/OR tree is derived for the initial assignment $f = 0$. The bold lines indicate an implication subtree according to Definition 3.11: all successors of the OR node corresponding to the unjustified gate d set to 0 are included in the tree. For each AND node included, one OR successor is included. The leaves of the IST are $L = \{(a = 0), (c = 0)\}$. According to Theorem 3.8, this IST represents the implicant $p = a \cdot c$. As can be noted, this implicant is not implemented in the circuit. It is the *consensus* of the implicants realized by gates d and e.

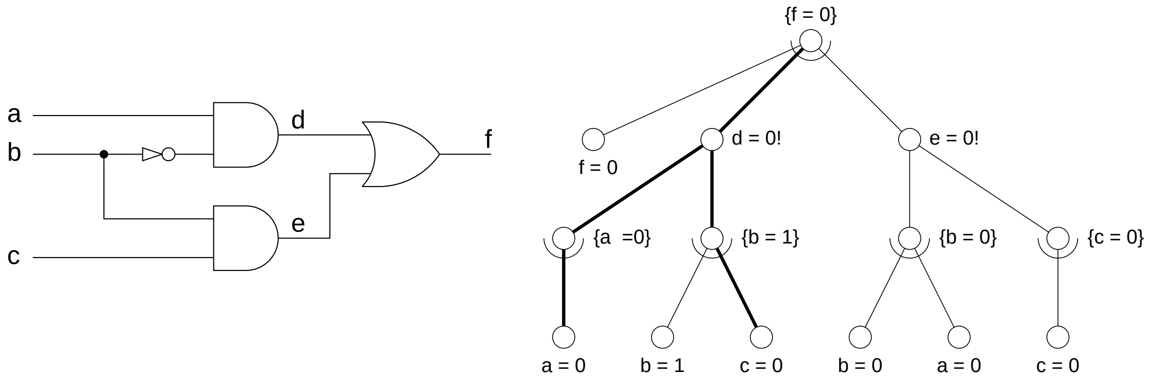


Figure 3.20: Example for IST corresponding to multi-literal network implicant

How are *prime* implicants represented in the AND/OR tree? Note in Definition 3.11 there is no requirement to include in the IST more than *one* child of each AND node of the original tree. In fact, including more than one OR child of any AND node makes the IST non-minimal. Since this non-minimal IST can contain leaves with new variable assignments not needed to make the product term an implicant, the corresponding implicant is non-prime.

Definition 3.12 (Minimal Implication Subtree) *An IST is called* minimal implication subtree (MIST) *if each AND node has exactly one OR child.*

Theorem 3.9 *Let y be an arbitrary node in a combinational network and T be the AND/OR reasoning tree for an initial set of value assignments $S = \{y = V\}$, $V \in \{0, 1\}$. For every* prime *implicant of y there exists a minimal implication subtree (MIST) of T such that the leaves of the MIST correspond to the literals of the prime implicant as given in Theorem 3.8.*

Proof: *see Appendix A, page 137*

Example 3.9 The AND/OR tree for the initial assignment $f = 0$ in the circuit of Fig. 3.20 indeed contains MISTs for every prime 1-implicant of function f. Enumerating all MISTs according to Definitions 3.11 and 3.12, one obtains the following list of network implicants: f, d, e, $a\overline{b}$ (implemented by d), ac (represented twice), bc (implemented by e). There are no further prime implicants of f.

Note that not every MIST corresponds to a prime implicant. For a given MIST with a set of leaves, L, there may be some other MIST with a set of leaves L' such that $L' \subset L$. Obviously, then, the implicant belonging to the first MIST cannot be prime. Fortunately, by tracing from the leaves towards the root of the AND/OR tree, it is very easy to check for a given MIST with a set of leaves, whether a subset of leaves can belong to another MIST.

3.5 D-AND/OR enumeration and Permissible Implicants

In Section 3.2.2 it was pointed out that it is desirable to not only consider equivalence transformations for modifying the structure of a combinational network. We also want to take into account observability don't cares by performing *permissible* transformations which can be identified using *permissible implicants*.

How can we determine permissible implicants? Let $\mathrm{OBS}(f) = \overline{\mathrm{ODC}(f)}$ (see page 24) denote the conditions for which node f is observable at a primary output of the circuit. By Definition 3.3, a permissible 1-implicant p for a node f is given if the following implication holds: $(p = 1) \longrightarrow (f = 1) \vee \overline{\mathrm{OBS}(f)}$. An implicant is a product of literals, $p = l_1 l_2 \ldots l_k$, associated with complemented or uncomplemented functions of network nodes. We can write:

$$(l_1 = 1) \wedge (l_2 = 1) \wedge \ldots \wedge (l_k = 1) \;\longrightarrow\; (f = 1) \vee \overline{\mathrm{OBS}(f)}$$

Using the law of contraposition and DeMorgan's law, we can restate this as

$$(f = 0) \wedge \mathrm{OBS}(f) \;\longrightarrow\; (l_1 = 0) \vee (l_2 = 0) \vee \ldots \vee (l_k = 0) \tag{3.2}$$

In order to identify implications according to Implication 3.2 we need an AND/OR enumeration technique that enumerates the logical consequences of a node f being set to 0 and, at the same time, f being observable at a primary circuit output. The logic conditions, $\mathrm{OBS}(f)$, for f to be observable at a primary output can, in principle, be calculated using the concept of *Boolean difference*, see, e.g., [1].

At this point we must assume that the reader is familiar with the terminology of automatic test generation (ATPG) for single stuck-at faults in combinational circuits as described, e.g., in [1, 54]. It is important to observe that basic ATPG notions can be useful to identify permissible implicants in multi-level circuits. In particular, it turns out that using Roth's D-calculus, we can efficiently incorporate observability constraints into our reasoning techniques. In [53] a technique has been formulated to find all necessary assignments for propagating a fault value to a primary output. It is based on including fault propagation into the AND/OR reasoning process. Table 3.3 shows the algorithm *D_and_or_enumerate()* which can be obtained from routine *complete_unique_sensitization()* of [53] by removing all statements to extract necessary assignments.

The algorithm uses the concept of a *D-frontier* which is a common concept in ATPG tools. The D-frontier consists of all fault signals, i.e., signals being assigned either D or $\overline{D}$, which are input signals of logic gates whose output signal is unspecified. The D-frontier indicates how far the faulty signals have propagated from the fault location towards the primary outputs. Since only one D or $\overline{D}$ is required to be propagated from the fault location to a primary output, the sensitizations at the elements of the D-frontier correspond to successors of an OR node in the AND/OR tree traversed by *D_and_or_enumerate()*. Note that *D_and_or_enumerate()* makes, in turn, use of algorithm *and_or_enumerate()* presented in Section 3.3.2.

```
/* this procedure operates on a global data structure representing the gate netlist of
the circuit with possibly pre-set value assignments at some of the nodes. F is a D-
frontier in the circuit, r is the current level of recursion, initially, r = 0. r_max is a
user-defined aborting criterion */

D_and_or_enumerate(F_r, r, r_max)
{
  /* sensitizations: every element in the D-frontier corresponds to an OR node
  in the D-AND/OR tree */
  for (all signals f_i ∈ F_r)
  {
    successor_signal := f_i;

    /* propagate fault signal along adjacent path towards the primary outputs */
    S := ∅;
    while (successor_signal has exactly one successor)
    {
      fault_value := value of successor_signal;
      successor_signal := successor of successor_signal;
      if (successor_signal is output of inverting gate)
        fault_value := INV(fault_value);
      S := S ∪ {successor_signal = fault_value};
    }
    consistent_i := and_or_enumerate(S, r + 1, r_max);
    set up list of new D-frontier F_{r+1};
    if (X-path check [1] for successor_signal fails)
      /* there is no path left for fault propagation */
      consistent_i := INCONSISTENT;
    if (r < r_max and consistent_i = CONSISTENT)
      consistent_i := D_and_or_enumerate(F_{r+1}, r, r_max);
  }
  /* check logic consistency */
  if (consistent_i = INCONSISTENT for all sensitizations i)
    return INCONSISTENT;
  return CONSISTENT;
}
```

Table 3.3: Routine *D_and_or_enumerate()*

Theorem 3.10 *Let y be an arbitrary node in a combinational network and T be the D-AND/OR enumeration tree for the fault y stuck-at-1. Consider a product term $p = l_1 \cdot l_2 \cdot \ldots \cdot l_k$ where l_i is a literal corresponding to a variable f_i or its complement in the combinational network. Further, consider an IST of T with a set of leaves, L, such that in the combinational network the nodes f_i* cannot be reached by the fault effect.

If there is a one-to-one mapping between the literals l_i of p and the elements $f_i = V_i$ such that $V_i = 0$ if l_i represents the uncomplemented variable f_i, and $V_i = 1$ if l_i represents the complemented variable $\overline{f_i}$, the product term p is a permissible 1-implicant of y. Analogously, p is a permissible 0-implicant of y if the IST is a subtree of the enumeration tree for the fault y stuck-at-0.

Proof: *see Appendix A, page 138*

Injecting a fault y stuck-at-1 means making the value assignment $y = \overline{D}$. This is equivalent to requiring $(y = 0) \wedge \text{OBS}(f)$. Analogously, a fault y stuck-at-0 is injected by setting $y = D$. By identifying MISTs in the D-AND/OR tree, we can determine permissible prime implicants. The following theorem states an important property of AND/OR trees which makes them attractive in logic synthesis:

Theorem 3.11 *Let y be an arbitrary node in a combinational network and T be the D-AND/OR enumeration tree for the fault y stuck-at-V, $V \in 0, 1$. For* every permissible prime implicant *at a node y there exists a minimal implication subtree (MIST) of T such that the leaves of the MIST correspond to the literals of the prime implicant as given in Theorem 3.10.*

Proof: *see Appendix A, page 138*

Example 3.10 Figure 3.21 shows a circuit for which the D-AND/OR reasoning tree is built in Figure 3.22. Consider the fault a stuck-at-1. There are two paths along which this fault can propagate to a primary output. At least one of them has to be sensitized for fault detection. One path traverses gates k and l. Its sensitization yields the value assignments $(d = 1)$ and $(j = 0)$. For the AND/OR tree in Figure 3.22, this produces the left AND node in level 1 with its successors. The second possibility is to sensitize the path through m and q resulting in the right portion of the AND/OR tree. The sensitizations yield value assignments and unjustified lines. These value assignments are enumerated in the usual way as given by Table 3.3, so that the AND/OR tree for the stuck-at-1 fault at signal a is as shown in Figure 3.22. Note that for reasons of simplicity we only consider unjustified gates with specified output signals, i.e., the gates referred to as unjustified lines in test generation literature, for inclusion in the AND/OR tree. Although unjustified gates with unspecified outputs as in the AND/OR tree of Figure 3.15 are necessary for the theoretical completeness of the enumeration, it is possible to neglect them for most practical purposes [53]. The bold lines in Figure 3.22 indicate a MIST that represents a permissible prime implicant $z = b \cdot c$ for node a in the circuit. We will return to this example in the next section.

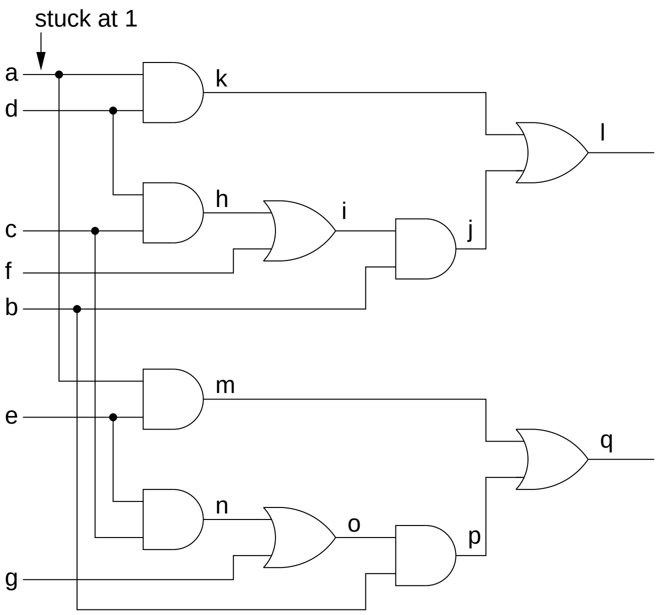


Figure 3.21: Example circuit for permissible implicants

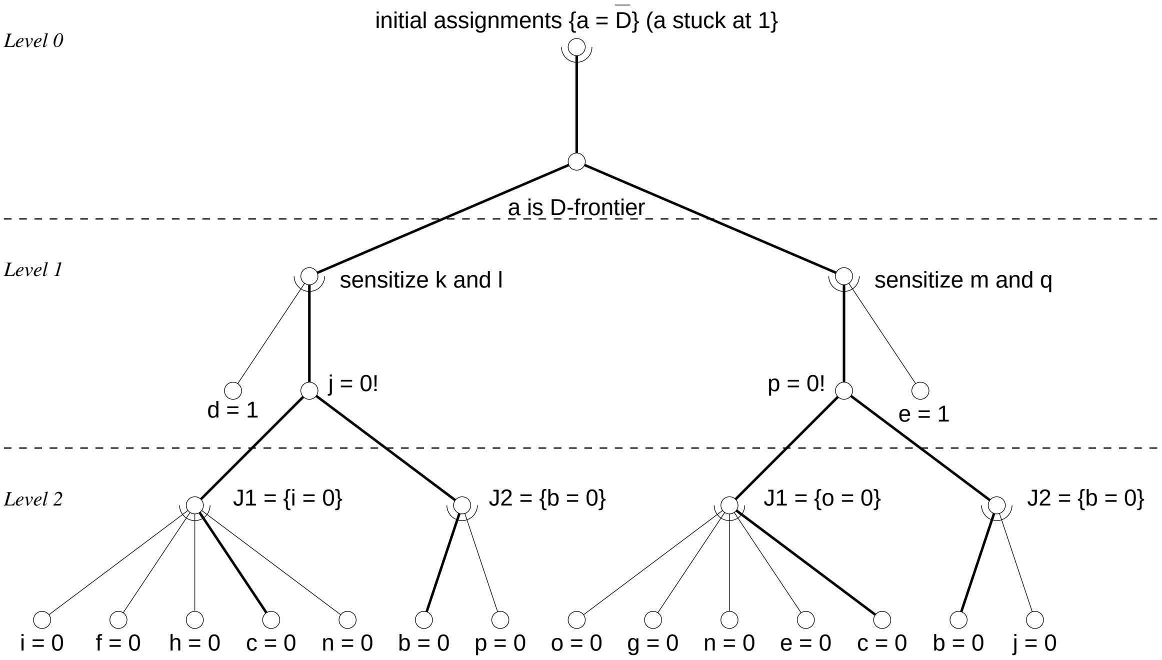


Figure 3.22: AND/OR tree for circuit in Figure 3.21 (bold lines mark MIST)

3.6 Heuristic Multi-Level Optimization

AND/OR reasoning trees, in principle, can generate all permissible prime implicants. Hence, they can be used to obtain any permissible function for some node in the network expressed in terms of arbitrary internal variables of the multi-level network. However, there may be a large number of prime implicants for a given node in the network, especially, if the implicants are expressed in terms of arbitrary (internal) nodes. Therefore, this section is dedicated to demonstrate how the topology of the AND/OR trees can be used to determine those implicants that are particularly promising for optimization.

3.6.1 Selecting Implicants

The motivation for the heuristic given here comes from the observation that there can be subtrees of an AND/OR enumeration tree with several identical leaves. Several such subtrees together can form an IST and represent an implicant. This implicant is then composed of less literals than the IST has leaves, indicating suboptimal circuitry. This suboptimality can informally be explained as follows. If the function for which the AND/OR tree is built contains an implicant which is not implemented directly, the effort necessary to derive the implications corresponding to this implicant is higher than if there existed a direct structural representation of the implicant. In the latter case, the value assignments corresponding to the literals of the implicant would be direct successors of an OR node in the first recursion level.

> **Example 3.11** Consider again the circuit of Figure 3.21 and its AND/OR reasoning tree of Figure 3.22. The MIST indicated by bold lines which corresponds to the implicant $z = bc$ contains two leaves with the value assignment $b = 0$, and two leaves with the value assignment $c = 0$. Therefore, the permissible implicant z is promising for inclusion in a permissible function at node a.

This heuristic for selecting implicants is directly related to the heuristic for selecting promising *divisors* in [52]. There, promising candidates for circuit transformations are indicated by indirect implications which are derived using recursive learning. Such indirect implications correspond to MISTs with the property that all leaves correspond to a single value assignment. For example, in Figure 3.15 (page 41) the bold lines mark such a MIST corresponding to the indirect implication $(y = 1) \longrightarrow (f = 1)$. A MIST that contains only leaves with a single value assignment corresponds to a prime implicant consisting of a single literal. As mentioned earlier, recursive learning can be seen as an AND/OR enumeration technique that determines *all single-literal implicants* of a given logic function.

> **Example 3.11 (continued)** It is now demonstrated how this permissible implicant of node a is constructed from the AND/OR tree in Figure 3.22. To avoid storing the graph we perform repeated AND/OR enumeration. In each pass we extract subtrees from the AND/OR tree that have several leaves corresponding to *one* value assignment. These subtrees correspond to promising literals included in an implicant. Figure 3.23 shows such a subtree for the assignment $c = 0$.

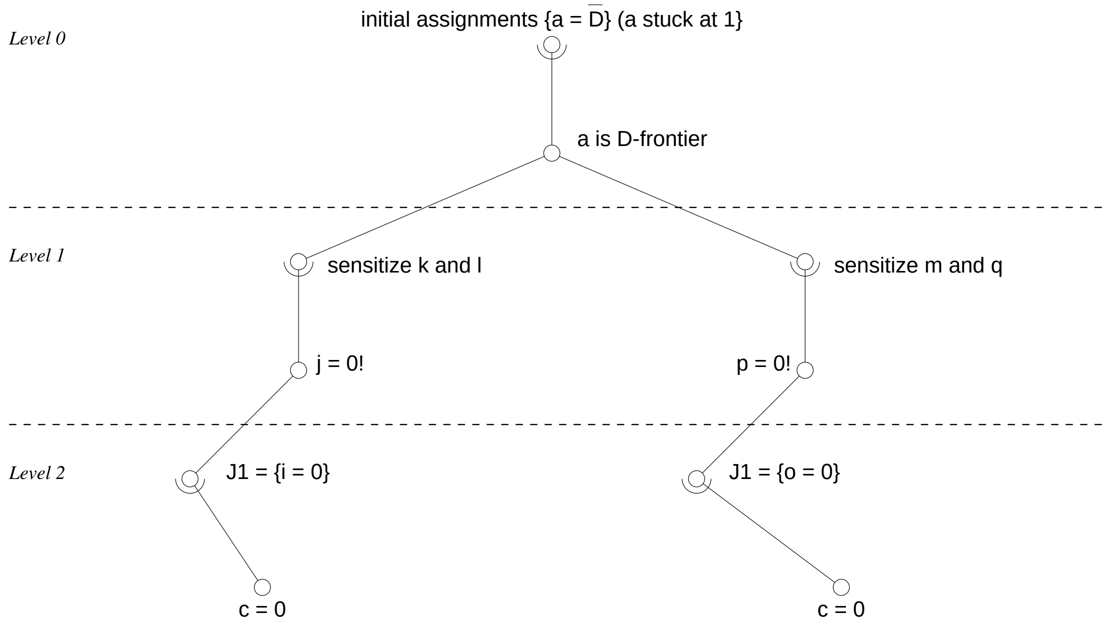


Figure 3.23: Literal subtree of AND/OR tree in Figure 3.22 suggests to include $c = 0$ in implicant

Subtrees with identical leaves belonging to literals of an implicant are called *literal subtrees (LSTs)* and are defined as follows:

Definition 3.13 *A* literal subtree (LST) *for a variable assignment* $f = V$, $V \in \{0, 1\}$, *of a MIST* T *is a subtree of* T *such that*

1. *it has the same root node as* T *and*
2. *it contains all leaves of* T *with* $f = V$ *and contains no other leaves.*

Example 3.11 (continued) The subtree shown in Figure 3.23 is an LST for literal c. The fact that this subtree has more than just one leaf means that c is an "important contribution" to (the observable part of) the function at node a and that it should be included in the implicant. An implicant is assembled step by step by identifying LSTs of large size. In the above example we can proceed as follows. We pick c as the first "seed" literal in the implicant. In order to capture the part of function a not covered by c, we now assign $c = 1$ in the circuit and re-enumerate the AND/OR tree. The resulting AND/OR tree is shown in Figure 3.24.

The resulting AND/OR tree suggests that variable b should be included in the implicant. The subtree for variable b is shown by bold lines in Figure 3.24 and represents an LST with two leaves. Other LSTs of the tree in 3.24 would have only one leaf. Note that this LST is also a MIST, because the subtree contains *all* children of the non-terminal OR nodes (there is only one in this example), so that $b = 0$ is an implication derived from the new set of value assignments. (Note that this MIST could

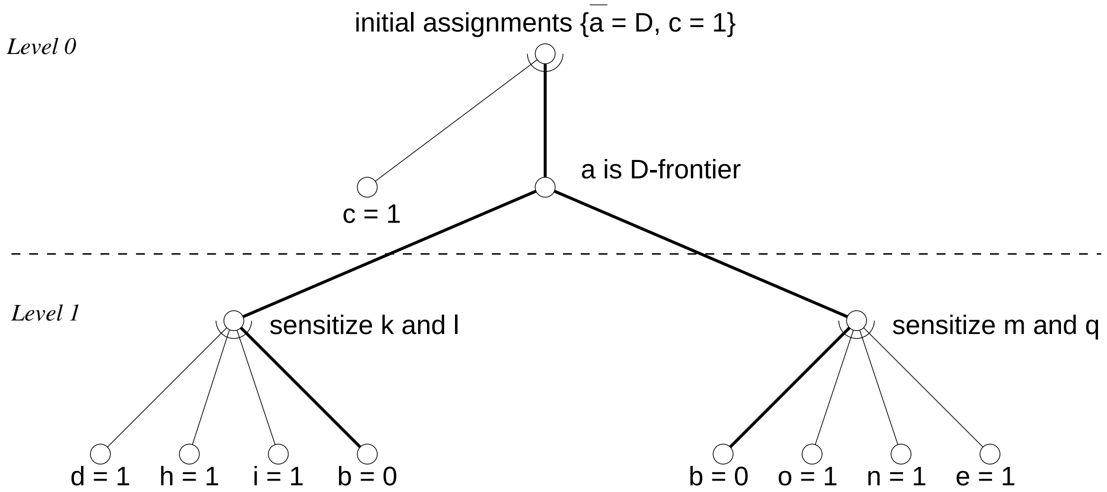


Figure 3.24: Subtree after assigning $c = 1$ indicates that $b = 0$ should be included in implicant

also be derived using recursive learning). Once a subtree represents an implication, the construction is finished and the implicant is complete. We obtain the permissible implicant $z = bc$ for node a.

Consider Table 3.4 and Table 3.5 that show the pseudo-code for calculating implicants as illustrated in the above example.

Table 3.4 describes routine *find_implicant()*. For simplicity, we assume that AND/OR enumeration is performed without any consideration of observability. The procedure starts with the assignment $y = 0$ which yields 1-implicants. (In order to determine 0-implicants, the initial assignment $y = 1$ must be chosen.) Then, the AND/OR tree is traversed as given in Table 3.5 to find large LSTs as illustrated in the example. The signals with their assignments are stored in a list (*LST_candidate_list*) and are ordered according to the size of the subtrees. The variable with the largest LST is chosen as the first variable for the implicant to be created.

If there are several variables with LSTs of the same size we pick the one which belongs to the smallest MIST. The reason for this heuristic is that we intend to generate implicants with as few literals as possible because they will require the least area in the circuit. If the MIST belonging to an LST is very large, we may have to add many more literals to obtain an implicant and therefore we prefer LSTs that are “almost” as large as the MIST they belong to.

Then, the loop in Table 3.4 adds more variables to the product term until the product term is an implicant. Each variable added to the product term is selected by the evaluation of the AND/OR tree in which the previously identified variables are assigned their opposite values. The loop terminates if the current variable and its assignment obtained by *and_or_based_variable_selection()* represent an implication (single-literal implicant) for the previous situation of value assignments. This can either be identified during AND/OR enumeration by the same kind of monitoring that is used to identify necessary assignments in recursive learning or as shown in Table 3.4 where the loop terminates in the next iteration due to a logic inconsistency.

```
/* this procedure operates on a global data structure representing the gate netlist of
the circuit, y is a node in the circuit for which implicants are determined, r_max is the
maximum recursion depth for AND/OR enumeration */

find_implicant(y, r_max)
{
  /* LST_candidate_list, f.LST.leaf_count.V_r, f.V.mark_list are global variables
  and are determined in routine and_or_based_variable_selection() */

  assign in the circuit y := 0;
  mark all gates for all levels;
  consistent := and_or_based_variable_selection(0, r_max);

  if (consistent = INCONSISTENT)
    return FALSE;                              /* y stuck-at-1 is redundant */

  select a (f = V) from the LST_candidate_list with
    maximal f.LST_leaf_count.V_{r = 0};
  if (several candidates have the maximal LST size)
    select from those one
    with minimal f.MIST_leaf_count.V_{r = 0}          /* minimal MIST size */

  if (V = 0) then x_1 := f; else x_1 := f̄;
  implicant := x_1;                /* "seed" literal for forming implicant */

  i := 1;
  loop
  {
    assign in the circuit x_i := V̄_i;
    i := i + 1;
    for (all elements (g, r) in x_i.V_i.mark_list )
      mark gate g for level r;
    consistent := and_or_based_variable_selection(0, r_max);

    if (consistent = INCONSISTENT)
      break;                                   /* implicant is complete */

    select a (f = V) from the LST_candidate_list as above;
    if (V = 0) then x_i := f; else x_i := f̄;
    implicant := implicant · x_i;             /* add literal to implicant */
  }
  return implicant;
}
```

Table 3.4: Routine to calculate implicants in a multi-level circuit

```
and_or_based_variable_selection(S, r, r_max)
{
  make all direct implications for S in circuit
    and set up a list U^r of unjustified gates in eventlist E(S);
  if (value assignments are logically inconsistent)
    return INCONSISTENT;
  else {
    for (every signal f during the implication process)
      if (signal f is assigned value V, V ∈ {0, 1})
        f.LST_leaf_count.V_r := f.LST_leaf_count.V_r + 1;
  }
  if (r < r_max) {
    for (each unjustified gate g in U^r which is marked in level r)
      set up list of justifications C_g^r;
    if (consistent_i = INCONSISTENT for all i)
      return INCONSISTENT;
    else {
      /* determine subtree sizes of LSTs and MISTs by counting leaves */
      n := number of consistent justifications;
      for (every signal f touched during implications in level r+1) {
        /* check heuristic criterion to select LST and MIST */
        if (f.LST_leaf_count.V_{r+1} > f.LST_leaf_count.V_r
            or (f.LST_leaf_count.V_{r+1} = f.LST_leaf_count.V_r
                and n + f.MIST_leaf_count.V_{r+1} −1 < f.MIST_leaf_count.V_r)) {
            /* take this LST and MIST */
            f.LST_leaf_count.V_r := f.LST_leaf_count.V_{r+1};
            f.MIST_leaf_count.V_r := n + f.MIST_leaf_count.V_{r+1} −1;
            /* mark the considered portion of the AND/OR tree */
            f.V.mark_list := {(u, l)| unjustified gate u belongs to an OR node
                of the selected MIST in recursion level l };
        }
        f.LST_leaf_count.V_{r+1} := 0;
        f.MIST_leaf_count.V_{r+1} := 0;
        if (r = 0)
            LST_candidate_list := LST_candidate_list ∪ {f = V};
      }
    }
  }
  return CONSISTENT;
}
```

Table 3.5: Selecting variables for implicants

Table 3.5 shows how variables are selected by AND/OR enumeration. As can be noted, the shown algorithm is based on *and_or_enumerate()* as shown in Table 3.2. Extensions are made to monitor the implications found and to set or unset flags for each node in the circuit and for each recursion level. The idea is to extract implicant variables by "monitoring" the AND/OR enumeration procedure. The additional statements in Table 3.5 show operations that collect information about the sizes of the LSTs and MISTs by counting the value assignments obtained during the consistent justifications. Further, by marking the unjustified gates for each recursion level, it is guaranteed that the next enumeration pass, i.e., the re-enumeration after having made a decision for a certain literal in *find_implicant()*, only considers a MIST that contains the corresponding LST. Also, note that the repeated enumeration is accelerated drastically by excluding all unjustified gates from consideration that have not been marked by the previous AND/OR enumeration pass.

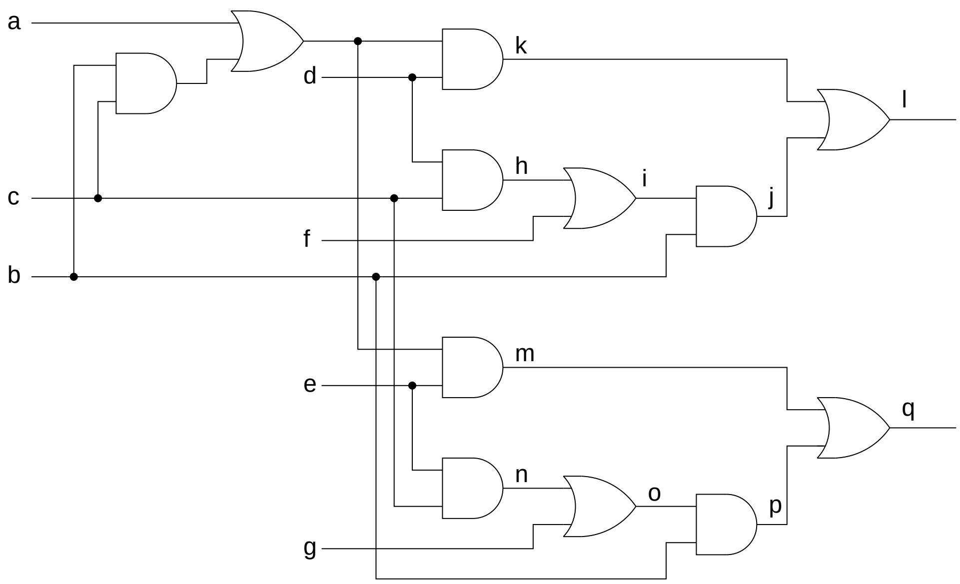


Figure 3.25: Adding permissible implicant bc at signal a

Example 3.11 (continued) Reconsider the circuit of Figure 3.22. The function of the circuit is given by the following Boolean expressions:

$$l = ad + b(cd + f) = (a + bc)d + bf$$

$$q = ae + b(ce + g) = (a + bc)e + bg$$

By manipulating the equations, it can be noted that there exists a common kernel, $a + bc$. Minimization can be achieved by sharing this kernel. With the routines of Table 3.4 and Table 3.5 we obtain the permissible 1-implicant bc as was illustrated above. Note that the suboptimality of the original circuit is reflected by the existence of a MIST with several leaves belonging to the same value assignments, $b = 0$

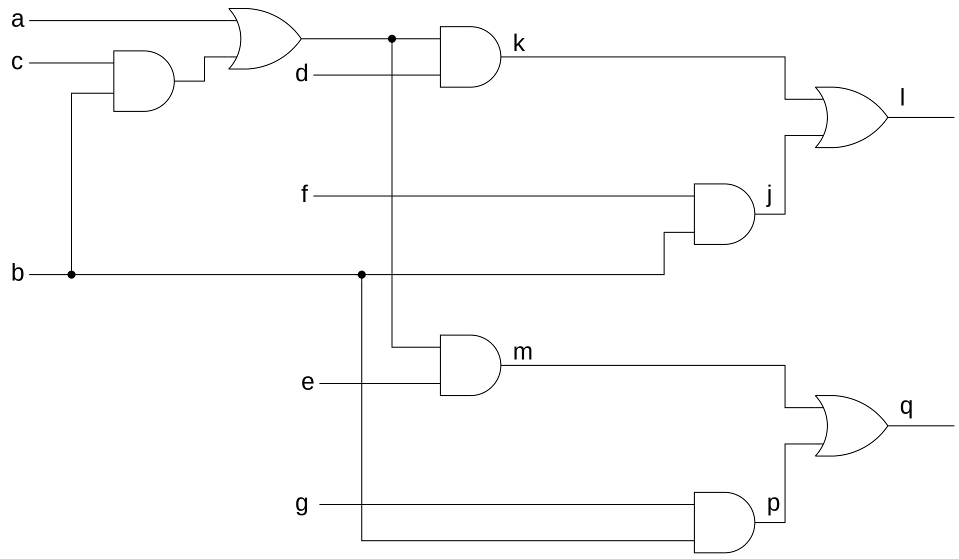


Figure 3.26: Circuit after redundancy removal

and $c = 0$. The fact that bc is a permissible 1-implicant of node a means that according to Lemma 3.4 of Section 3.2.2 node a can be replaced by the permissible function $a + bc$. This leads to an additional OR gate inserted behind the input a with one input connected to an AND gate implementing the product bc. This is shown in Figure 3.25. The circuit optimization procedure follows the two-step methodology described in Section 3.2.2. After having added the implicant to the circuit, redundancy elimination is used to simplify it. This results in the circuit structure shown in Figure 3.26.

As already explained, the optimization in this example can also be obtained by an algebraic kernel extraction technique [10, 74]. Note, however, that the procedures based on AND/OR trees and redundancy elimination are capable of performing *general Boolean manipulations* and are not restricted to "algebraic" transformations using the terminology of [10].

3.6.2 Optimization Procedure

Table 3.6 summarizes a procedure for circuit optimization. It is a refinement of the general two-step methodology described in Section 3.2.2. Logic optimization is performed by applying the described concepts to all nodes in the combinational network. The procedure moves from node to node. Experiments have shown that the optimization results are only moderately sensitive to the order in which the circuit nodes are selected, however, best results are generally obtained by selecting the nodes according to their topological level moving from primary inputs towards primary outputs. For a selected node the concepts of AND/OR enumeration are used to derive promising implicants. The candidates found promising are stored in lists and tried one after the other.

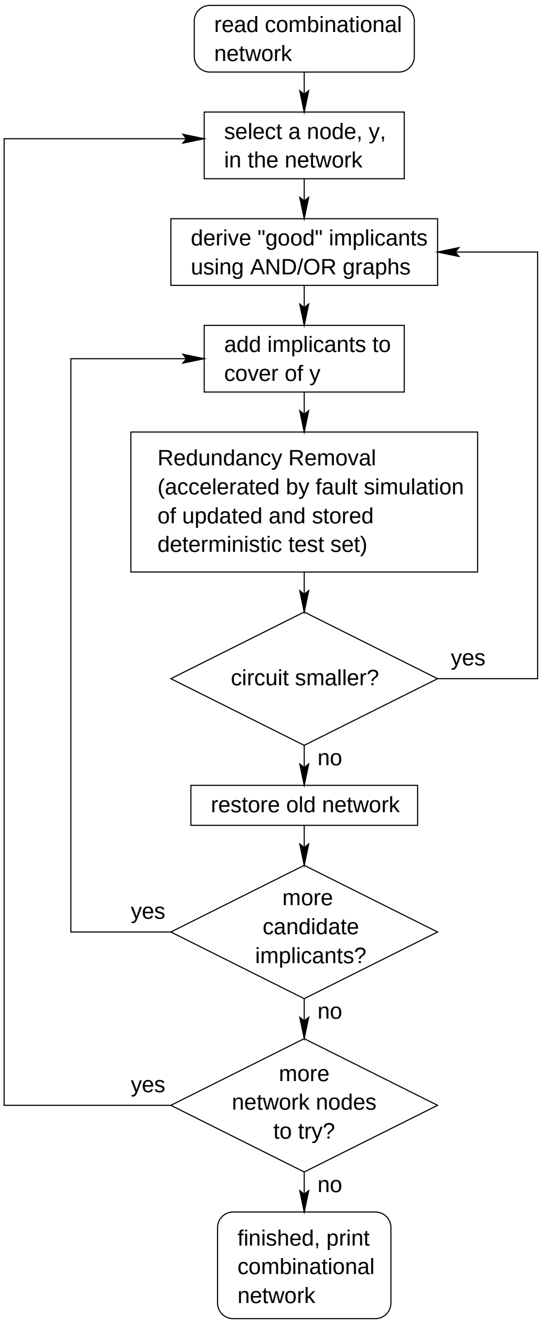


Table 3.6: Procedure for circuit optimization

For each implicant the circuit is transformed according to the lemmas given in Section 3.2.2. After each transformation redundancy elimination is employed. Redundancy identification is quite a time-consuming process. Some speed-up can be gained by not considering all faults in the circuit for redundancy identification but to restrict the search for untestable faults to those areas of the circuit where they are most likely to occur. Experimental results have shown that untestable faults occur almost always at those lines that are involved in the reasoning process of deriving the implicants used for the circuit transformations. Thus, after each transformation, the fault list for redundancy elimination is set up by including both stuck-at faults at only those signals that were "touched" by the AND/OR enumeration when deriving the current implicant.

To further accelerate the process of redundancy elimination, the deterministic test set is always maintained for the most recent version of the circuit. After each circuit transformation, this test set is simulated to quickly discard many faults from further consideration so that only few faults have to be targeted explicitly by deterministic ATPG. After redundancy elimination has been completed it is checked whether or not the circuit has become smaller. If it is smaller the new current circuit is maintained, otherwise the previous version is recovered. This is continued for all nodes in the network until no more improvements can be found. Several runs can be made through the circuit varying the recursion depth and the number of candidate implicants tried at each node in different runs.

3.6.3 Experimental Results

The described methods have been implemented by integrating them into the HANNIBAL tool system. For efficient fault simulation, HANNIBAL contains the fault simulator FSIM [59]. Logic transformations are derived by AND/OR reasoning techniques. Our experiments have shown that circuit transformations using only single-literal implicants have already a high optimization potential. In many cases, adding a single-literal implicant to the cover of a logic function only means introducing a single connection from the implicant's signal source to the gate producing the function. A multi-literal implicant adds higher costs because an additional gate is needed to represent the product of the literals. Nevertheless, there are applications where single-literal implicants alone do not produce satisfactory results. We will show experiments for heuristic area optimization using single-literal implicants as well as optimization using multi-literal implicants.

Single-Literal Implicants

We compare HANNIBAL with other state-of-the-art optimization tools. For a fair comparison it is important to remember that several different ways of measuring the area costs are used in practice. HANNIBAL and RAMBO [32] operate on a gate netlist description and measure the area in terms of the number of *connections*. Technology-independent optimization tools like SIS measure the area in terms of numbers of literals. The difference between these two reflects the different models of a multi-level circuit. A literal count is used when the circuit is represented by a general Boolean network (Def. 2.4, page 14). The functions at the individual internal nodes of the network contribute to the area measure by the number of literals needed to describe the function. Obviously, the literal count depends on how the individual node functions are represented. A SOP expression usually uses more literals than a factored form. If the circuit is given as a gate netlist description, it is common to count connections. A connection is defined to be a distinct input of a gate having at least two inputs. Inputs to inverters are not counted, because most optimization techniques disregard the cost of inverters during the main optimization procedure and minimize the number of inverters in a post-processing phase called *phase assignment*. Furthermore, neglecting inverters when counting connections is analogous to treating a variable and its complement equally as one literal when determining the literal count.

For a fair evaluation of our tool, we present the results in terms of both, number of connections and number of literals. For RAMBO and HANNIBAL the number of literals (factored form) has been obtained by reading the optimized circuits into SIS and post-processing them such that a technology-independent factored form is obtained. For this purpose we used a SIS script which performs some standard network manipulations. To count connections for SIS we map the optimized circuit to a generic library which contains the basic gates that are allowed in our netlist description. Note that comparing connections or literals may slightly bias the results. Since RAMBO and HANNIBAL optimize in terms of connections whereas SIS uses literals, comparing connections can bias the results in favor of HANNIBAL and RAMBO. Comparing literals gives a certain advantage to SIS. Therefore, for all circuits we always present both area measures.

In all experiments HANNIBAL passes through the circuit four times performing modifications of the cover at every internal network node where single-literal implicants can be identified. The recursion depth is 1 for the first two passes and 2 for the final two passes. We also experimented with greater depths of recursion. It turned out that recursion depth higher than 2 did not lead to improved optimization results because the same transformation which can be derived by high recursion depth can usually also be obtained by a sequence of local transformations derived by small recursion depth. Also, for larger designs a recursion depth of 4 and higher is usually not affordable in terms of CPU time.

Table 3.7 shows results for SIS-1.2, RAMBO_C and HANNIBAL. SIS-1.2 is run using *script.rugged* which includes the powerful techniques of [74] and [82]. No pre-optimization is used to process the circuits in RAMBO and HANNIBAL. As can be noted, for most benchmark circuits HANNIBAL produces the smallest circuits. This is quite remarkable because it shows that most circuit manipulations performed by conventional technology-independent multi-level minimization techniques are covered by the implicant-based transformations described in this chapter using single-literal implicants. In particular, heuristic guidance by indirect implications corresponding to single-literal MISTs proved remarkably powerful.

In the next experiment it is examined how much optimization is possible by HANNIBAL if the circuits are pre-processed by SIS. As shown in Table 3.8 substantial area gains are possible in many cases. For 7 out of 25 circuits, the gain is more than 20%. Also note that the CPU times for HANNIBAL are significantly shorter in many cases if the circuits are first run through a technology-independent minimization.

Name	Original	SIS-1.2	RAMBO	HANNIBAL	
	#conn. (#lit.)	#conn. (#lit.)	#conn. (#lit.)	#conn. (#lit.)	CPU h:min:s
c1355	992 (562)	778 (554)	837 (546)	746 (540)	00:01:31
c1908	1059 (769)	708 (535)	784 (551)	696 (511)	00:04:38
c2670	1559 (1023)	1082 (752)	1520 (816)	1064 (701)	00:03:37
c3540	2226 (1658)	1649 (1288)	1810 (1331)	1628 (1221)	01:13:57
c432	296 (270)	247 (205)	271 (207)	207 (181)	00:00:55
c499	368 (562)	776 (554)	837 (546)	348 (540)	00:00:16
c5315	3492 (2425)	2548 (1731)	3201 (1851)	2661 (1779)	00:15:39
c6288	4768 (3315)	4695 (3337)	3834 (3294)	3723 (3252)	00:29:18
c7552	4734 (3087)	3457 (2312)	3385 (2188)	2542 (1826)	00:36:00
c880	640 (433)	594 (413)	643 (410)	578 (400)	00:00:44
9sym	387 (237)	384 (186)	324 (217)	286 (217)	00:03:19
alu2	669 (453)	507 (361)	509 (359)	347 (274)	00:05:29
alu4	1299 (855)	975 (694)	1006 (722)	826 (646)	00:36:40
apex6	1214 (835)	1074 (743)	1327 (759)	1000 (697)	00:03:54
apex7	410 (289)	331 (245)	412 (251)	309 (229)	00:00:24
dalu	3533 (2610)	1364 (979)	2007 (1344)	1710 (1102)	02:10:18
frg2	2244 (2005)	1182 (887)	1734 (1157)	1315 (982)	00:27:35
pair	2795 (1803)	2356 (1602)	2594 (1636)	2155 (1636)	00:15:29
rot	1085 (764)	928 (672)	1093 (662)	834 (633)	00:02:20
term1	773 (456)	235 (170)	363 (248)	208 (149)	00:00:44
ttt2	434 (324)	303 (239)	300 (191)	204 (148)	00:00:26
x1	627 (445)	409 (298)	503 (333)	392 (298)	00:00:58
x3	1589 (1133)	1101 (787)	1547 (985)	1110 (1035)	00:03:59
x4	843 (1607)	512 (380)	777 (449)	583 (400)	00:03:34

Table 3.7: Logic minimization results for HANNIBAL (Sun Sparc 5)

Name	SIS-1.2 (script.rugged)	SIS-1.2 (script.rugged) + HANNIBAL	
	#conn.(#lit.)	#conn.(#lit.)	CPU time h:min:s
c1355	778(554)	759(543)	00:01:20
c1908	708(535)	690(516)	00:02:38
c2670	1082(752)	1021(773)	00:02:37
c3540	1649(1288)	1571(1144)	00:34:18
c432	247(205)	212(165)	00:00:28
c499	776(554)	763(540)	00:01:26
c5315	2548(1731)	2425(1679)	00:07:16
c6288	4695(3337)	3720(3210)	00:31:28
c7552	3457(2312)	2516(1778)	00:22:24
c880	594(413)	589(416)	00:00:48
9sym	384(186)	227(178)	00:01:49
alu2	507(361)	355(279)	00:04:44
alu4	975(694)	776(596)	00:26:38
apex6	1074(743)	999(687)	00:02:36
apex7	331(245)	294(224)	00:00:14
dalu	1364(979)	1171(735)	00:44:02
frg2	1182(887)	1052(834)	00:05:46
pair	2356(1602)	2149(1509)	00:15:35
rot	928(672)	822(641)	00:01:45
term1	235(170)	183(131)	00:00:10
ttt2	303(219)	225(165)	00:00:24
vda	688(615)	630(566)	00:09:36
x1	409(298)	377(287)	00:00:31
x3	1101(787)	995(758)	00:03:48
x4	512(380)	482(357)	00:00:47

Table 3.8: Results for HANNIBAL after pre-processing with SIS (Sun Sparc 5)

Multi-Literal Implicants

The AND/OR graph based techniques to identify multi-literal implicants have been applied to the problem of PLA factorization. This application is interesting because this task cannot be accomplished in a satisfactory way by only using single-literal implicants.

PLA factorization results		SIS-1.2 *(script.rugged)*		HANNIBAL			
		single run	multiple runs	fixed settings		best	RL
name	# conn.	# conn.	# conn.	# conn.	CPU	# conn.	# conn.
5xp1	369	164	159	79	0:01:58	78	237
9sym	609	320	206	152	0:17:00	83	609
clip	1055	195	187	110	0:10:24	90	520
con1	32	30	30	27	0:00:01	27	30
duke2	995	540	510	416	1:12:33	355	612
e64	2144	253	253	253	0:15:19	253	253
misex1	154	77	77	59	0:00:51	55	81
misex2	206	121	121	121	0:02:47	111	134
o64	195	-	-	195	0:00:08	195	195
rd53	176	52	52	36	0:00:42	34	99
sao2	501	192	190	116	0:05:38	108	195
vg2	914	124	124	115	0:03:22	112	141

Table 3.9: Results for multi-literal implicants (Sun SPARC 5)

Table 3.9 shows results for two-level circuits from the MCNC benchmark set which are factorized into a multi-level description. Since our implementation does not accept external don't-cares, we only selected examples that are completely specified. The results of HANNIBAL are compared with SIS-1.2 (using *resub -a*, *simplify -d* followed by *script.rugged*). The area is measured in terms of connections based on a generic library of the basic gate types. For both tools we show results for the same fixed settings (single run of *script.rugged* for SIS) and interactive use (column "multi-run" for SIS and column "best" for HANNIBAL). Column "RL" shows the results if only single-literal implicants (recursive learning) are used.

Implicants are determined by repeated enumeration as described in Section 3.6.1 without actually building the graphs in memory. If the graphs are actually constructed, implicants can be determined much faster by simple operations on the graph. Future implementations may investigate appropriate trade-offs between memory and time (see also Chapter 6). An advantage of the presented AND/OR trees is that they need not be constructed to their full size in order to be useful. In these experiments, AND/OR trees have been examined only up to a recursion depth of 3. This, however, proved sufficient to obtain the shown optimization results.

The experimental results confirm our conjecture that topological properties of AND/OR reasoning graphs can be used to guide an optimization process. In the examined cases HANNIBAL obtained (sometimes significantly) better optimization results than SIS-1.2. This also remained true when we ran SIS interactively, repeating *script.rugged* multiple times.

Our experiments also demonstrate the practical relevance of the theoretical result in Theorem 3.7 (page 43). This is illustrated by the MCNC benchmark circuit *o64*. This circuit is small, nevertheless it is impossible to build an OBDD for this circuit. No literal in the circuit description appears in more than one product term (hence the SOP is unate) and all prime implicants are essential. Therefore, optimization is impossible, either with two-level or with multi-level optimization techniques. SIS-1.2 runs out of memory in both *script.rugged* and *script.algebraic* after wasting about ten minutes of CPU time in each script. However, since the circuit is unate, Theorem 3.7 applies. This explains why HANNIBAL has no problem with this example. The AND/OR reasoning trees for this circuit are of linear size and of trivial structure. No MISTs exist where several leaves belong to the same value assignment. Therefore, no promising implicants can be generated. The fact that this circuit cannot be further optimized is determined very quickly by the tool and only little CPU time is spent.

3.7 Determining a Cover

The techniques introduced so far allow us to calculate implicants in multi-level combinational networks. We have proposed a general network transformation scheme to perform arbitrary network manipulations and we have studied heuristics that help us to steer the implicant-based transformations so that they optimize the network.

We do not, however, have a method that performs exact minimization of a multi-level combinational circuit. The absence of a structural constraint on the circuit representation as opposed to the two-level case provides additional degrees of freedom that can be exploited in heuristic procedures. On the other hand, due to the much greater search space, algorithms for exact multi-level circuit optimization are of such high complexity that they are not of practical interest. Only a few exact methods have been reported in the literature. For example, the method proposed by Lawler [58] extends the Quine-McCluskey [73, 63] method to the multi-level case. The method minimizes the number of literals in single-output multiple-level networks. It is based on a definition of a multi-level prime implicant, i.e., a multi-level function p expressed in terms of the primary circuit inputs which implies another function f to be 1. Note that this definition differs from our definition of a network implicant because literals cannot be internal network nodes, and implicants are multi-level factored forms. Given a fixed number of gate levels, Lawler's method finds implicants and a minimal cover of the circuit function. An "absolutely minimal" form is found by trying increasingly large number of levels until an aborting criterion is met and the best solution found so far is taken. As in the Quine-McCluskey procedure, Lawler's technique computes prime implicants (based on his definition) and solves a covering problem to find a minimal n-level representation of a Boolean function.

Although we are not interested in solving the minimization problem for multi-level circuits exactly, the subproblem of finding a set of prime implicants repesenting a cover of a Boolean function is of both, theoretical and practical interest. In this section we show how we can solve a covering problem using AND/OR graphs. This algorithm fits nicely into the set of algorithms developed so far and it is useful in the synthesis of structural set representations which will be introduced in Section 3.8. Interestingly, using AND/OR reasoning techniques, the algorithmic steps for finding a minimal cover are the same as the steps for finding prime implicants of a Boolean function. Also, the heuristics used to find "good" implicants for heuristic multi-level optimization, as discussed in Section 3.6.1, can be reused for a heuristic covering algorithm. The reason for this analogy results from the fact that sums and products are dual concepts in

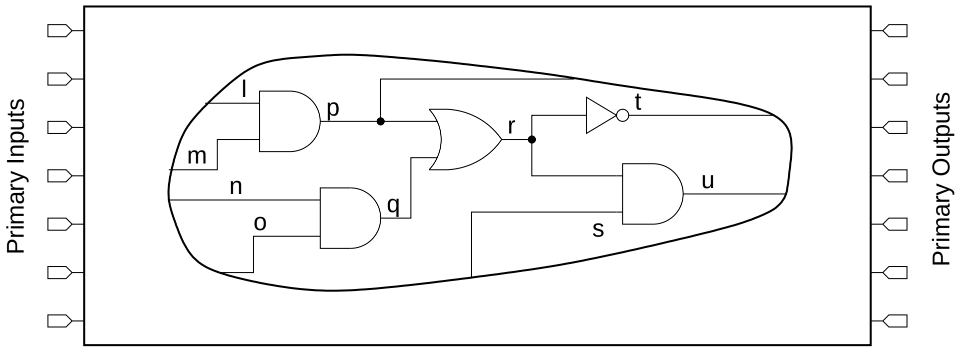


Figure 3.27: Fragment of multi-level circuit

Boolean algebra. Consider the fragment of a multi-level combinational circuit in Figure 3.27. The function at node r can be seen as being implemented by a (local) sum of products. OR gate r is fed by two AND gates p and q representing implicants in a SOP. The function at r is represented by a cover consisting of these two implicants p and q. On the other hand, the OR gate r can be seen as the implementation of an *implicate* contributing to the cover of a POS-type subcircuit at the AND node u. The cover of the function at u is composed of the implicates r and s. So, the OR node r serves as a sum term of both, a SOP and a POS.

Using our definition of multi-level network implicants, we say that p and q are 1-implicants of r. Also, using DeMorgan's law, the OR node r can be seen as an implementation of the complement of the product $\overline{p}\,\overline{q} = \overline{r}$ which is a 0-implicant of function u.

In addition, $\overline{p}\,\overline{q}$ is a 0-implicant of function r itself. This implicant has an important property: its complement consists of 1-implicants of r that represent a cover of r. This relationship may be trivial in this case because it is obvious from the circuit structure, but it can be generalized to arbitrary 1-implicants.

Theorem 3.12 *Let f be a function of a node in a combinational network and $p_1 \dots p_n$ a set of product terms with literals belonging to complemented or uncomplemented variables of the network. Let d be the disjunction of the product terms: $d = p_1 + p_2 + \dots + p_n$. Then, f is functionally equivalent to d if, and only if, the p_i are 1-implicants of f and $\overline{d} = \overline{p_1}\,\overline{p_2} \dots \overline{p_n}$ is a 0-implicant of f. In this case, d is called a* 1-cover *of f.*

Proof: *see Appendix A, page 139*

The literals of the 0-implicant $\overline{d}$ in Theorem 3.12 are all complemented variables belonging to the prime 1-implicants of the function. The expression $\overline{d} = \overline{p_1}\,\overline{p_2}\ldots\overline{p_n}$ is *unate*. This is related to the well-known fact that the problem of finding a minimal set of implicants covering a function is a *unate covering problem*.

The dual form of the above theorem relates a function to its 0-implicants (corresponding to implicates). It can be formulated in the following way:

Theorem 3.13 *Let f be a function of a node in a combinational network and $p_1 \ldots p_n$ a set of product terms with literals belonging to complemented or uncomplemented variables of the network. Let c be the conjunction of the* complements *of the product terms: $c = \overline{p_1} \cdot \overline{p_2} \cdot \ldots \cdot \overline{p_n}$. Then, f is equivalent to c if, and only if, the p_i are 0-implicants of f and c is a 1-implicant of f. In this case, d is called a* 0-cover *of f.*

The occurrence of complements in the conjunction of Theorem 3.13 comes from the fact that our notations are biased towards the SOP form of functional expressions. Instead of defining implicants (product terms) and implicates (sum terms) we have defined 0- and 1-implicants using only products. Sum terms have to be converted to products using DeMorgan's law. Nevertheless, both types of expressions, conjunctive and disjunctive, can be described in these terms. And the AND/OR graph-based methods operating on them are the same for both types.

Theorem 3.12 and Theorem 3.13 suggest the following technique for calculating a cover of the function. Remember that a 1-cover corresponds to a 0-implicant whose literals belong to uncomplemented prime 1-implicants of the function. An AND/OR graph for the initial assignment $y = 1$ contains ISTs corresponding to 0-implicants of function y. By selecting appropriate ISTs we can find the 0-implicants which represent a cover. Since we are interested in irredundant covers, we identify *minimal* ISTs (MISTs) which correspond to prime 0-implicants.

The following steps determine a 1-cover of a function f at a node in a multi-level combinational network.

1. Calculate a set of prime 1-implicants $S = \{p_1, p_2, \ldots, p_n\}$ for function f (using the techniques in Sections 3.4 and 3.5).
2. Construct AND gates representing these 1-implicants in the circuit.
3. Calculate a set of 0-implicants of f such that the literals belong to the *complemented* variables $p_1, p_2, \ldots, p_n$ of step 1. Each calculated prime 0-implicant represents an irredundant 1-cover for f.

If in step 3 we calculate *all* prime 0-implicants of function f, we obtain *all irredundant 1-covers* of f which are composed of implicants in the set S. A prime 0-implicant with the least number of literals corresponds to a *minimum* cover. The technique to determine a 0-cover of the function works analogously.

Example 3.12 Figure 3.28 shows a circuit implementing the function $y = (a + b)(\overline{a} + c)$ and the AND/OR graph for an initial assignment $y = 0$. From the graph, all

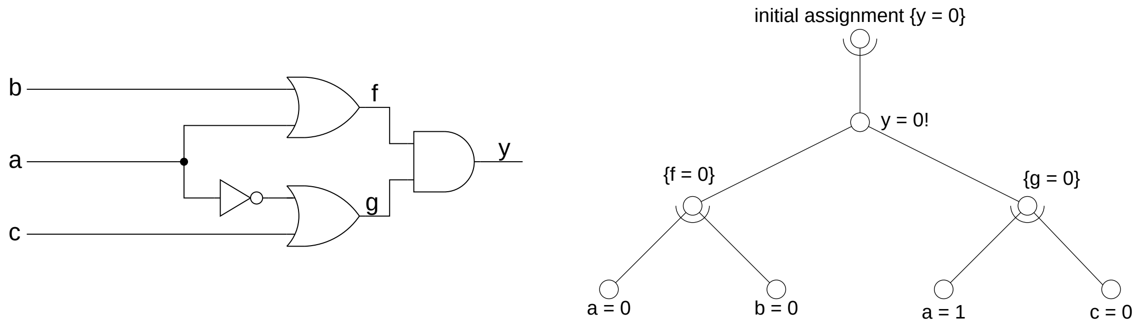


Figure 3.28: Example circuit and AND/OR graph for initial assignment $\{y = 0\}$

prime 1-implicants of the function in terms of the primary inputs can be determined using the methods described in Section 3.4. The MISTs in the graph correspond to the implicants $h = \overline{a}b$, $i = ac$, and $j = bc$.

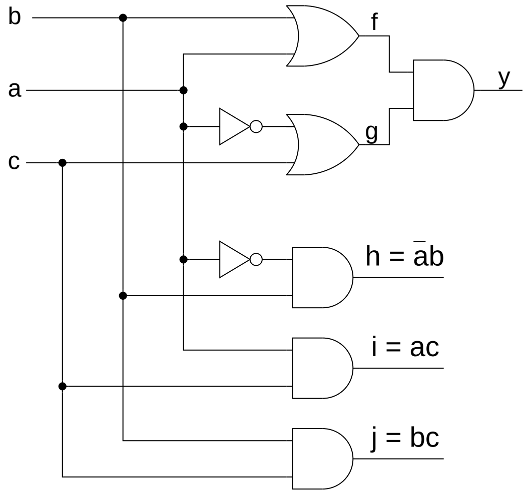


Figure 3.29: Circuit with representation of 1-implicants

Representing these implicants as AND gates yields the circuit structure of Figure 3.29. Now we construct an AND/OR graph for the initial assignment $y = 1$ in the new circuit containing the implicant representations. These AND gates can also become unjustified gates according to Definition 3.5. Therefore, the AND/OR graph for the new circuit contains additional levels. The AND/OR reasoning explores the different possibilities for the function to become 1, thereby enumerating all configurations of implicants that cover the function. The AND/OR graph is shown in Figure 3.30. (For reasons of space, the enumeration of the unjustified line g in recursion level 0 is not depicted.)

The bold lines in the graph mark a MIST corresponding to prime 0-implicant $\overline{h}\,\overline{i}$.

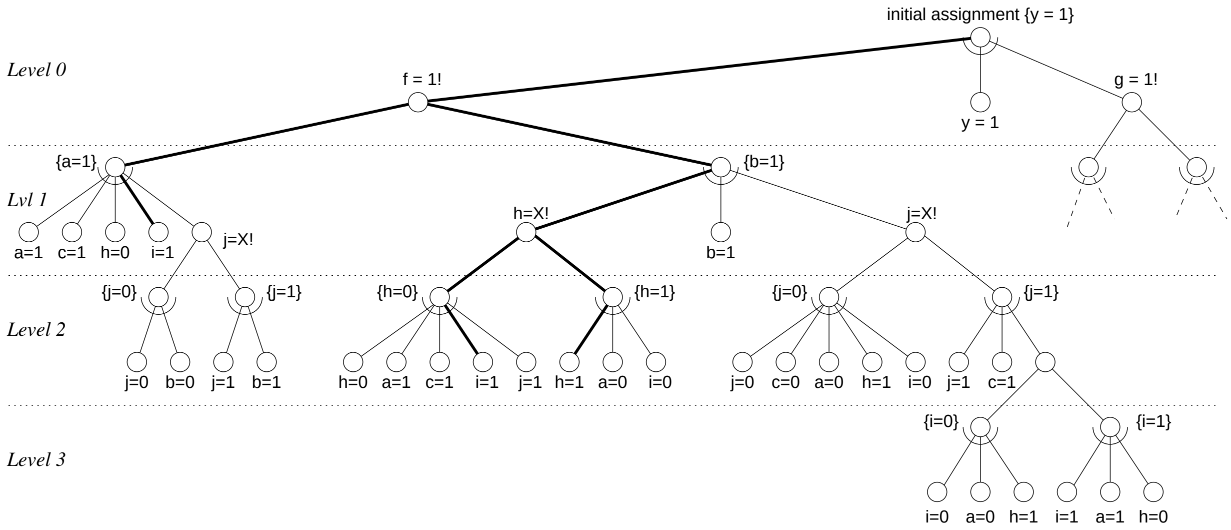


Figure 3.30: AND/OR graph for initial assignment $\{y = 1\}$ in circuit of Figure 3.29

This 0-implicant corresponds to a minimum cover: $y = \overline{a}b + ac$. The following observations can be made.

- The cover can be identified without visiting the complete graph. A maximum recursion level of 2 and a restriction to one of the two unjustified lines f and g would be sufficient in this example.
- The 1-cover ISTs in the subgraph rooted at $f = 1!$ can only be chosen such that $h = 1$ and $i = 1$ are contained. These correspond to *essential prime implicants*. They are part of any cover. (They are part of any cover that can be determined from the shown subgraph. The subgraph rooted at $g = 1!$ which is not shown in the figure has a symmetric structure. No other MISTs exist so that h and i are indeed essential primes.)

As seen in the example, AND/OR graphs provide an interesting technique to determine a *minimum cover* of a function f consisting of prime implicants in multi-level networks as defined in Section 3.4. The topological analysis of the AND/OR graphs identifying implicants is the same as the analysis determining the cover. The resulting solution is exact, if the AND/OR graphs are traversed exhaustively.

Also, AND/OR graphs are very well suited for a heuristic approach to find a minimal cover. It is not necessary to visit the complete graph in order to find a solution. In the first step of the procedure outlined above, we do not need to determine *all* prime 1-implicants (which can be a number growing exponentially with the number of input variables). We only need a set of implicants such that a disjunction of all implicants found represents a cover. This is possible without visiting the complete AND/OR graph for the initial assignment $y = 0$. In the third step of the above procedure, we do not need to determine *all* prime 0-implicants to find a minimal cover. Again we can restrict AND/OR reasoning for the initial assignment $y = 1$ to a limited number of recursion levels.

3.8 Implicant-Based Set Representations

The previous sections have introduced AND/OR reasoning graphs for determining prime implicants in multi-level combinational networks. This concept has allowed us to formulate an approach to multi-level optimization as a natural generalization of well-known algorithms for two-level circuits. This is theoretically very appealing, because it permits for the first time the unified view on two-level and multi-level optimization algorithms. Furthermore, our experimental results clearly show the practical usefulness of the proposed approach.

The main focus of this thesis, however, is a different application: the verification of sequential circuits. Remember that the output of sequential systems does not only depend on the current input but also on the internal state of the system. This internal state is a result of previous inputs to the system. A large sequential circuit can assume a tremendous amount of internal states. The states are encoded as binary bit vectors corresponding to signal patterns being stored in the memory elements of the circuit. Representing large sets of states is one of the main problems in sequential circuit verification.

In the following sections it will be shown that the concept of an implicant in a multi-level combinational circuit can also be very useful for implicitly representing large state sets. This forms the basis for the *structural FSM traversal* introduced in Chapter 4.

3.8.1 Characteristic Function and Cap Circuit

When using Boolean techniques, sets of bit vectors can be represented using the concept of the *characteristic (Boolean) function* [26, 91] of the set. The following definition is taken from [91].

Definition 3.14 (Characteristic Function) *Let E be a set and $A \subseteq E$. The* characteristic function *of A is the function $\chi_A : E \longmapsto \{0, 1\}$ defined by $\chi_A(x) = 1$ if $x \in A$, $\chi_A(x) = 0$ otherwise.*

When considering the state space of a sequential circuit the set E is equal to $B^n = \{0, 1\}^n$ and is the set of all 2^n combinations of binary value assignments to a vector $\underline{x}$ of n Boolean variables ("bit vector" or "bit pattern"). In Chapter 4 these variables are referred to as "state variables" and the bit patterns correspond to the "states" of the sequential circuit. A is a set of such states. The characteristic function χ_A evaluates to 1 for every state in A and to 0 for every state not in A. In other words, the patterns in A define the ON-set of function χ_A.

Such a representation for a set is convenient, because it is a scalar function of n variables and can efficiently be represented and manipulated as a BDD. This concept forms the basis for many state-of-the-art algorithms used in optimization and verification of sequential systems. In these applications, large sets (e.g., sets of don't-care conditions, sets of states, transition relations) are represented not explicitly but "symbolically" by their characteristic functions. We will discuss these issues in some more detail in the next chapter.

Since the algorithms developed so far operate on combinational networks, it is worthwhile investigating how to represent sets of bit vectors in a structural form as gate netlists.

Of course, the characteristic function $\chi_{\underline{x}}$ of a set A of vectors $\underline{x} = (x_1, x_2, \ldots, x_n)$ can also be represented as a combinational network (Figure 3.31). This is called *cap circuit* in the sequel. The cap circuit has a single output which is associated with the characteristic function of the

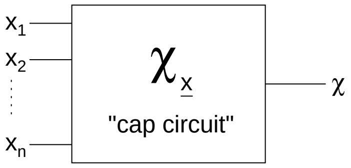


Figure 3.31: Circuit representation of a characteristic function

set A. For every combination of value assignments at the inputs x_i representing a vector of the set the output χ assumes the value 1. For every vector not in the set the output χ is 0. The cap circuit implements the ON-set of function $\chi_{\underline{x}}$.

How can we use this kind of set representation? Consider, for example, the application of optimizing a network under the assumption that don't care conditions exist for the primary inputs. The Boolean input space can be divided into two sets of vectors, the *care* set and its complement, the *don't care* set. We can represent, e.g., the care set by a cap circuit implementing its characteristic function and attach it to the primary inputs of the network as shown in Figure 3.32. The output χ of the circuit evaluates to 1 whenever an input vector is applied that belongs to the

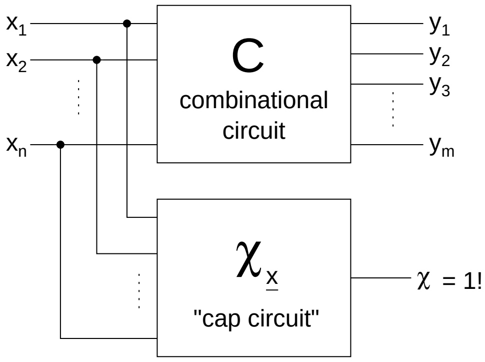


Figure 3.32: Circuit with don't-care set represented by cap circuit

care set, and to 0 if the vector belongs to the don't-care set. When optimizing the network C, we calculate implicants for internal nodes. In order to take the don't care conditions into account, we simply add the assignment $\chi = 1$ to the initial set of value assignments passed as arguments to procedure *and_or_enumerate()*. Value assignments belonging to the don't-care input space then produce a reasoning conflict and are excluded from the implicant calculation. In this way, we have effectively restricted the reasoning to those areas of the search space corresponding to the care set.

3.8.2 Controllability Don't Cares and Tap Circuit

There is another possibility to structurally represent a set. It does not require an additional value assignment during AND/OR reasoning and has the advantage that it allows us to exploit structural circuit properties more easily when performing sequential equivalence checking as we will see in later chapters.

Consider a circuit with k inputs and n outputs implementing a function $\tau_{\underline{x}}$ defined as follows:

Definition 3.15 (Tap function) *Let $A \subseteq B^n$ be a non-empty set of n-bit patterns. A function $\tau_{\underline{x}}$: $B^k \longmapsto A$ is called* tap function *of A. A circuit as shown in Figure 3.33 with k input variables and n output variables implementing this function is called* tap circuit.

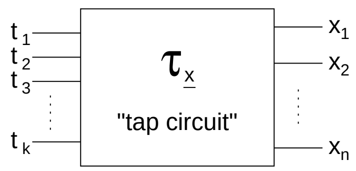


Figure 3.33: Set representation by the output range of a tap circuit

The set A is given by the set of vectors this circuit can produce at its outputs, i.e., it is determined by the *range* of the n-output function $\tau_{\underline{x}}$. Note that for the purpose of representing set A, we are actually not interested in the functional relationship between specific input patterns $\underline{t}$ and the output response $\underline{x}$ of the tap circuit. We only care about the set of possible output vectors, that is, the range of tap function $\tau_{\underline{x}}$. For a given set of vectors to be represented, arbitrarily many functions $\tau_{\underline{x}}$ with an arbitrary number k of input variables can be found that have this set as their range. Obviously, if there are n variables forming the output vector $(x_1, \ldots, x_n)$, there always exists a $\tau_{\underline{x}}$ with $k \leq n$ input variables $(t_1, \ldots, t_k)$ that has the set A as its range. The reason is that there are at the most 2^n vectors to be represented, each of which can be the image of a minterm of $\tau_{\underline{x}}$. Therefore, we need a domain of no more than 2^n points in the input space of $\tau_{\underline{x}}$, so that $k \leq n$ input variables are sufficient.

To see how a tap circuit can be used to represent a set of vectors, consider again the example of optimizing a combinational circuit exploiting don't cares. We connect a tap circuit as shown in Figure 3.34 to the primary inputs of the combinational circuit being optimized. All reasoning

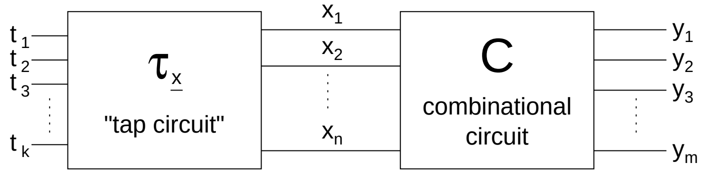


Figure 3.34: Circuit with don't-care set represented by tap circuit

performed in circuit C implicitly uses the don't-care conditions given by circuit $\tau_{\underline{x}}$. Any combination of value assignments at the inputs x_i which cannot be produced by $\tau_{\underline{x}}$ produces a logic inconsistency. In this case, the don't care set is represented by the complement set of the range of $\tau_{\underline{x}}$. When deriving implicants for network transformations, no additional value assignments have to be made to incorporate the don't-care set into the reasoning.

Note that a tap circuit is a very "natural" representation of a don't-care set: external don't cares in practical circuits often arise from the fact that the surrounding logic cannot produce certain vectors. This environmental logic in the transitive fanin of a considered logic block acts as a tap circuit. In multi-level logic optimization, a standard technique is to calculate the *controllability don't-care set (CDC)* of an internal logic block of a circuit which can then be used to simplify succeeding logic in the transitive fanout of the block. Since the controllability don't care set is the complement set of the range of a multi-output function, some approaches indeed use BDD-based image and range computation techniques to calculate the don't-care sets. See, e.g., [67] for an overview of these methods.

3.8.3 Synthesis of Set Representations, Stub Circuit

How are these two types of set representations related to each other? Consider a tap circuit and a cap circuit, both representing the same set of vectors A. The set A is equal to both, the range of the tap function $\tau_{\underline{x}}$, and to the ON-set of the characteristic function $\chi_{\underline{x}}$. To study the relationship between both circuits, we connect them such that the outputs of the tap circuit are the inputs of the cap circuit as shown in Figure 3.35.

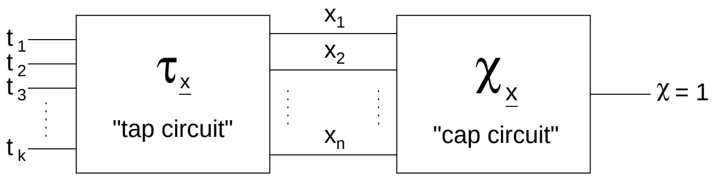


Figure 3.35: Tap and cap circuit representing the same set, combined

Obviously, since the characteristic function evaluates to 1 for every pattern being an element of the set A, and the tap circuit produces only patterns from A, the output χ in Figure 3.35 is constant 1. (A practical example for such a construction is the *miter* circuit introduced in Section 3.1. The two designs under comparsion with their inputs connected form the tap circuit. The two sets of outputs form the vector $\underline{x}$, and the output e of the XOR tree is associated with the complement of the characteristic function $\chi_{\underline{x}}$. For equivalent designs, $\chi = \overline{e}$ is constant 1).

The n-output function $\tau_{\underline{x}}$ and the single output function $\chi_{\underline{x}}$ have an interesting relationship:

Lemma 3.14 *Consider a function $\tau_{\underline{x}}$ with n primary outputs x_i and a range set A for the output vectors, a corresponding function $\chi_{\underline{x}}$ of n primary inputs x_i which represents the characteristic function of the set A, and a product term p of k literals corresponding to complemented or uncomplemented variables x_i. Every such product term which is a prime 0-implicant of $\chi_{\underline{x}}$*

corresponds to one prime 0- or 1-implicant of each of the k outputs x_j of $\tau_{\underline{x}}$ appearing in p. Analogously, every product term which is a 0- or 1-implicant of an output x_j of $\tau_{\underline{x}}$ corresponds to a 0-implicant of $\chi_{\underline{x}}$.

Proof: *see Appendix A, page 140*

The rule for converting a 0-implicant p of $\chi_{\underline{x}}$ into an implicant q of an output x_j of $\tau_{\underline{x}}$ can be extracted from the the above proof. The output x_j appears as literal l_j the product term p in complemented or uncomplemented form. The implicant q of x_j is obtained by simply removing the literal l_j from the product term p, i.e., it is $l_j \cdot q = p$. If $l_j = x_j$ then q is a 0-implicant of x_j. If $l_j = \overline{x_j}$ then q is a 1-implicant of x_j. For example, if $p = ab\overline{c}$ is a 0-implicant of $\chi_{\underline{x}}$ then ab is a 1-implicant of c, $a\overline{c}$ is a 0-implicant of b and $b\overline{c}$ is a 0-implicant of a.

This immediately leads to possible techniques for synthesizing one set representation from the other. If we have a tap circuit, we can calculate all prime 0-implicants and all prime 1-implicants of every primary output function of the tap function $\tau_{\underline{x}}$. Each of these implicants is converted into a 0-implicant of the characteristic function, and added to the OFF set of $\chi_{\underline{x}}$. We obtain a two-level POS-type description for the cap circuit which can be minimized or factorized into a multi-level circuit.

The opposite conversion is more difficult. We have the characteristic function $\chi_{\underline{x}}$ of a set A and would like to find a tap function $\tau_{\underline{x}}$ repesenting the same set A. Similarly as above, we are able to calculate all 0-implicants of $\chi_{\underline{x}}$ and convert them to 0- or 1-implicants for outputs of $\tau_{\underline{x}}$. However, these implicants are product terms whose literals correspond to other primary outputs. To construct a tap circuit, we need to find a functional representation between an auxiliary input vector $\underline{t}$ and the output vector $\underline{x}$ such that the derived implicant relationships among the output variables exist. The number of possible functions $\tau_{\underline{x}}$ is unlimited, because we may allow an arbitrarily large number of input variables and find a function that has set A as its range. To narrow the search space, we add some additional constraints in our definition of a tap function $\tau_{\underline{x}}$.

Definition 3.16 (Stub function) *Let $A \subseteq B^n$ be the set of n-bit patterns which can be produced at a vector of variables, $\underline{x}$. The function $\sigma_{\underline{x}} : B^n \longmapsto B^n$ is a special tap function for A with the following properties*

a) $\sigma_{\underline{x}}(\underline{x}) = \underline{x}$ *if* $\underline{x} \in A$

b) $\sigma_{\underline{x}}(\underline{x}) = \underline{y}$ *with* $\underline{y} \in A$ *if* $\underline{x} \notin A$

and is called a stub function *of A. A circuit with n input variables and n output variables implementing this function is called* stub circuit.

A stub function is a special case of a tap function. It has the same number of input variables as output variables. A stub circuit implementing this function works as a "pattern sieve". Every pattern applied at the inputs which is in the set A to be represented goes through unchanged. Every pattern which is not in the set A is modified such that a pattern from A appears at the outputs of the stub circuit. Thus, the stub function $\sigma_{\underline{x}}$ maps the OFF set of the characteristic function $\chi_{\underline{x}}$ onto its ON-set.

Corresponding input variables and output variables of a stub circuit have a similar meaning, because for all patterns in A an output x_j has the same value as its corresponding input t_j. For patterns not in A, at least one output x_j must have a different value than its corresponding input t_j. To create such a behaviour, we can use the 0-implicants of the characteristic function which are 0- or 1-implicants for output variables x_j. Whenever a pattern is applied to the stub circuit which is not in the set A, some implicant of some output x_j is "activated" such that x_j has the opposite value as t_j. The corresponding bit in the pattern is complemented by the implicant. This is the basic mechanism of the procedure shown in Table 3.10. However, special care must be taken such that the patterns obtained by this bit complementation do belong to the set A.

```
synth_stub(x̲, χ_x̲)
{
  /* x̲ is a vector of n variables x_i */
  /* χ_x̲ is a characteristic function on x̲ representing a non-empty set A */

  initialize C as a circuit with n inputs named t_i and n outputs named x_i,
    with corresponding inputs and outputs connected: x_i = t_i;
  determine a set S of prime 0-implicants representing a cover of χ_x̲;
  p := 1;    /* product term describing mapping subspace */

  for i := 1 to n
  {
    if (p·x_i is a 0-implicant of χ_x̲)
      l_i := ‾x_i;
    else
      l_i := x_i;

    I := the set of all prime implicants from S
      which represent prime 1-implicants of l_i:

    convert implicants in I to implicants of x_i;
    modify C by adding these implicants to the function of x_i;

    S := S \ I;
    p := p · l_i;
  }
  redundancy removal in C;
  return C;
}
```

Table 3.10: Synthesizing a stub circuit from a characteristic function

The procedure *synth_stub()* works in the following way. The arguments passed to the procedure are a vector of variables, $\underline{x}$, and a characteristic function, $\chi_{\underline{x}}$, of a non-empty set A. (Note that an empty set cannot be represented by a stub circuit). Let us call the patterns in A *correct* patterns, and the patterns not in A *wrong* patterns. The procedure returns a circuit C implementing a stub function $\sigma_{\underline{x}}$ for A.

The stub circuit C is initialized with n inputs and n outputs such that each output x_i implements the identity function of its corresponding input t_i. From the characteristic function $\chi_{\underline{x}}$ a set of prime implicants S covering the 0-minterms is determined. These implicants are converted to implicants for the primary outputs x_i of C. In case $\chi_{\underline{x}}$ is represented by a cap circuit, the methods described in section 3.7 can be used to determine a cover.

The *for* loop considers one output variable after the other. For each variable x_i, either all prime 0-implicants of x_i or all prime 1-implicants of x_i are determined and implemented. Consider the case where the 1-implicants of x_i are determined and added to the function of x_i. Whenever a wrong pattern is applied to the stub circuit that has $x_i = 0$, it either belongs to a prime 1-implicant of x_i or it belongs to an implicant that is independent of x_i. In the former case, the 1-implicant forces the output x_i to become 1. In the latter case, the pattern cannot be corrected using variable x_i. It has to be corrected by one or more of the remaining variables. Therefore, the implicants implemented for the remaining variables are expressed in terms of the current output variables x_i (and not the auxiliary input variables t_i). This is an important detail which is crucial for the correctness of the stub function.

In each step of the *for* loop, the wrong patterns in a certain subspace of the Boolean input space of $\sigma_{\underline{x}}$ are mapped to a remaining *mapping space* which becomes smaller with every iteration. In each step, it must be guaranteed that there still exist 1-minterms (correct patterns) in the remaining mapping space to which the wrong patterns can be mapped. This is checked using a product term p that describes the mapping subspace remaining after each iteration. If the subspace to which wrong patterns are to be mapped contains no 1-minterms the product $p \cdot x_i$ must be a 0-implicant of $\chi_{\underline{x}}$. (Note that in this case the product $p \cdot \overline{x_i}$ cannot be a 0-implicant either, otherwise p itself would be a 0-implicant of $\chi_{\underline{x}}$which means that the check in the previous iteration $(i - 1)$ had returned a wrong result).

Whenever a 0-implicant q in the cover S is implemented, all wrong patterns that belong to q are corrected. The order in which these implicants are implemented as 0- or 1-implicants of the primary outputs x_i of the stub circuit guarantees that the patterns produced by the circuit are correct even if several bits are complemented.

Note that depending on the 0-cover S determined from $\chi_{\underline{x}}$ it is possible that the resulting stub circuit contains redundancies which can be removed before returning the result. The following example illustrates how a stub circuit can be generated from a characteristic function $\chi_{\underline{x}}$ using procedure *synth_stub()* of Table 3.10.

Example 3.13 Consider the characteristic function $\chi_{\underline{x}}$ of the following set of values for the vector $\underline{x} = (a, b, c, d)$: $A = \{0000, 0100, 0101, 1000, 1010\}$. Figure 3.36 shows a Karnaugh map of $\chi_{\underline{x}}$. The prime 0-implicants of $\chi_{\underline{x}}$ are: ab, $\overline{a}c$, $\overline{b}d$, bc, ad, cd. The implicants $S = \{ab, \overline{a}c, \overline{b}d\}$ represent a cover of $\chi_{\underline{x}}$ as illustrated in Figure 3.36.

In the *for* loop, we start with variable a. We check whether it is impossible to derive 1-implicants, i.e., whether the subspace to which wrong patterns would be mapped

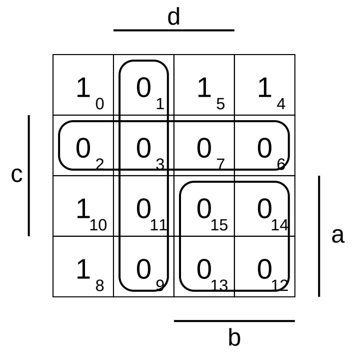


Figure 3.36: Characteristic function $\chi_{\underline{x}}$ and a 0-cover

contains no 1-minterms of $\chi_{\underline{x}}$. This is done by testing whether $p \cdot a = 1 \cdot a = a$ is a 0-implicant of $\chi_{\underline{x}}$. This is not the case, so we may derive the prime 1-implicants of a. The only prime 1-implicant found is c. (It corresponds to the 0-implicant $\overline{a}c$ of $\chi_{\underline{x}}$, encircled in Figure 3.37). We implement this implicant, remove it from the list S and set the product term describing the remaining mapping space to $p = a$ (shaded grey in Figure 3.37).

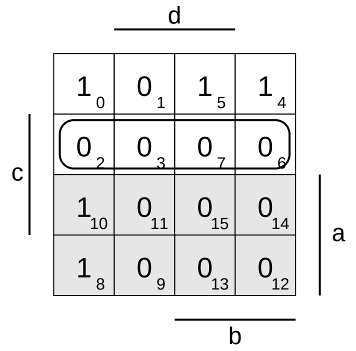


Figure 3.37: Prime 1-implicant of a: c

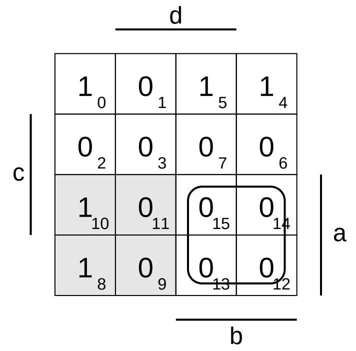


Figure 3.38: Prime 0-implicant of b: a

We continue with variable b. Again we check whether it is possible to implement prime 1-implicants by testing whether $p \cdot b = ab$ corresponds to a 0-implicant of $\chi_{\underline{x}}$. This time there are indeed no 1-minterms of $\chi_{\underline{x}}$ in this subspace. Therefore, we have to use the prime 0-implicants of b. From S we obtain the implicant a as indicated in Figure 3.38. The remaining mapping subspace is described by $p = a\overline{b}$ (shaded grey in the figure).

The next variable is c. The product $p \cdot c = a\overline{b}c$ is not a 0-implicant of $\chi_{\underline{x}}$, so we may determine prime 1-implicants of c. However, there are none. (c has only 0-implicants: $\overline{a}$, b, d). We need not implement any implicant, hence any value at the c-input of the final stub circuit propagates unchanged to the c-output. The remaining mapping subspace is set to $p = a\overline{b}c$.

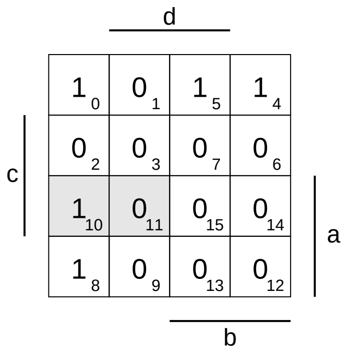


Figure 3.39: No prime 1-implicants of c

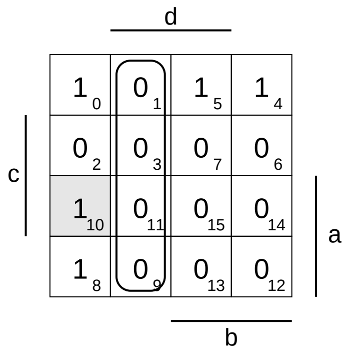


Figure 3.40: Prime 0-implicant of d: $\overline{b}$

For the last variable, d, the product $p{\cdot}d = a\overline{b}cd$ is a 0-implicant of $\chi_{\underline{x}}$, so that we need to derive 0-implicants of d. There is only one implicant left in S: $\overline{b}$ is a 0-implicant of d. We implement this implicant. The final stub cicuit is shown in Figure 3.41.

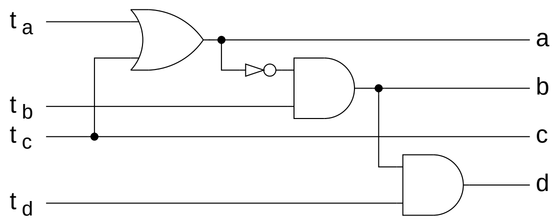


Figure 3.41: Stub circuit derived for characteristic function of Figure 3.36

3.8.4 Existential Quantification

Using procedure *synth_stub()* we can calculate a stub circuit from a characteristic function $\chi_{\underline{x}}$. This allows us to synthesize a stub circuit from a cap circuit such that both represent the same set of patterns, A. In addition, we are now also capable of synthesizing a stub circuit from an arbitrary tap circuit, using the concept of a characteristic function. This is very interesting, because it allows us to reduce the complexity of a set representation. Consider a tap circuit representing a set $A \subseteq B^n$ which has k inputs with $k \gg n$. Since we are not interested in the functional relationship between input patterns and output patterns but merely in the question which patterns exist in A and which do not, generating a stub circuit with n inputs helps us to get rid of unneeded information and to significantly reduce the complexity of our set representation. In some sense, this operation can be understood as a way of performing *existential quantification*, which is a fundamental operation in conventional image computation and FSM traversal algorithms.

When performing this existential quantification using procedure *synth_stub()*, we actually do not need to explicitly represent the characteristic function of the range of the tap circuit. All 0-implicants of $\chi_{\underline{x}}$ can be determined as 0- or 1-implicants of the primary outputs of the tap circuit according to Lemma 3.14 (page 73). In the pseudo-code of the procedure, S is not required to be a *minimal* cover of the 0-minterms of $\chi_{\underline{x}}$ (although a non-minimal cover produces a larger stub circuit than a minimal one). A cover can be obtained by determining all 0-implicants or all 1-implicants of each primary output x_i. This can be done on-the-fly when iterating through the n primary output variables in algorithm *synth_stub()*.

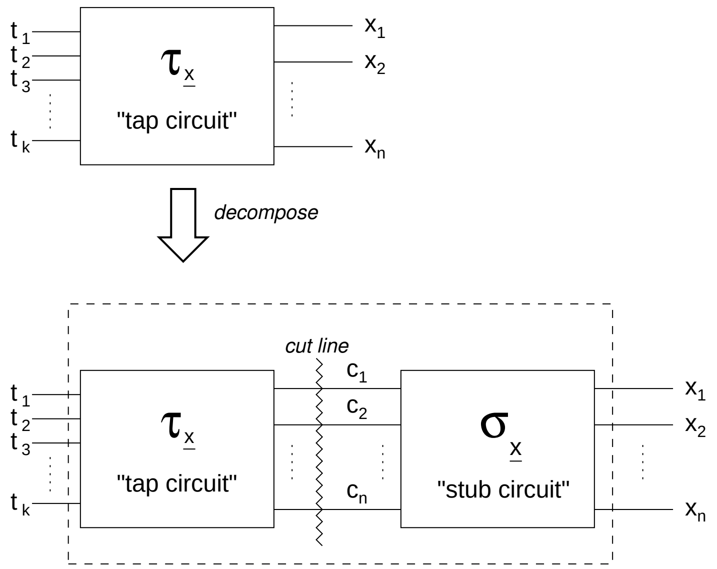


Figure 3.42: Existential quantification by decomposition and cut

Consider the tap circuit in Figure 3.42. When generating the stub circuit $\sigma_{\underline{x}}$ as described above, the concatenation of both, the tap circuit and a stub circuit representing the range of the tap circuit yields a circuit that is equivalent to the original circuit. The tap circuit shown at the top of Figure 3.42 is equivalent to the circuit represented by the dashed box shown at the bottom. When the stub circuit is being derived using procedure *synth_stub()*, adding implicants to the functions of the outputs x_i is a transformation preserving equivalence of the combined circuitry. Therefore, the synthesis of the stub circuit can also be seen as a functional decomposition of the tap circuit into the two components shown. Then, cutting through the circuit along the indicated cut line between the components performs the existential quantification operation. The functionality of the original tap circuit is lost. Only the range of the circuit remains, represented by the stub circuit.

This structural approach to existential quantification plays an important role in the structural *FSM traversal* to be described in the next chapter. For an example of existential quantification based on the above decomposition and cut, see Example 4.2 in Section 4.4 (page 93).

Chapter 4

Structural FSM Traversal

The research described in this thesis has the goal of setting up a framework for formal verification of sequential circuits which is based on structural techniques rather than conventional symbolic techniques. The key problem in sequential verification is reachability analysis which is associated with a traversal of the state transition graph of a finite state machine. This chapter treats the questions of how structural approaches to reachability analysis and FSM traversal can be formulated and how the notions and concepts describing conventional techniques for sequential verification are related to the concepts of structural FSM traversal.

The chapter begins with an examination of the problem of checking the equivalence of two sequential circuits, which is a typical application of FSM traversal. It introduces the basic idea for a sequential equivalence checking algorithm based on a time frame expansion of the product machine. (This algorithm will be further developed in Chapter 5). The nature of the underlying FSM traversal will be analyzed theoretically, and, based on this analysis, an exact algorithm for structural FSM traversal will be formulated making use of the implicant-based synthesis techniques developed in the previous chapter.

4.1 Introduction

In the introduction to Chapter 3 it was explained how structural techniques like implicant-based network optimization can be effectively applied to combinational equivalence checking using a so-called *miter* circuit. The question arises of how these techniques can be extended to be applicable to the verification of sequential circuits.

A sequential circuit is composed of memory elements such as latches and registers and combinational circuitry implementing state transition and output functions. We model a sequential circuit by a *Finite State Machine (FSM)*. A finite state machine M is described by a 6-tuple $M = (I, S, \delta, S_0, O, \lambda)$ where I is the input alphabet, S is the set of states, $\delta : S \times I \longrightarrow S$ is the next-state function, S_0 is a set of initial states, O is the output alphabet and $\lambda : S \times I \longrightarrow O$ is the output function. For simplicity we restrict our discussion to a single initial state $S_0 = \{s_0\}$. However, a set S_0 containing several initial states can be treated in a similar way.

A sequential circuit implementing such a finite state machine has a set of *(primary) inputs* $\underline{x}$, a set of *(present) state variables* $\underline{s}$ which are fed by registers, a set of *next-state variables* $\underline{z}$ providing the input of the registers and a set of *(primary) output variables* $\underline{y}$.

We use the following notion of equivalence. Two sequential circuits A and B are considered equivalent if their reset states are equivalent. This means that both machines produce the same output values for every possible input sequence if they both start in their respective initial states. In this thesis we assume that either the initial state (or a set of initial states) is known for each machine, or an initializing sequence is given to bring the circuit into a well-defined state after power-up.

Based on the concept of reset equivalence, we can reformulate the sequential equivalence checking problem using a miter construction as introduced for combinational equivalence checking in Chapter 3. Figure 4.1 shows a miter circuit constructed for two sequential circuits A and B. Just like in the combinational miter of Figure 3.1, corresponding inputs of the designs are connected and corresponding outputs are checked by a comparator with output e. In standard

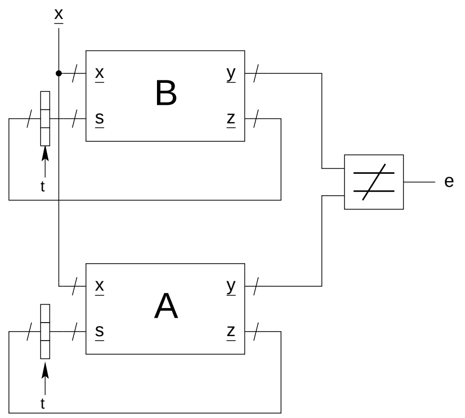


Figure 4.1: Miter circuit of designs A and B

literature [39], this construction is called *product machine for equivalence checking* of the two FSMs A and B. The name emphasizes that the state set S of the composed machine is given by the product $S = S_A \times S_B$ of the state sets S_A and S_B of the individual machines.

Our goal is to use the combinational techniques developed in Chapter 3 for this product machine. The idea is to use a *time frame expansion model* of the product machine in order to be able to consider transformations across register boundaries. We expand the sequential miter into time frames by breaking up all feedback loops and by replicating the combinational logic for each clock cyle being considered, as shown in Figure 4.2. Each time frame is a copy of the combinational logic implementing the transition function $\delta(\underline{s}, \underline{x})$ and output function $\lambda(\underline{s}, \underline{x})$ of the FSM. The circuit structure obtained by this time frame expansion is a purely combinational structure called *iterative circuit array* or *miter array*. There are no storage elements. Instead, the values of the state variables of the product machine at different points in time are associated with signal values in different time frames of the iterative circuit array.

In this construction it becomes more obvious why the sequential equivalence checking problem is much harder than combinational equivalence checking. To prove that the designs are

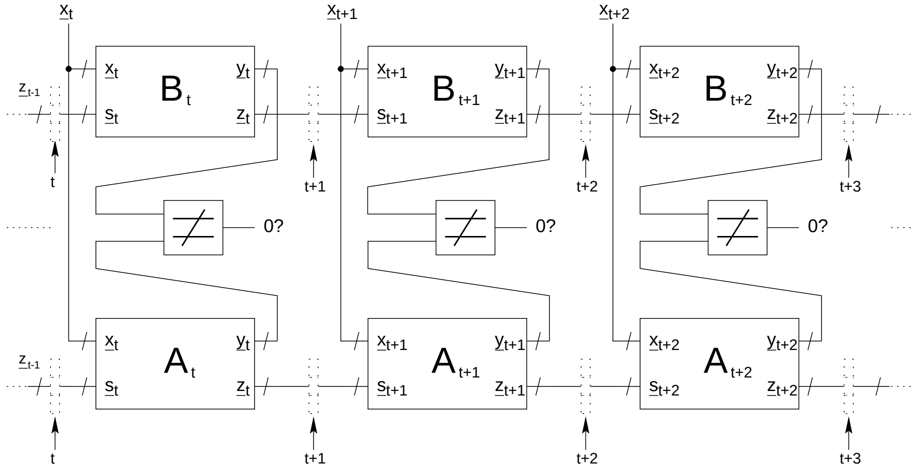


Figure 4.2: Iterative circuit array of product machine for equivalence checking ("miter array")

equivalent, we have to show that output e is 0 in *every time frame* for $t = 0, 1, 2, 3, \ldots$ after the machines have been initialized, for any sequence of input values $\underline{x}_t$. We start our combinational comparison at $t = 0$ by assigning the values of the initial state to the state variables and forward imply the constant values. The circuit array can then be simplified by removing all constant nodes. We can now traverse the iterative circuit array by combinational techniques that in each time frame check whether or not a constant 0 is obtained at the output of the comparator.

The Boolean functions associated with the state variables between time frames in the miter array represent the states of the product machine at a specific point in time for a given input sequence. When moving from time frame to time frame to check the validity of $e = 0$ we are exploring the state space of the product machine. Since the number of states in the machine is finite we can stop this procedure after a finite number of time frames when we are sure that all possible states have been considered. However, without any additional concepts we can stop only after we have traversed 2^n time frames where n is the number of state variables of the product machine. This is impractical for all but the smallest designs. Fortunately, often the *sequential depth* of finite state machines is much smaller and all legal states can be reached after a relatively small number of clock cycles for many practical circuits. Additionally, the number of reachable states in the product machine of two designs A and B is usually much smaller than the total number of states in the state space $S = S_A \times S_B$. Obviously, we can finish the verification process at a time t_{fix} if we know at t_{fix} that no states of the product machine are reachable in later time frames that have not been reached before. How such a point in time t_{fix} can be determined will be the subject of Section 4.3.

Reachability analysis, i.e., determining the set of states R which are reachable from a given set of initial states S_0 is a fundamental problem in formal verification of sequential systems. A given property has to be checked for all states that can possibly occur in a system. For example, in sequential equivalence checking the property to be verified is the comparator output e being 0 in all states reachable from the initial states. Reachability analysis requires traversing the state transition graph of an FSM, this is also called *FSM traversal*. Before discussing how FSM traversal can be performed using an iterative circuit array, we take a look at the standard techniques which to-date are based on the use of binary decision diagrams.

4.2 Symbolic FSM Traversal

For the purpose of calculating the set of states reachable from a set of initial states, we can neglect the output function λ of the FSM. It is sufficient to consider the so-called *finite state transition structure (FST)* corresponding to the FSM given by $H = (I, S, \delta, S_0)$.

Viewed in terms of the *state transition graph (STG)* of the FST, the task of reachability analysis is to determine all nodes (states) to which a path starting at nodes in S_0 (initial states) exists. In standard (BDD-based) verification methods this task is usually accomplished by a breadth-first search starting at the initial states, as shown in Table 4.1. The procedure computes

```
reachable_state_set(δ, S0)
{
  t := 0;
  R_t := S0;    /* initial state set */
  loop
  {
    R_t+1 := R_t ∪ img(δ, R_t);
    if (R_t+1 = R_t)    /* fixed point reached */
      break;
    t := t + 1;
  }
  return R_t;
}
```

Table 4.1: Forward FSM traversal using breadth-first search

the set of reachable states using *image computations*. The image of a set of states S is the set of next states for each state in S considering all possible transitions for all possible inputs to the machine as given by the next state function δ. Viewed in the state transition graph, the image of a set of nodes S is the set of immediate successors of all nodes in S. Beginning with the set of initial states, $R = S_0$, the state set R is iteratively augmented by the set of all its immediate successor states using the image operation. This process is repeated until the set R represents a fixed point under the image given by δ.

In BDD-based verification methods, see, e.g., [26, 91, 15], sets are represented implicitly using characteristic functions (see Definition 3.14 on page 70) which are in turn represented as BDDs. Using this approach large sets of states can be represented and manipulated simultaneously and implicitly. Also, sets of *state transitions* can be represented in this manner. A BDD is used to represent the characteristic function of a transition relation. An element of the represented set is a triple $(\underline{s}, \underline{x}, \underline{z})$ representing a single transition in the FST from state $\underline{s}$ to state $\underline{z}$ under the input $\underline{x}$ as given by the transition function δ. The characteristic function of the transition relation is defined by

$$T(\underline{s}, \underline{x}, \underline{z}) = \prod_{i=1}^{i=n} (z_i \overline{\oplus} \delta_i(\underline{s}, \underline{x}))$$

where n is the number of state variables. It evaluates to 1 for every triple $(\underline{s}, \underline{x}, \underline{z})$ that represents a valid state transition in the FST. Using the transition relation we can now calculate the symbolic image of a set of states S. By forming the conjunction of the characteristic function $\chi(\underline{s})$ of S and the transition relation we obtain a function $f(\underline{s}, \underline{x}, \underline{z})$ that evaluates to 1 for every triple $(\underline{s}, \underline{x}, \underline{z})$ that represents a valid state transition in the FST *from a state in the set* S:

$$f(\underline{s}, \underline{x}, \underline{z}) = \chi(\underline{s}) \cdot T(\underline{s}, \underline{x}, \underline{z})$$

We are, however, not interested in the functional relationship between states in S and their successor states. We need a characteristic function of only the vectors $\underline{z}$ that are members of valid transitions described by $f(\underline{s}, \underline{x}, \underline{z})$. This can be obtained using a Boolean operation called *existential quantification*. Given a function of m variables $f(x_1, \ldots, x_m)$, the existential quantification of f with respect to x_i is

$$\exists_{x_i} f = f|_{x_i=0} + f|_{x_i=1}$$

where $f|_{x_i=0}$ and $f|_{x_i=1}$ denote the negative and positive cofactors of f with respect to x_i. Given this operator, we can now formulate image computation based on characteristic functions by

$$\iota(\underline{z}) = \exists_{\underline{x}} \exists_{\underline{s}}\, \chi(\underline{s}) \cdot T(\underline{s}, \underline{x}, \underline{z})$$

Here, $\iota(\underline{z})$ denotes the characteristic function of the image of a set of states, S, which in turn is represented by $\chi(\underline{s})$.

Existential quantification can be efficiently implemented as a BDD operator, so that the forward FSM traversal in Table 4.1 can be executed completely symbolically. By performing the existential quantification on the BDD representing $f(\underline{s}, \underline{x}, \underline{z})$ as given above, the abstracted variables are removed from the BDD which can significantly reduce its complexity. Although huge numbers of states can be represented and manipulated using BDDs, for many larger finite state machines reachability analysis cannot be completed because of complexity reasons. The major problems are peak BDD sizes during image computation and the size of the BDDs representing the reachable state sets. Several improvements address these problems in different ways. Finding a good variable ordering ([45, 80]) improves the efficiency of the BDD representations in order to increase the size of the problems that can be handled. Targeting reachability analysis, in many works divide-and-conquer approaches are used which decompose, partition or reencode the FSM, the set of reached states or the transition relation (e.g., [19, 17, 18]). A complementary approach is to use state space approximations on designs which are too large for an exact reachability analysis (e.g., [21, 75]). Despite all improvements made in the past, for many circuits of practical size reachability analysis still remains a severe problem. This has motivated our search for alternative forms of representation and algorithms.

4.3 FSM Traversal by Time Frame Expansion

In this section we examine how the state transition graph of a finite state machine is traversed using a time frame expansion as shown in Figure 4.2. When speaking of the states reached by a time frame expansion model of an FSM, we mean the set of vectors that can be produced by the iterative circuit array at the state vector $\underline{s}_t$ of a given time frame t. In the FSM being modelled as circuit array this set of reachable states $R(t)$ corresponds to the set of states which the machine can assume when considering all possible input sequences of length t.

Definition 4.1 (Reachable state set) *The set of all states, $R(t)$, being possible in an FSM at a specific time t after initialization is called* reachable state set at time t.

Note that $R(t)$ contains only the states that are reachable *exactly at time t*. This is different from the understanding of the reachable state set R_t in conventional FSM traversal, e.g., as used in Table 4.1. There, R_t refers to all states that can be reached at *some time t'*, with $t' \leq t$. Obviously, the sets of states that can be produced by the iterative circuit array of Figure 4.2 at the state variables $\underline{s}_t, \underline{s}_{t+1}, \underline{s}_{t+2}, \ldots$ are the sets of reachable states $R(t), R(t+1), R(t+2), \ldots$, respectively. Since this time series $R(t)$ differs from the sequence of sets R_t of a conventional FSM traversal, it is of interest to examine the nature of this series. The questions we are interested in and which will be answered in this section are:

- How is the state transition graph of an FSM traversed by the sequence of sets, $R(t)$?
- What is the convergence behaviour of $R(t)$?
- How can a point in time, t_{fix}, be determined for which it is guaranteed that all states reachable from the initial state have been visited?

The main difference between this FSM traversal and a conventional FSM traversal is the fact that after the set of new states $R(t+1)$ has been calculated in each iteration (given by a new time frame) no union is formed with the set of states reached so far. In order to find out how the state transition graph is traversed by the time series $R(t)$ and how its convergence behaviour is related to the convergence behaviour of conventional FSM traversal, it is necessary to analyze the structure of the state transition graph (STG).

The STG is a directed graph $G(V, E)$ with a set of vertices, V, corresponding to the states of the FSM and a set of edges, E. E is a relation on V. Each edge $(s, z) \in E$ is directed from a state $s \in V$ to a state $z \in V$ and signifies a possible state transition. The edges can be labelled with sets of input vectors for which the transition takes place.

Definition 4.2 (Reachability Relation) *For a state transition graph $G(V, E)$ the* reachability relation *R is defined as follows: $(s, z) \in R$ if there is a path from state s, $s \in V$ to state z, $z \in V$ ("z can be reached from s"), or if $s = z$.*

It is convenient to define this relation to be reflexive. Firstly, the set of states reachable from some initial state S_0 contains S_0 also in the case that the initial state S_0 does not have an explicit self-edge in the state transition graph. Secondly, the reachability relation can be used as a *weak ordering* of the state set V. A weak ordering relation R is a relation which is reflexive and transitive. By considering only state pairs which are mutually reachable from each other (i.e., by selecting the "symmetric elements" of R), we can define *strong connectivity* between states.

Definition 4.3 (Strong Connectivity) *Given the reachability relation R, the* strong connectivity relation S *is defined by:* $\quad (s, z) \in S \iff (s, z) \in R \wedge (z, s) \in R$

State pairs (s, z) *which are elements of* S *are called* strongly connected.

This relation S induces a partition of the state set V into equivalence classes. Each equivalence class consisting of more than one state has the the property that each of its states is reachable by any other state in the class. The equivalence classes are therefore called *strongly connected components (SCCs)* of the state transition graph. Note that this name can be somewhat misleading in the special case that an equivalence class contains only a single state which does not have a self-edge. Using this partition induced by the relation S we can define a new graph with the SCCs as vertices:

Definition 4.4 (SCC Graph) *The* SCC graph $G_S(V_S, E_S)$ *of a state transition graph* $G(V, E)$ *is defined as follows:*

1. V_S *is the set of equivalence classes of the partition induced by the strong connectivity relation* S *of* $G(V, E)$. *The classes are called* strongly connected components (SCCs) *of the STG.*

2. E_S *is defined such that there is a directed edge* $(s_S, z_S) \in E_S$ *between two distinct SCCs* s_S *and* z_S *if there exists an edge from an arbitrary state in* s_S *to an arbitrary state in* z_S.

The edge relation E_S of the SCC graph is anti-symmetric. If there is a directed edge from an SCC s_S to an SCC z_S then there cannot be any back-edge from z_S to s_S, because otherwise they would have to be in the same SCC. Hence, the SCC graph must be acyclic. The edge relation E_S of the SCC graph is a partial order among the set of SCCs, V_S. The sinks of the SCC graph are commonly referred to as *terminal strongly connected components (TSCCs).*

The SCC graph allows to decompose the state transition graph of a finite state machine into its "cyclic" parts. The states lying on cycles in the state transition graph determine the "long-run" or "steady-state" behaviour of the machine, because the machine can be in these states arbitrarily often. This allows the FSM to respond to infinitely long input sequences, even though the total number of possible states is finite.

The question we are interested in is: what is the long-run behaviour of the set of reachable states $R(t)$? Intuitively, since the FSM has a finite set of states, and its next-state behaviour is deterministic, the time series $R(t)$ must at some point in time enter a stationary behaviour. Either there will be a single final set R_∞ or $R(t)$ will periodically cycle through a set of state sets.

For analyzing the evolution of the reachable state set $R(t)$ over time, it is helpful to imagine $R(t)$ as a set of states in the state transition graph with a special mark that is passed on to successor states in the next time step. Consider a set of marked states, $R(t)$, at time t. One time step later, at $t+1$, these states pass their marks on to all of their immediate successor states. Their own mark is erased, unless they receive a new one from an immediate predecessor state. Now, all marked states form the reachable state set $R(t+1)$. As we proceed in time, the states in the STG become repeatedly marked and unmarked as described. If $R(t)$ converges to a final set R_∞, this set corresponds to a pattern of marks which beginning at a certain time t_{fix} does not change any more, i.e., it becomes a static pattern. If $R(t)$ exhibits a periodic long-term behaviour, this corresponds to periodically repeating patterns of marks in the STG, beginning at a certain time t_{fix}. As we will see, whether we have an aperiodic or a periodic behaviour is determined by the structure of the SCC graph and certain SCCs in it.

For the closer analysis of the SCC graph we need the following recursive definition:

Definition 4.5 (Recurrent State) *A state in the state transition graph G is called* recurrent state *if it lies on a cycle in G, or if it has a predecessor which is a recurrent state.*

Recurrent states are contained in the reachable state set $R(t)$ infinitely often, and they appear periodically. This is easy to see for recurrent states which are lying on a cycle (called *cycle states* in the sequel), by observing the $R(t)$–marks they are sending and receiving. The length of the cycle determines how many time steps it takes until a mark that has been sent out by a state returns back to it and is sent again. This time is called the state's *period of recurrence* and it is equal to the length of the cycle. However, there are also states which are not involved in a cycle but are nevertheless recurrent. Such states are reachable from cycle states, and therefore they receive, with some delay, all $R(t)$–marks which the cycle states send out. We say, a state *inherits* the recurrence periods of the states from which it can be reached. Note that in a particular run of the machine, a recurrent state that is not a cycle state can occur only once. However, for different runs of the machine it may occur at different times. The reachable state set $R(t)$ contains information about all possible runs of the machine. Therefore, a recurrent but non-cycle state is an element of $R(t)$ periodically just like a "true" cycle state.

Definition 4.6 (Transient State) *A state which is not recurrent is a* transient state.

Transient states are not reachable by recurrent states, they can be found "upstream" in the SCC graph. They occur only once in the reachable state set $R(t)$. Transient states are only possible if the initial state of the FSM is a transient state. They are usually part of the initialization process for the machine and do not belong to the normal mode of operation.

Note that our definitions of recurrent and transient states differ from literature concerned with a probabilistic analysis of finite state machines such as [37, 38]. The objective there is to determine the long-run probability for an FSM to be in a certain state. In that context, for example, a state is defined transient if there is a non-zero probability that the FSM will not return to it. Our definition requires that it is *impossible* to return to a transient state. In case there is a *possibility* that the FSM returns to a state, we call it a recurrent state, because it will be an element of the reachable state set, even if the *probability* for this to happen may be zero.

Since transient states occur only once, they cannot be part of the reachable state set in the fixed point we are seeking. So we can focus the analysis of our FSM traversal solely on the recurrent states. Transient states do not have a recurrence period, however, they are represented in the

SCC graph as individual vertices. Therefore, when speaking of strongly connected components in the sequel, we refer to SCCs containing recurrent states.

As discussed above, a recurrent state may inherit recurrence periods from its predecessor states. It also has periods of recurrence associated with the cycles on which it is located. The following lemma tells us how these different recurrences interact.

Lemma 4.1 *Consider an arbitrary state s lying on a cycle of the state transition graph of length q. Furthermore, let s have a recurrence period p. Then, after a finite number of time steps, state s also has a recurrence period p_g which is the greatest common divisor of p and q.*

Proof: *see Appendix A, page 140*

It can be shown [89] that the time t_{trans} it takes until s occurs every p_g time steps is upper-bounded by the least common multiple of p and q.

The recurrence period p in the lemma may be inherited from a predecessor state. Or it may be due to another cycle of length p that state s is lying on. We can apply Lemma 4.1 successively to all pairs of periods p and q that a state has due to period inheritance and the cycles in which it is involved. This leads us to an interesting lemma for the states of an SCC:

Lemma 4.2 *After a finite transition time, the smallest recurrence period, p_{SCC}, is the same for all states in an SCC. This period p_{SCC} is given by the greatest common divisor of all cycle lengths in the SCC and of all recurrence periods for states in the SCC that have been inherited from predecessor states outside the SCC.*

Proof: *see Appendix A, page 141*

If we view again the set of reachable states, $R(t)$, as a set of states in the STG carrying a mark, then this lemma says that after a sufficient amount of time each state in an SCC will be marked periodically. If the period is, for example, $p_{SCC} = 5$, then, a state will be marked in every fifth time step and will be unmarked during the remaining four time steps. Since at every time step at least one state of the SCC is marked, we can partition the states in the SCC into equivalence classes of simultaneously marked states:

Lemma 4.3 *The set of states of an SCC with a recurrence period p_{SCC} can be partitioned into p_{SCC} disjoint subsets C_m with $m \in \{0, 1, ..., (p_{SCC} - 1)\}$, such that $C_m = R(t_{SCC} + k \cdot p_{SCC} + m)$, for all integers $k \geq 0$.*

Proof: *follows immediately from Lemma 4.2*

This is an interesting first result. Let us consider the special case of a finite state machine that has all its states including the initial state within one strongly connected component. Such systems are sometimes called *non-decomposable* sequential systems, because the SCC graph of their state transition graph consists of only a single vertex. In this case, the set of reachable states, $R(t)$, converges to a series of final sets C_m that oscillate with a period p_{SCC} related to the structure of the STG.

It is possible, however, that p_{SCC} is equal to 1, in which case there is only a single final set C_0 into which $R(t)$ converges. This is the aperiodic behaviour mentioned in the beginning of this section. In fact, such a behaviour is very common in FSMs describing practical systems. A "degenerated" period of one time step is the direct result of a state in the SCC which has a self-edge, or of cycles with lengths whose greatest common divisor is 1.

One way of obtaining a non-decomposable system is by modelling the process of initialization within the FSM description. The FSM can then be put into its initial state by applying a special input sequence called *initializing* or *synchronizing sequence*.

Definition 4.7 *A* synchronizing sequence *of a finite state machine M is an input sequence that brings M to a known state s_0 regardless of the initial state or the output sequence. The state s_0 is called* synchronization state *of M.*

If a finite state machine has a synchronizing sequence, then the synchronization state s_0 and all states reachable from it must be located within one terminal strongly connected component (TSCC). The reason is that from all states reachable from s_0 the machine can be put back into s_0 by applying the synchronizing sequence. In other words, s_0 is reachable from all states which are reachable from s_0. Hence, machines with synchronizing sequences are non-decomposable.

In the special case of a non-decomposable system, the basic definitions of transient and recurrent states of an FSM and those of a Markov chain are equivalent. Therefore, the following lemma which was originally derived in [38] for homogeneous discrete-parameter Markov chains with a finite state space can also be formulated in this context.

Lemma 4.4 *If a finite state machine has a synchronizing sequence, then the fixed point recurrence period of all its states is 1.*

Proof: see Appendix A, page 141

Lemma 4.5 *If a finite state machine has a synchronizing sequence of length l and if d is the sequential depth of the machine, then it takes at most $l + d$ time steps until all states of the machine are in $R(t)$ and have a recurrence period of 1.*

Proof: see Appendix A, page 141

Note that the *sequential depth* of a finite state machine is given by the longest path among all shortest paths from the initial state to all nodes in the state transition graph of the FSM. In conventional symbolic FSM traversal, the sequential depth is equal to the number of iterations needed to reach the fixed point.

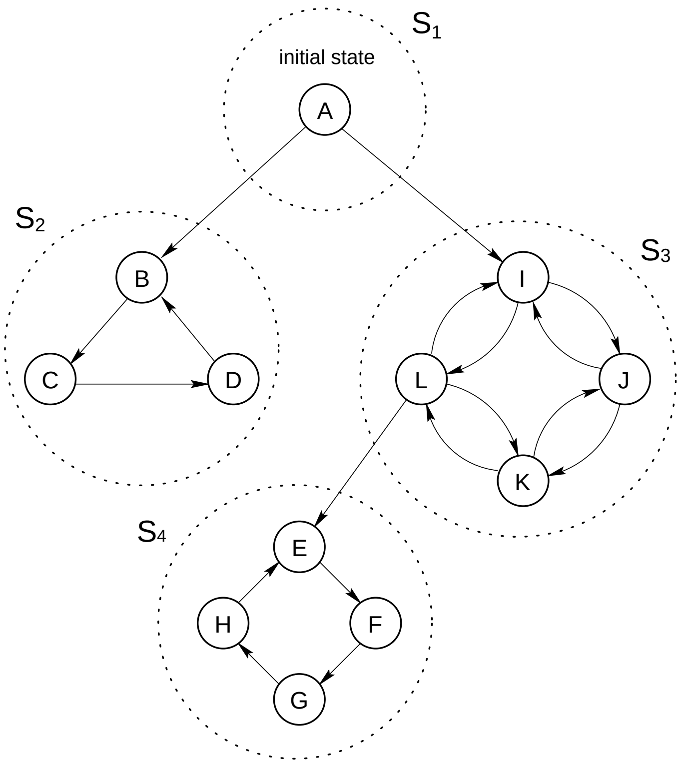
Although most finite state machines encountered in practice actually fall into the category of non-decomposable systems [70], it is generally possible that the SCC graph contains more than one SCC. Also, the initial state does not have to be a recurrent state. We therefore need to discuss the general case of an arbitrarily structured SCC graph.

If the SCC graph of a state transition graph has more than one SCC vertex, then different recurrence periods arise in different regions of the graph. These periods are inherited along the directed edges between the SCCs. Each period inherited through an incoming edge of an SCC "interferes" with the cycle lengths in the SCC according to Lemma 4.2. The periods leaving the SCC can never be greater than the periods coming in, because the greatest common divisor of a set of numbers is never greater than the numbers themselves. In fact, the period of recurrence of an SCC is always an integer multiple of the periods of its (transitive) successor SCCs. This means that the largest periods of recurrence are found in the "earliest" SCCs after initialization of the FSM. We call them *entry SCCs (ESCCs)*.

Definition 4.8 (Entry SCC) *An SCC in the state transition graph which is entered by initialization or reached exclusively via transient states after initialization is called* entry SCC (ESCC).

Note that an entry SCC need not necessarily be a source of the SCC graph. Only if the initial state is a recurrent state there is a unique entry SCC which is also the source of the acyclic SCC graph. If, however, the initial state is a transient state, then there can be several entry SCCs.

Example 4.1 Figure 4.3 shows an example of such a state transition graph.



t	$R(t)$	C_j
0	$\{A\}$	—
1	$\{B, I\}$	—
2	$\{C, J, L\}$	—
3	$\{D, E, I, K\}$	—
4	$\{B, F, J, L\}$	—
5	$\{C, E, G, I, K\}$	C_0
6	$\{D, F, H, J, L\}$	C_1
7	$\{B, E, G, I, K\}$	C_2
8	$\{C, F, H, J, L\}$	C_3
9	$\{D, E, G, I, K\}$	C_4
10	$\{B, F, H, J, L\}$	C_5
11	$\{C, E, G, I, K\}$	C_0
12	$\{D, F, H, J, L\}$	C_1
13	$\{B, E, G, I, K\}$	C_2
...	...	...

Figure 4.3: A state transition graph and the first values of its reachable state set $R(t)$

This STG is composed of four SCCs: $S_1 = \{A\}$, $S_2 = \{B, C, D\}$, $S_3 = \{E, F, G, H\}$ and $S_4 = \{I, J, K, L\}$. S_1 is an SCC without cycles. It contains only the initial state A which is a transient state. S_2 and S_3 are entry SCCs according to Definition 4.8. S_4 is a terminal SCC.

Also shown in Figure 4.3 are the first values of the time series of the reachable state set, $R(t)$. The initial state is contained in $R(t)$ only once for $t = 0$. The remaining states are recurrent, and we can easily verify that Lemma 4.1 and Lemma 4.2 are correct. In SCC S_2 there is a unique cycle of length 3. Therefore, the states B, C and D are contained in $R(t)$ alternately every three time steps. In SCC S_3 there are several cycles whose lengths all are multiples of 2. Hence, the states in S_3 recur in $R(t)$ in two alternating sets, $\{I, K\}$ and $\{J, L\}$. The SCC S_4 contains only one cycle of length 4. However, the recurrence period of its states is 2. The reason for this is that whenever state L (of SCC S_3) is in $R(t)$, state E will be in $R(t + 1)$, one time step later. State E inherits state L's recurrence period. The greatest common divisor of the inherited period of 2 and the cycle length of 4 is 2.

Obviously, after a sufficient amount of time ($t = 5$), all recurrence "interferences" have taken place and a stationary oscillation has evolved. In this example, there are six different values for the set of reachable states, $R(t)$, which are repeated in the same order with a period of six time steps. These six sets together form a cover of the set of all recurrent states.

Theorem 4.6 *Let P be the set of all recurrent states of a finite state machine. There always exists a cover $\{C_0, C_1, ..., C_{T-1}\}$ of P with $C_k \in 2^P$, and there is a time $0 \le t_{\text{fix}} \le \infty$, such that*

$$R(t_{\text{fix}} + n \cdot T + k) = C_k, \quad 0 \le n, \quad 0 \le k < T$$

T is equal to the least common multiple of the recurrence periods of the entry SCCs *of the FSM. T is called* fixed point oscillation period.

Proof: *see Appendix A, page 142*

In our example of Figure 4.3 there are two entry SCCs, S_2 and S_3, with periods $p_2 = 3$ and $p_3 = 2$. The least common multiple of these two numbers is $T = 6$. This is the fixed point oscillation period of the reachable state set that we have found also empirically for this state transition graph.

It is interesting to note that for the fixed point oscillation period only the ESCCs of the state transition graph are relevant. All other SCCs including the TSCCs (unless they are, at the same time, ESCCs) do not influence the oscillation period T. (It should be noted, however, that the cycle lengths of non-entry SCCs determine how long it takes until the fixed point is reached.) This observation again points out the difference between a possibilistic state space analysis such as the characterization of the time series $R(t)$, and a probabilistic state space analysis [38], where the terminal SCCs play the important role in the analysis.

Theorem 4.6 justifies the formulation of a structural FSM traversal based on a time frame expansion of a finite state machine. By considering the states that can be produced by the state vectors $\underline{s}_t$ of the iterative circuit array we are able to traverse the state transition graph of the machine, visiting all states reachable from the set of initial states. For most practical systems (e.g., systems with synchronous resets or initializing sequences), the set of reachable states grows monotonically from time frame to time frame and the fixed point of the iteration consists of a single set, R_∞. For these systems, the number of iterations needed to reach the fixed point is only slightly larger (by the length of the initializing sequence) than in conventional FSM traversal.

4.4 A Structural Fixed Point Iteration

In order to formulate an FSM traversal algorithm based on a time frame expansion of the finite state transition structure, we need a method to determine that we have reached the fixed point of the expansion. The iterative circuit array represents the set of reachable states, $R(t)$, in a non-canonical and implicit form. Consider a segment of the array of t time frames starting at $t = 0$. The set $R(t)$ corresponds to the *range* of the n-output function implemented by this combinational circuit. At the fixed point ($t \geq t_{\text{fix}}$) it is $R(t) = R(t + T)$, with T being the fixed point oscillation period of the FSM according to Theorem 4.6. This means that the expansion into $(t + T)$ time frames results in a combinational network implementing a function with the same range as the expansion into only t time frames.

Note that in BDD-based FSM traversal it is easy to recognize the fixed point, because sets of states and their images under the transition function are stored canonically as BDDs. The BDDs representing R_t and R_{t+1} in Table 4.1 simply have to be checked for isomorphism to detect the fixed point.

In our case of structural representations of the sets $R(t)$ and $R(t + T)$ by time frame expansions of lengths t and $(t + T)$, respectively, this simple check is not possible. In addition, our structural representations for the reachable state sets can become very large if the number of time frames in the circuit array is high. Note that an iterative circuit array of k time frames contains the complete functional information relating input sequences of length k to the resulting states of the FSM. The information we are interested in, however, is only *what* states are possible, not *how* they can be produced in the machine. We are only interested in the *range* of the n-output function $\underline{s}_t$ implemented by an iterative circuit array of length k. Therefore, this array can be seen as a combinational circuit functioning as a *tap circuit* as introduced in Section 3.8.2 for the set of vectors $\underline{s}_t$. Using the decomposition introduced in Section 3.8.4 we can perform a structural form of existential quantification that gives us a representation of $R(t)$ which is independent of the sets of input variables $\underline{x}_0, \underline{x}_1, \dots, \underline{x}_{t-1}$. This greatly simplifies the set representation, and, most importantly, it allows us to detect the fixed point. If $R(t)$ and $R(t + T)$ are equal, it must be possible to decompose the miter arrays of length t and of length $(t + T)$ such that identical stub circuits are produced in both cases. This means that the fixed point check amounts to checking the *identity of the decomposition steps* used for the existential quantifications. In other words, if $R(t)$ and $R(t + T)$ are equal, it must be possible to derive the same implicants for a state variable $s_{t,i}$ (expressed in terms of the other state variables) as for the corresponding state variable $s_{t+T,i}$ just T time frames later. If redundancies are removed in procedure *synth_stub()* of Table 3.10, it must be possible to remove the corresponding redundancies also in the stub circuit T time frames later. In practical implementations, such as *record_and_play()* introduced in Chapter 5, an *instruction queue* can be used for the purpose of storing the decomposition steps in a well-defined way. Figure 4.4 shows the flowchart of the structural FSM traversal algorithm using these concepts.

Example 4.2 Consider the two circuits of Figure 4.5. For both circuits the structural netlist representation and the state transition graph are given. The state transition graph for the upper circuit has been "unfolded" so that both states, 0 and 1 appear

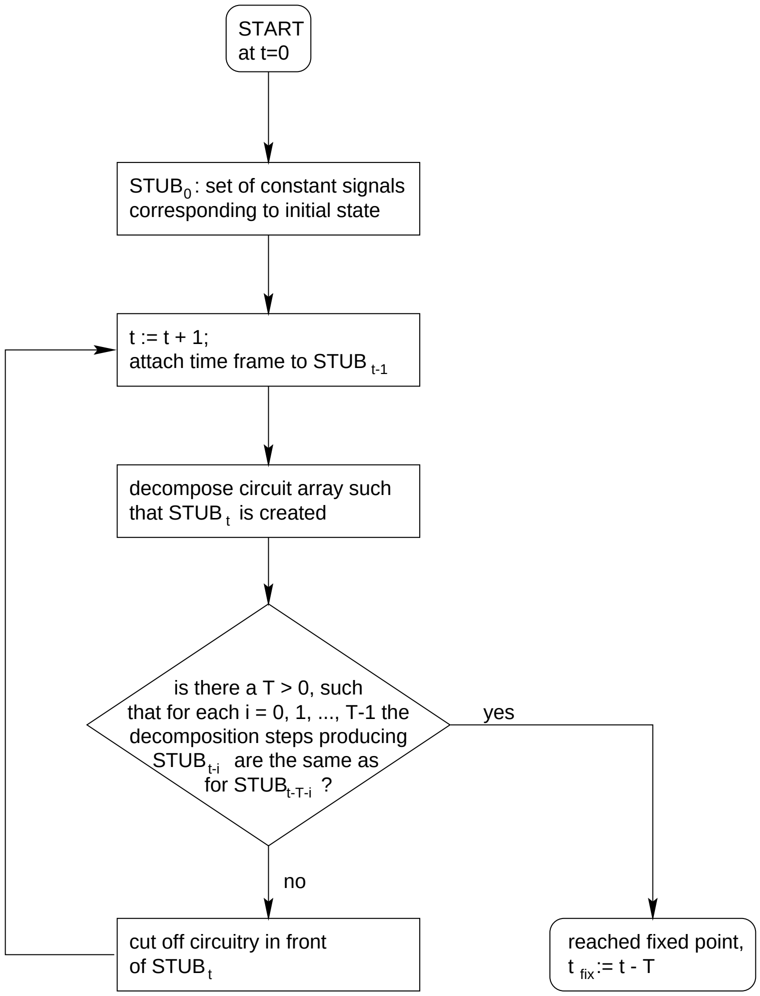


Figure 4.4: Algorithm for Structural FSM Traversal

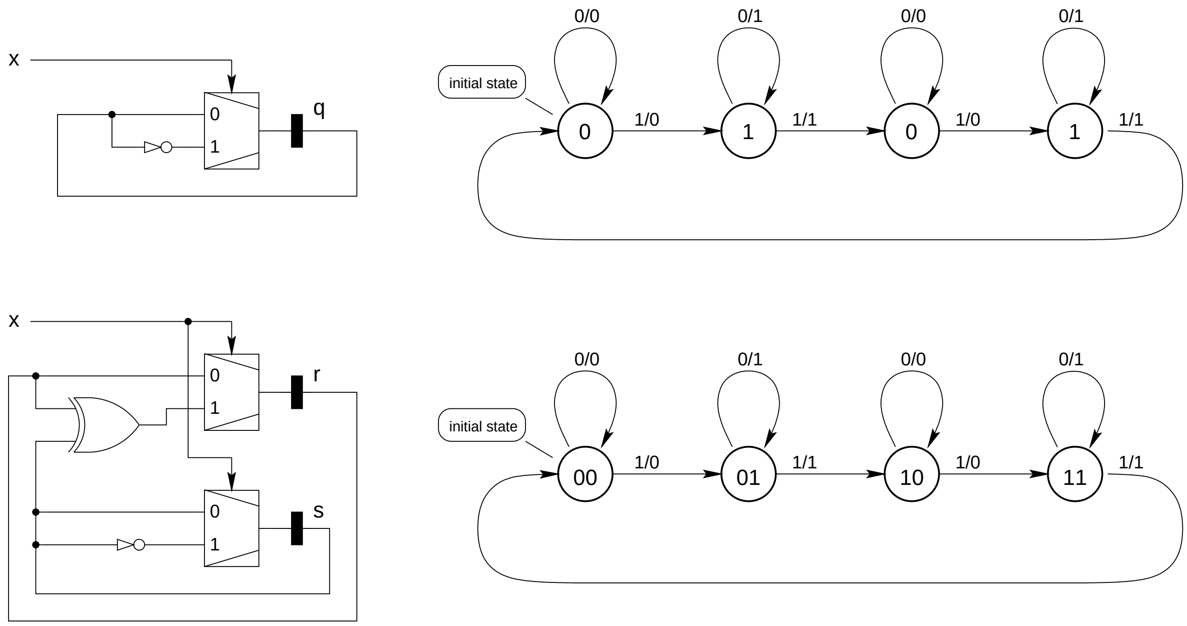


Figure 4.5: Example for two equivalent FSMs

twice. This makes it easier to see that the two machines are equivalent. (For reasons of clarity the output logic has been omitted. The shown circuitry implements the finite state transition structure of the FSMs.)

The two circuits are combined to form a product machine which is expanded into an iterative circuit array until time $t = 3$. The resulting circuit array is shown in Figure 4.6. Note that the circuit array has been simplified by assigning the initial state $\underline{s}_0 = (q_0, r_0, s_0) = (0, 0, 0)$ to the state variables q, r and s in the first time frame and by removing all constant nodes that result from the forward implications from the initial state. Not all states are reachable at a specific time. Table 4.2 lists the combinations of value assignments that are possible at the state variables for a specific t. The reader may verify that these sets are reachable by examining the

states (q, r, s) reachable at $t = 0$	states (q, r, s) reachable at $t = 1$	states (q, r, s) reachable at $t = 2$	states (q, r, s) reachable at $t = 3$
000	000	000	000
	101	101	101
		010	010
			111

Table 4.2: Reachable state sets $R(t)$ of product machine for $t = 0, 1, 2, 3$

state transition graphs of Figure 4.5. Note that the state variables (q_3, r_3, s_3) in the

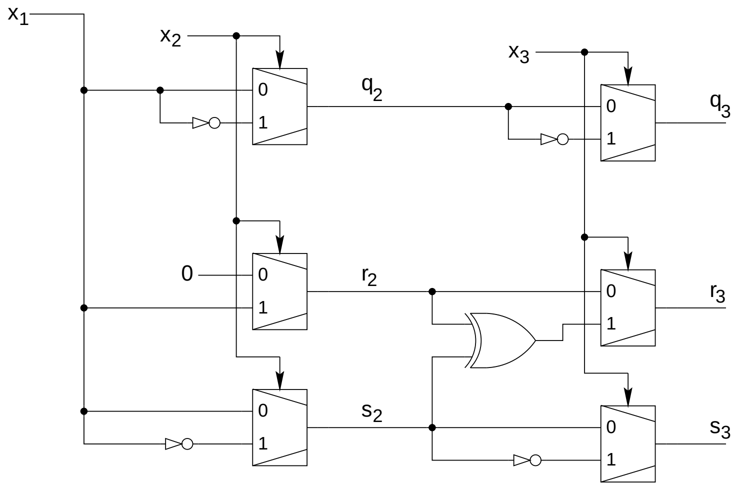


Figure 4.6: Iterative circuit array of product machine for circuits of Figure 4.5 until $t = 3$

iterative circuit array of Figure 4.6 can only assume the values as listed in Table 4.2 for $t = 3$. Hence, the stub circuit that will now be constructed must exclude all other combinations of value assignments.

The circuit array of Figure 4.6 is the tap circuit on which structural existential quantification as described in Section 3.8.4 of Chapter 3 is to be performed. The characteristic function $\chi_{\underline{S}}$ representing the range of the 3-bit function $\underline{s}$ of the circuit array is considered only implicitly. During execution of the algorithm *synth_stub()* as defined in Table 3.10 the 0-implicants of $\chi_{\underline{S}}$ are calculated on-the-fly as 0- or 1-implicants of state variables in terms of other state variables.

The procedure *synth_stub()* starts by examining state variable q_3 and calculates all 1-implicants of q_3. Figure 4.7 shows the AND/OR graph for the corresponding initial assignment $\{q_3 = 0\}$. The bold lines in the graph mark a MIST corresponding to the only prime 1-implicant, s_3, that can be expressed in terms of s_3, r_3 and q_3. (This implicant corresponds to the 0-implicant $\overline{q_3}s_3$ of the characteristic function $\chi_{\underline{S}}$). The 1-implicant of q_3 is implemented and added to the cover of q_3 using an OR gate.

Next, state variable r_3 is examined. Examining the AND/OR graph for $r_3 = 0$ (not shown) yields no 1-implicants. The cover of r_3 remains unmodified. For the last state variable, s_3, the 1-implicant q_3 is identified. It is implemented and added to the cover of s_3. Before returning, procedure *synth_stub()* performs redundancy removal within the created stub circuit. As shown in Figure 4.8, one input of an OR gate is found redundant stuck-at-0 and can be removed.

Note that the output functions implemented by the iterative circuit array combined with the stub circuit are functionally equivalent with the original outputs of the array. The existential quantification is completed by cutting through the inputs of the stub circuit. The set of reachable states, $R(3)$, is now represented in a compact way by

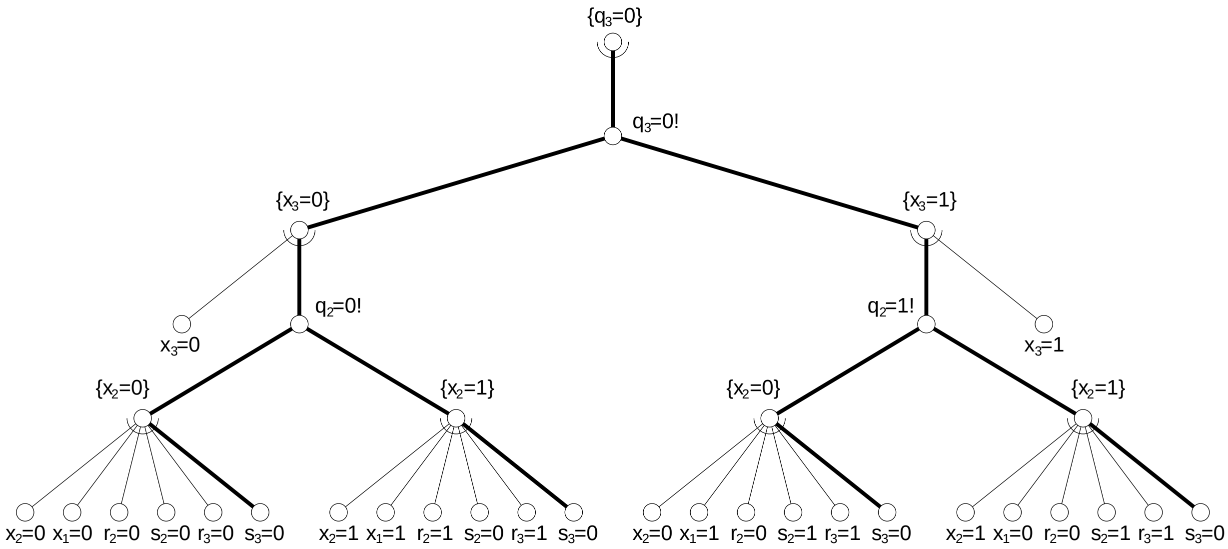


Figure 4.7: AND/OR graph representing 1-implicant of variable q_3

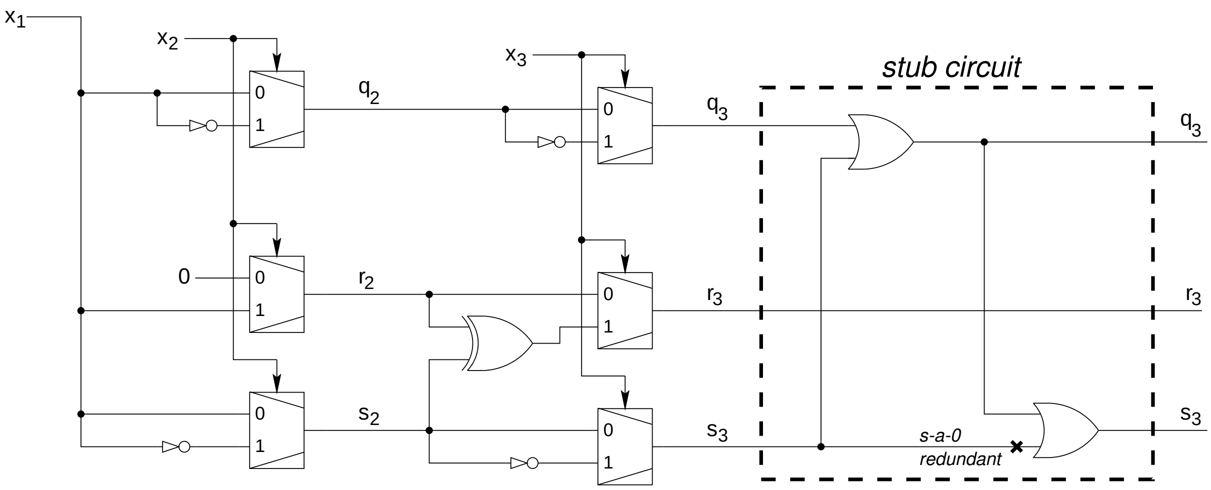


Figure 4.8: Adding implicants to create stub circuit

the resulting stub circuit which does no longer depend on input variables of earlier time frames but only on the auxiliary variables t_q, t_r, t_s.

In this example, the reachable state set reaches its fixed point at $t = 3$. The set $R(3)$ is the final set of states and the recurrence period is 1. We can detect the fixed point by proving that the same transformations that yielded the stub circuit for $t = 3$ are possible also for $t = 4$. Therefore, we represent the set of states at $t = 3$ by the stub circuit obtained by decomposition and cut of the miter array for $t = 3$. We attach a new time frame to the stub circuit as shown in Figure 4.9.

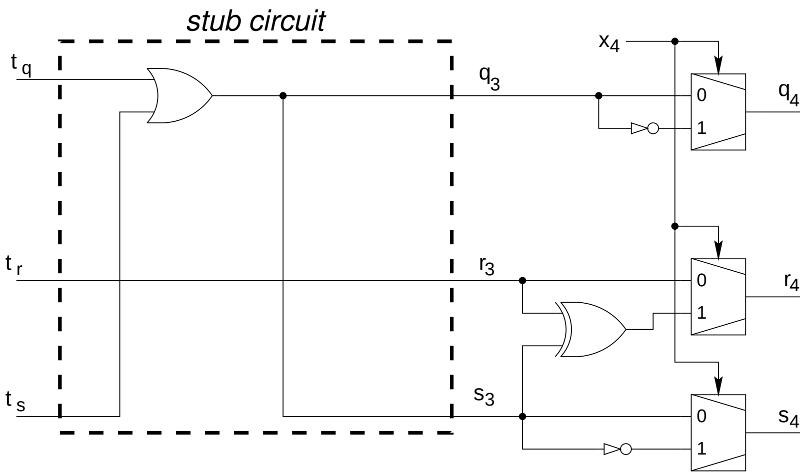


Figure 4.9: Iterative circuit array until $t = 4$ after existential quantification

If the set of reachable states $R(4)$ is the same as $R(3)$ we can determine the same implicants for the state variables as in the time frame before. Performing procedure *and_or_enumerate()* for an initial assignment $q_4 = 0$ yields the AND/OR graph of Figure 4.10. The bold lines mark a MIST which indeed corresponds to the prime 1-implicant s_4 for q_4, as was s_3 for q_3. Also for the other state variables the same implicants can be determined as one time frame before (not shown). The resulting stub circuit has an identical structure as the previous one. We can conclude that we have reached the structural fixed point.

Using this stub circuit representation of the reachable state set we can now check certain properties being subject of a given verification task. For example, consider checking the equivalence of the two circuits of Figure 4.5. As can be observed in the state transition graphs, the outputs of the FSMs are given by the state variables q and s, respectively. The equivalence of q and s is immediately obvious from the stub circuit itself.

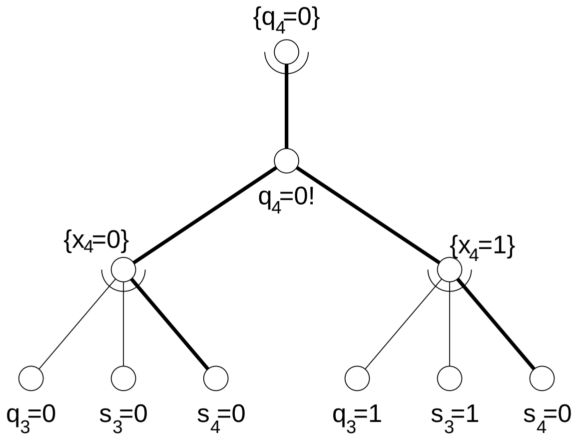


Figure 4.10: AND/OR graph for initial assignment $q_4 = 0$

Chapter 5

Equivalence Checking of Sequential Circuits

This chapter presents how structural FSM traversal as developed in the previous chapter can be applied to equivalence checking of sequential circuits. Although conventional state encoding techniques are used only rarely in today's design compilers, the state encoding of a given circuit is often modified through *structural* modifications of the design. For example, designers often improve the performance of their design by changing the pipelining at the RTL level. Similarly, at the gate level, *retiming*, i.e., the relocation of registers, becomes increasingly important in the industrial design process. However, there is great concern among designers because to date no techniques exist that can verify circuits of realistic size after such transformations. Therefore we focus on the problem of verifying circuits after synthesis with retiming and show that the structural FSM traversal presented in the previous chapter offers a promising solution to this problem.

5.1 Introduction

There are several notions of equivalence proposed in the literature. The most widely used concept is that of *reset equivalence*. It is assumed that designated initial states (or sets of initial states) for the two circuits under comparison exist together with a method (e.g., an external reset line) to bring each machine into a designated initial state. Checking design equivalence then amounts to checking the equivalence of the initial states. A similar notion proposed by Pixley is *sequential hardware equivalence* [70], which models the initialization process within the FSM descriptions of the designs. For each circuit, an input sequence (or a set of input sequences) exists which drives the circuit into a desired start state (which may be one in a set of desired start states). The output behaviour of the circuits during initialization is disregarded when checking equivalence. Only the post-synchronization behaviour is considered. There are also other notions of equivalence which are not considered in this work as, e.g., *safe replaceability* [71] and *3-valued equivalence* [42].

Checking the equivalence of the initial states of two finite state machines is a classical application of reachability analysis. As described in the introductory section of Chapter 4, the problem is typically modeled by building a product machine (depicted again for convenience in Figure 5.1). The two components A and B of the product machine have the same set of primary

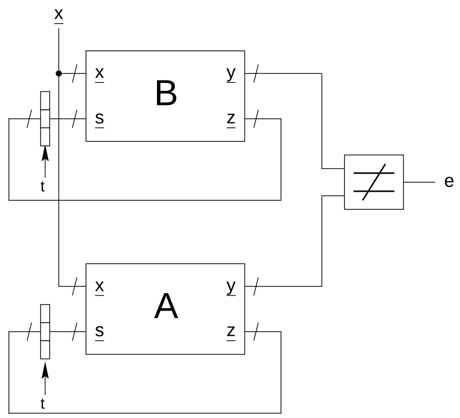


Figure 5.1: Product machine of designs A and B (sequential "miter")

inputs. Their corresponding outputs feed a comparator producing a single output e. The combined states of the FSMs A and B form the product states of the product machine. The initial state of the product machine is given by the concatenation of the individual intial states of A and B. The designs are equivalent if the product machine produces $e = 0$ for all inputs and all its reachable states. If there is a reachable state of the product machine for which $e = 1$ is possible, the sequence of input patterns driving the product machine into this state is called a *distinguishing sequence* or *(sequential) counter example* of the designs A and B. Actually, equivalence checking is typically not separated into the two steps of calculating the complete set of reachable states of the product machine and then checking equivalence for every state. Instead, the equivalence check is embedded into the breadth-first algorithm for FSM traversal [39] as shown in Table 5.1. During traversal of the product machine, all newly reached states are checked to find an input i from the input alphabet I for which the comparator output e (implementing the output function λ of the product machine) produces a 1. If such a state is found, the procedure returns immediately. It is then possible to invoke a backtrace procedure from this differentiating state to the initial state to find a distinguishing sequence.

As mentioned before, the standard techniques [26, 91, 15] represent the state sets and the transition functions or relations symbolically using BDDs. There are many techniques to improve general reachability analysis [45, 21, 75, 19, 17, 18] and to make it less vulnerable to state explosion. Several works also specifically target the problems that arise due to the special nature of verification problems. In [40] it is pointed out that BDD variables which are functionally dependent on other BDD variables are a common cause for BDD blow-up in verification. In [92] this idea is used for the verification of FSMs with similar state encodings by detecting functionally dependent state variables automatically. In [72], reencoding of the states of the FSMs under comparison is used with the goal of making the state encodings more similar.

```
check_equivalence(δ, λ, I, S0)
{
  t := 0;
  Rt := S0;   /* initial state set */
  loop
  {
    Rt+1 := Rt ∪ img(δ, Rt);
    N := Rt+1 \ Rt    /* new states found */
    foreach s ∈ N
      foreach i ∈ I
        if (λ(s,i) = 1)    /* check output e of sequential miter */
          return INEQUIVALENT;
    if (Rt+1 = Rt)    /* fixed point reached */
      break;
    t := t + 1;
  }
  return EQUIVALENT;
}
```

Table 5.1: Sequential equivalence checking using forward FSM traversal

Other approaches to sequential equivalence checking are based on automatic theorem proving, such as [49]. These approaches are attractive because of their general formulation of the problem. However, in their general form they do not have the ability to exploit domain-specific knowledge causing them to fail for larger circuit examples.

If only combinational synthesis transformations have been used on a sequential circuit, the state encoding of the transformed circuit is the same as that of the original circuit and the equivalence of both can be verified using only combinational techniques. Before it is possible to invoke the combinational verifier, however, the register correspondences between the two designs under comparison need to be known. If they are not given, a common approach for automatically deriving the state variable correspondences (or "latch mapping") is to use an inductive filtering process which gradually splits the set of storage elements into equivalence classes until a fixed point is reached [34, 41, 92, 16]. In [93], such a filtering process is combined with retiming steps to be applied to circuits which have been submitted to retiming. The method tries increasingly large values of forward retimings in order to find as many equivalent variables as possible. The equivalence of the designs is then proven by combinational BDD-based equivalence checking, avoiding a traversal of the state transition graph of the product machine.

All these methods do not exploit structural similarities between the designs other than those given by similar state encodings. Huang et al [41, 42] explored structural techniques based on sequential ATPG. Internal equivalence pairs are used by representing them as *constraints* to be used during *backward justification*. However, for their approach to be efficient it is also required that the designs under comparison contain a large number of equivalent state variables. Similar to the latch mapping approaches described above, the procedure in [41, 42] starts with a set of candidate pairs for equivalent state variables and performs a step-wise elimination of the wrong candidates. It is assumed that simple relationships exist between state variables which can be obtained by simulation. This is promising for circuits with very similar encodings but may fail in other cases. Therefore, we take a different approach based on the structural FSM traversal developed in Chapter 4. The resulting algorithm leads to a natural way of exploiting structural similarites between designs but does not rely on the equivalence of state variables.

5.2 Approximate Structural FSM Traversal

Our practical implementation [88] is an approximation of the structural FSM traversal algorithm of Section 4.4. Actually, the decomposition used to perform existential quantification for creating a stub circuit is not carried out by determining all 1-implicants or all 0-implicants of each state variable in terms of the other state variables as shown in Example 4.2 (page 93). Instead, we perform implicant-based transformations at *all* nodes in the circuit. This makes efficient use of structural similarites between the designs under comparison. By "merging" the logic between different parts of the product machine we compress important information about the reachability of states into a relatively small area near the next state variables of the last time frame.

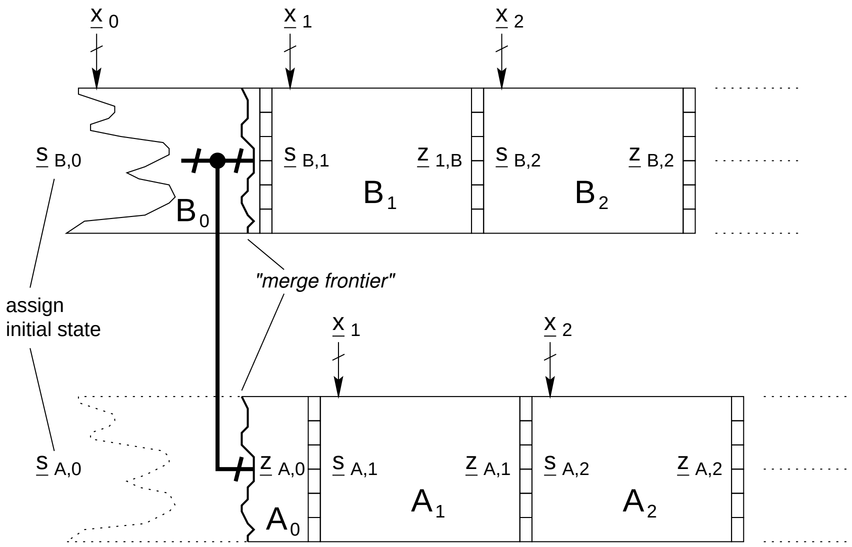


Figure 5.2: Time frame merging by sharing of logic

Consider Figure 5.2. The verification procedure is based on performing implicant-based logic transformations as developed in Chapter 3. We speak of the "merge frontier" denoting a set of gates that identifies a border line between the circuitry that is shared between the two machines A and B and the separate circuitry for each machine. This is schematically shown in Figure 5.2. In our practical method, the circuitry around the merge frontier is an approximation for the exact stub circuit as introduced in Section 3.8.3 and Section 4.4. The approximate stub circuit has as inputs the merged signals of the merge frontier and as outputs the state variables $\underline{s}_{A,t}$ and $\underline{s}_{B,t}$. It maps every vector of value assignments possible at the merge variables onto state vectors of the product machine which represent equivalent states of machines A and B.

As described in Section 4.4 we detect a fixed point by noting that the decomposition steps to create a stub circuit repeat with a certain period. Motivated by the theoretical results about the recurrence periods of $R(t)$ as developed in Section 4.3, for the time being, we assume that this period is 1. In our practical procedure, the main goal is to identify in a give time frame a set of circuit transformations between the nodes of machine A and the nodes of machine B being valid also in future time frames. The basic instrument to find such a set is an *instruction queue* Q_t that contains a set of instructions for circuit transformations at some time t. The instruction queue is processed in a *first-in-first-out* manner. Circuit transformations at time t are stored in Q_t in the order in which they have been performed. This operation is called ***record***. In the next time frame, at $t+1$, we try to make maximum use of the instructions recorded previously and for each recorded circuit transformation we check whether or not it is still valid at time $t+1$. Only if it is invalid it is removed from the instruction queue, otherwise the transformation is performed also at $t+1$. This process of reusing the stored instructions is referred to as ***play***. If the circuit transformations in the instruction queue are not sufficient to establish the equivalence of the output signals at time $t+1$ additional transformations are identified and recorded.

By *recording* and *playing*, the instruction queue is improved in each time frame. Finally, an instruction queue is created that remains valid also in later time frames and which fulfills the task of proving the equivalence of the circuit outputs. Now, it must be shown that this instruction queue is also valid in *all* future time frames. For this induction the *cutting* procedure which performs existential quantification on the representation of the reachable state set, $R(t)$, is very important. The merge frontier heuristically indicates the inputs of the stub circuit that has been created by the circuit transformations. If the stub circuit is exact, the signals of the merge frontier correspond to the variables $\underline{c_t}$ in the decomposition shown in Figure 3.42 in Section 3.8.4. Generally, though, by cutting through the circuit array at the merge frontier, we only approximate the reachable state set of the product machine. The stub circuit created mainly excludes those states that are impossible because reachable states of the individual machines cannot occur simultaneously and hence do not constitute a reachable product state. We neglect, however, that some product states are unreachable because the states of the individual designs themselves are unreachable. This may result in so-called *false negatives*, i.e., the circuits are incorrectly determined to be inequivalent. Often, this problem can be cured by keeping additional levels of logic, called *stub levels*, in front of the merge frontier. In this way, the resulting stub circuit includes more information about the reachability of states of the individual designs. This will be further discussed below.

If an appropriate cut has been found and the instruction queue can be played a sufficient number of times, the combinational structure in the circuit array repeats periodically. We say that our procedure has reached a "structural fixed point". This completes the verification process. The procedure will now be described in more detail.

Figure 5.3 shows the proposed algorithm for FSM equivalence checking that is a practical refinement of the general procedure described in Figure 4.4 on page 94. The pseudo-code for

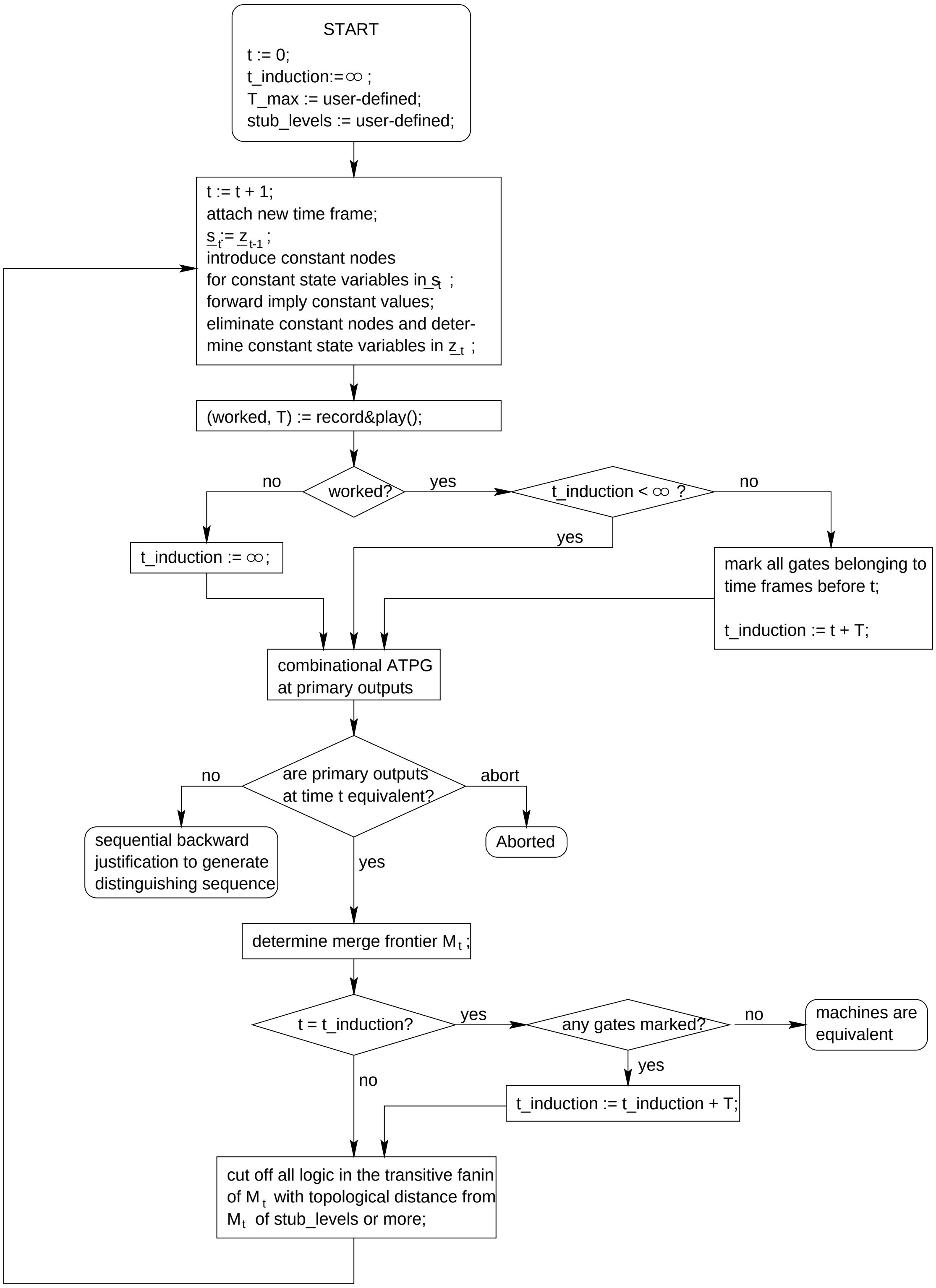


Figure 5.3: Sequential equivalence checking algorithm

```
/* Routine operates on a global data structure for the current miter array with present state
variables s and next state variables z. It has t, t_induction and T_max of Figure 5.3 as global
variables */
record_and_play()
{
  if (t_induction < ∞)     /* trying induction */
    PLAY := {Q_{t-T}};
  else    /* check old queues to find fixed point */
    PLAY := { Q_i | i ∈ {t-1, t-2, . . . , t-T_max} and s_i = s_t};
  worked := false;
  for (each Q_i ∈ PLAY)
  { worked := true;
    Q_t := ∅;
    for (each instruction α_j ∈ Q_i)
    { verify whether or not circuit transformation α_j is
        valid in current time frame;
      if (valid)
      { execute α_j (perform circuit transformation);
        Q_t := Q_t ∪ {α_j};     /* put in queue */
      }
      else worked := false;
    }
    if (worked = true)
    { T := t-i;
      break;
    }
    else reverse all transformations made for Q_t;
  }
  if (worked = false)
  { Q_t := ∅;
    for (each node in circuit array)
      { identify implicant-based circuit transformation α;
        if (α reduces literal count of circuit array)
        { perform transformation α;
          Q_t := Q_t ∪{α};    /* put in queue */
        }
      }
    }
  return (worked, T);
}
```

Table 5.2: Procedure *record_and_play()*

the algorithm *record_and_play()* is given in Table 5.2. The various steps of the algorithm are illustrated in detail by the following example.

Example 5.1 The variables *stub_levels* and *T_max* must be defined by the user as will be discussed later. At the beginning of each iteration a new time frame is attached to the current circuit array. Initially, the circuit array is empty. Whenever a new time frame is attached we assign constant values of the state variables to the corresponding nodes in the circuit array and simplify the logic accordingly. Initially, the constant values are given by the initial state. These constant values may propagate to the next state variables. Consider Figure 5.4. For both machines we are given

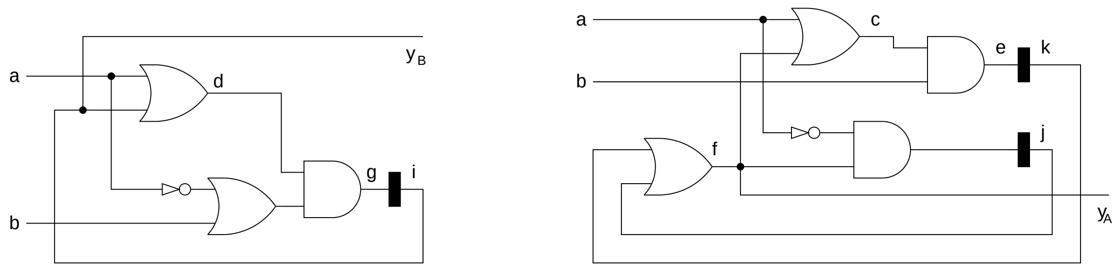


Figure 5.4: Circuit examples with initial states $S_{0,A} = S_{0,B} = \underline{0}$

an initial state of 0 for all registers. This leads to the situation shown in the left part of Figure 5.5. A constant value of 0 has propagated to the next state vector and it is

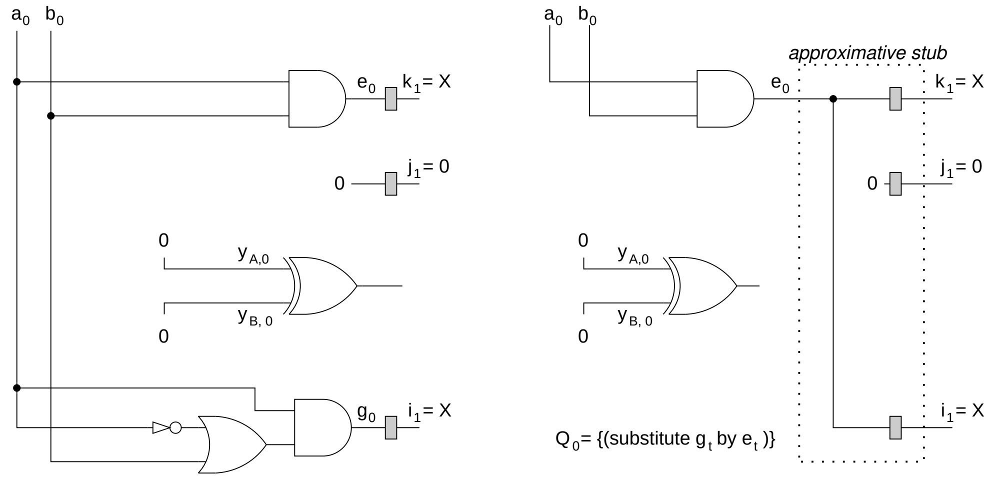


Figure 5.5: Circuit array in first iteration

$j_1 = 0$. This value will be propagated further when the next time frame is attached.

After the time frame has been attached the algorithm optimizes the logic to facilitate equivalence checking at the outputs in this and subsequent time frames. The current implementation determines only single-literal implicants (implications) to identify node substitutions similarly like in many combinational equivalence checkers. These transformations are performed in a controlled way by an instruction queue in order to detect a fixed point. For each time frame we store a set of instructions Q_t that keeps exact records of all transformations performed in that time frame. Routine *record_and_play()* has the task to select one of the previous instruction queues and to determine for the selected queue whether or not the recorded circuit transformations are still valid in the current time frame. If this is not the case another instruction queue is tried. Trying a large number of instruction queues can be time-consuming, therefore the user-defined parameter *T_max* is used to restrict the search to the last *T_max* instruction queues. If no instruction queue is found that can be played successfully new circuit transformations are identified and stored in the instruction queue. In this example, no previous instruction queues exist. The circuit is optimized as shown in the right part of Figure 5.5 and the performed transformations are stored in Q_0.

Next, it is checked whether the primary outputs in the current time frame are equivalent. If this is not the case, the circuits are not equivalent and a backward justification process like in conventional ATPG tools is invoked to calculate a distinguishing sequence.

The algorithm now determines whether previously processed portions of the circuit array can be cut off. Note that the algorithm only performs local transformations in the circuit array. For this reason it usually does not affect the performed circuit transformations if we cut off previously processed circuitry in a sufficiently large distance from the area currently processed. A heuristic procedure based on *retiming* [60] determines the "merge frontier". This is accomplished by moving the registers at the end of the last time frame backwards until they are located in fanout branches such that different branches of the same fanout stem feed registers of different machines. The corresponding fanout stems represent the nodes of the merge frontier.

Note that retiming is only used as a heuristic to identify the merge frontier. We could just as well have formulated a "tracing procedure" that traces from the state variables backward through the circuit until merge points have been identified according to some rules. There is no need for retiming in the sense of re-encoding the circuit. We only use this terminology because it concisely describes the proposed heuristic of identifying the merge frontier.

The cut through the circuit array is located in the transitive fanin of the merge frontier. This determines the approximate stub circuit. In principle, *false negatives* can occur as a result of this approximation. In practice this can often be avoided by leaving a sufficient number of logic levels in front of the merge frontier. This number of logic levels is called *stub_levels* in Figure 5.3 and is a user-defined parameter. Typical values are between 0 and 5.

Consider again the right portion of Figure 5.5. We only consider the registers that are not assigned a constant value and are still physically present in the circuit array. These are the registers at k_1 and i_1. Note that k_1 stems from machine A and i_1 from machine B. They are located in fanout branches of the same fanout stem, hence e_0 belongs to the merge frontier, which here does not have any other nodes. In the example, we assume *stub_levels* = 0. Therefore, the circuit array can be cut at signal e_0 and a new time frame is attached. The result is shown in the left portion of Figure 5.6. The newly introduced variable at the cut line is called s. The fanout

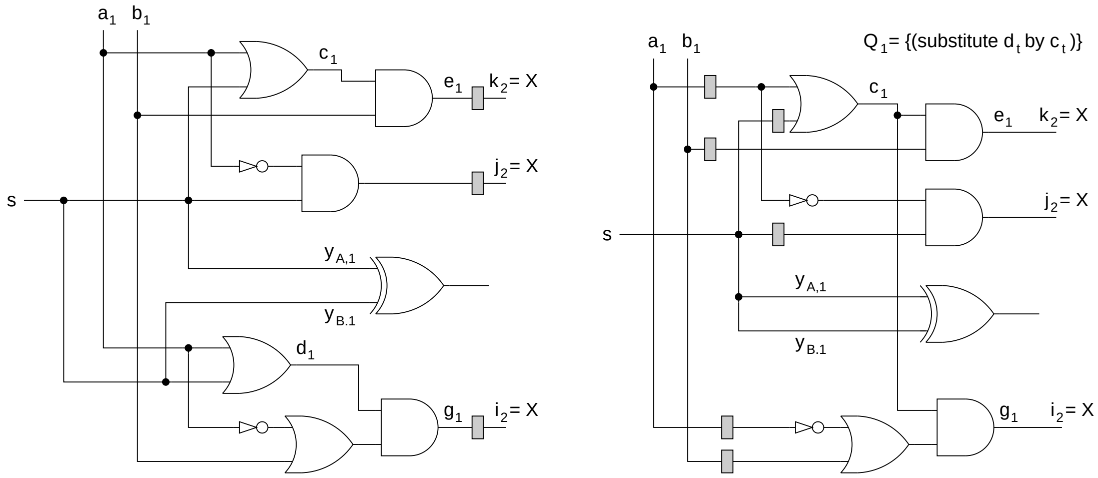


Figure 5.6: Circuit array in second iteration before and after merging

system of s and the constant signal j_1 constitute our approximation of the stub circuit for the reachable state set $R(0)$ as indicated in Figure 5.5.

After a new time frame has been appended it is always checked whether a previous instruction queue can be played. As given by the definition of the set named *PLAY* in procedure *record_and_play()* an instruction queue can only be played if it was recorded with the same constant values at the state variables as are given in the current time frame. Here, no instruction queue can be played. New transformations are recorded in Q_1 as shown in the right portion of Figure 5.6. As a result of the optimization it is trivial to determine the equivalence of the primary outputs. Next, a merge frontier is determined as shown in the right portion of Figure 5.6. Assuming *stub_levels* = 0 we cut the circuit array at the stems of these fanout systems. This does not result in any removal of logic.

A new time frame is attached as shown in Figure 5.7 and a new instruction queue must be recorded. The transformations lead to the circuit array as shown in Figure 5.8. We determine a new merge frontier suggesting a cut at signal f_2. The next iteration leads to the circuit array of Figure 5.9. Just like in the previous time frame no constant values exist at the state variables and it is determined in *record_and_play()* that the instruction queue Q_2 can be played. Actually, all recorded transformations turn out to be valid in the current time frame so that the circuit of Figure 5.10 results.

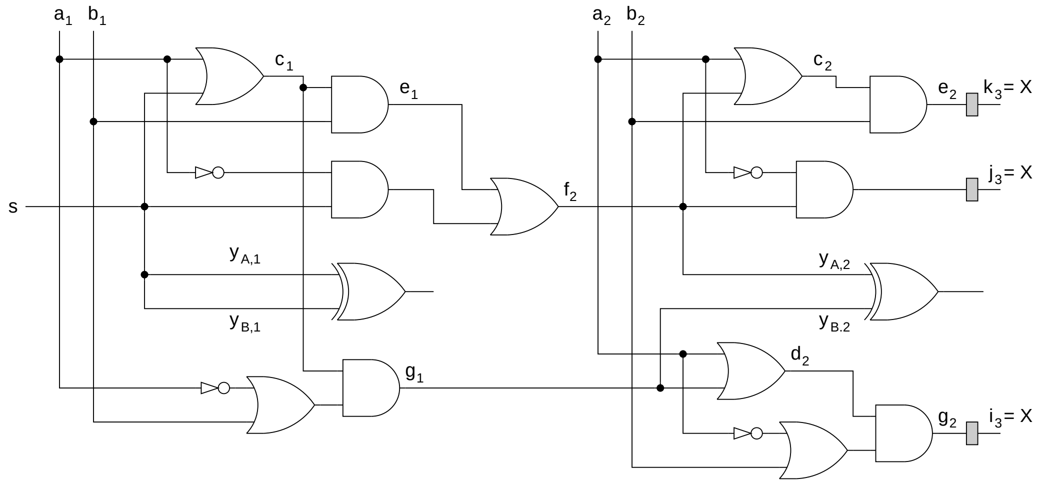


Figure 5.7: Circuit array in third iteration before merging

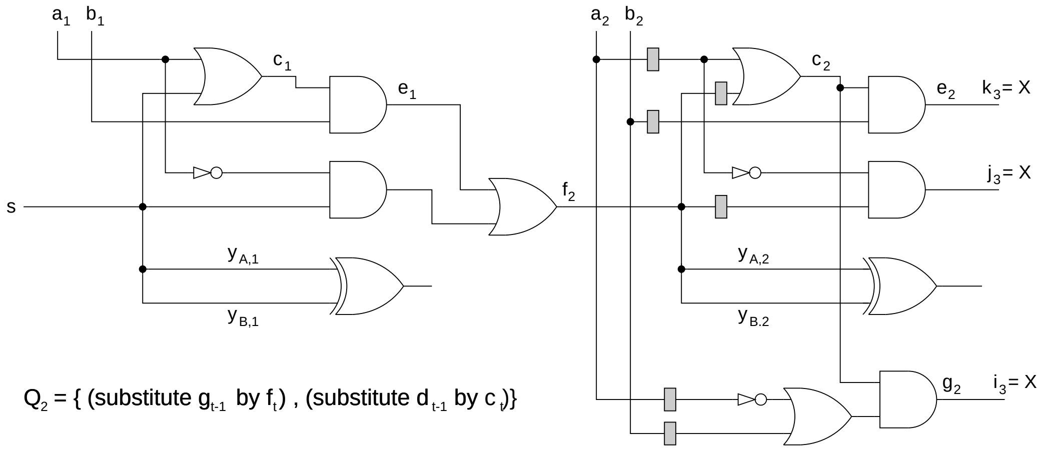


Figure 5.8: Circuit array in third iteration after merging

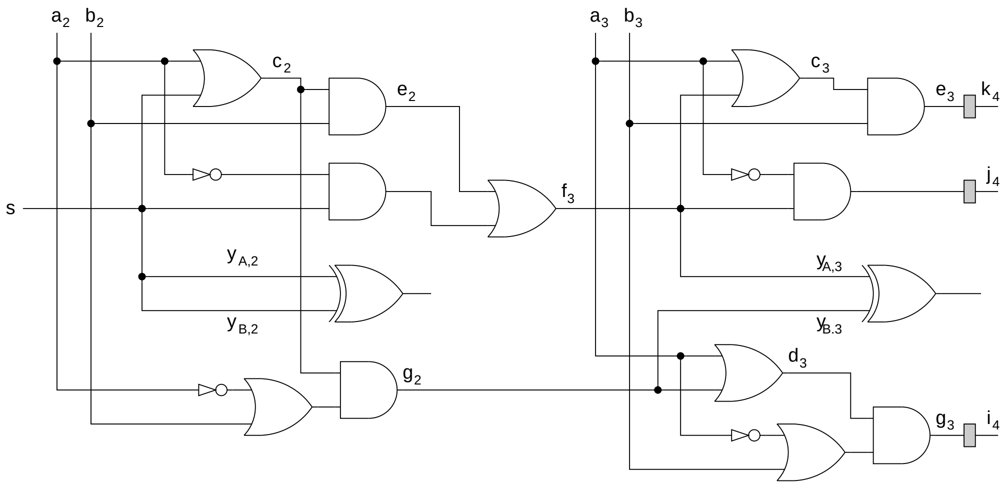


Figure 5.9: Circuit array in fourth iteration before merging

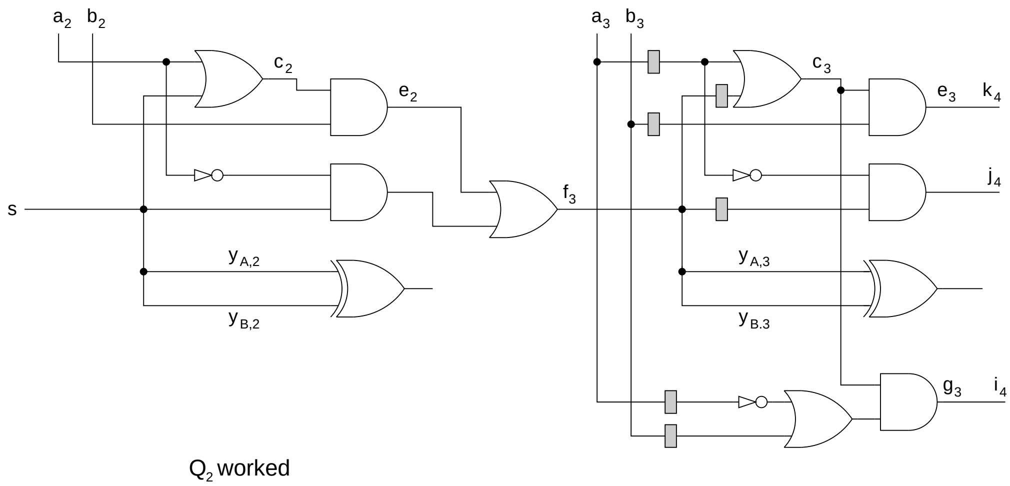


Figure 5.10: Circuit array in fourth iteration after merging

If an instruction queue has been played successfully the algorithm enters the induction mode. This is done by setting variable *t_induction* to *t* + *T* where *T* is the number of cycles since the successful instruction queue has been recorded. In most practical cases, this is the most recently recorded instruction queue so that *T* = 1. Furthermore, to ensure the correctness of the induction all gates belonging to time frames prior to the current time frame are marked. The algorithm continues the iteration and in each new time frame the instruction queue recorded at time *t* - *T* is played. This is done until all queues of a period have been played. Remember that all gates of previous time frames were marked when the induction mode was started. We continue to play the instruction queues until all marked gates have disappeared as a result of the cutting procedure. At this point, it is guaranteed that the combinational structures generated in the circuit array will repeat periodically and hence, a *structural fixed point* of the iteration is reached.

Note that this method is not restricted to a single initial state. If a set of initial states is used additional circuitry must be attached in front of the first time frame that represents the given set of initial states. In other words, a stub circuit must be created representing the set of initial states that feeds the first time frame. If an initializing sequence is given the above iteration has to be modified slightly. Instead of assigning an initial state at the state variables, the values of the initializing sequence are assigned to the primary inputs for each iteration. During the application of the initializing sequence the equivalence check at the outputs is switched off unless the designer wants to check the equivalence of the machines also during the initialization process [71].

Since the *record_and_play()* procedure only calculates an approximate solution to the decomposition problem described in Section 4.4 the resulting stub circuit represents a superset of $R(t)$, i.e., more states may be considered than are actually reachable. As mentioned earlier, this can lead to false negatives. However, since a complete symbolic state traversal is impossible for most large designs, the same information about unreachable states and local don't-cares that has been used by the synthesis tool when transforming design A into design B usually can also be compressed into the stub circuit using local transformations. Furthermore, the fact that we may consider more states than are actually reachable can have a very beneficial effect. With an approximate decomposition the fixed point can often be reached much faster than with the exact solution.

Example 5.2 As an example for the acceleration of FSM traversal due to over-approximation of the reachable state set, consider the product machine for equivalence checking of two versions A and B of an n-bit binary up-counter. The counter has an enable input named x. Figure 5.11 shows one time frame of the corresponding product finite state transition structure.

After expanding the product FST twice, assigning the initial state $(0, \ldots, 0, 0)$ to the state variables $(b_{n-1}, \ldots, b_1, b_0,\ a_{n-1}, \ldots, a_1, a_0)$ of the first time frame and merging all logic, we obtain the circuit of Figure 5.12, which is a tap circuit representing the set of reachable states $R(2)$ of the product FST. The state vector of the product

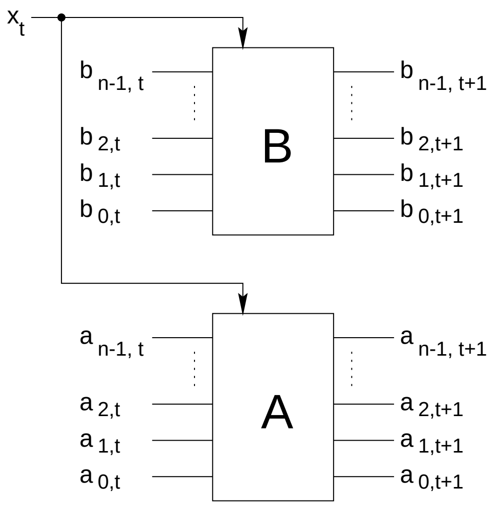


Figure 5.11: Block diagram of one time frame of product FST

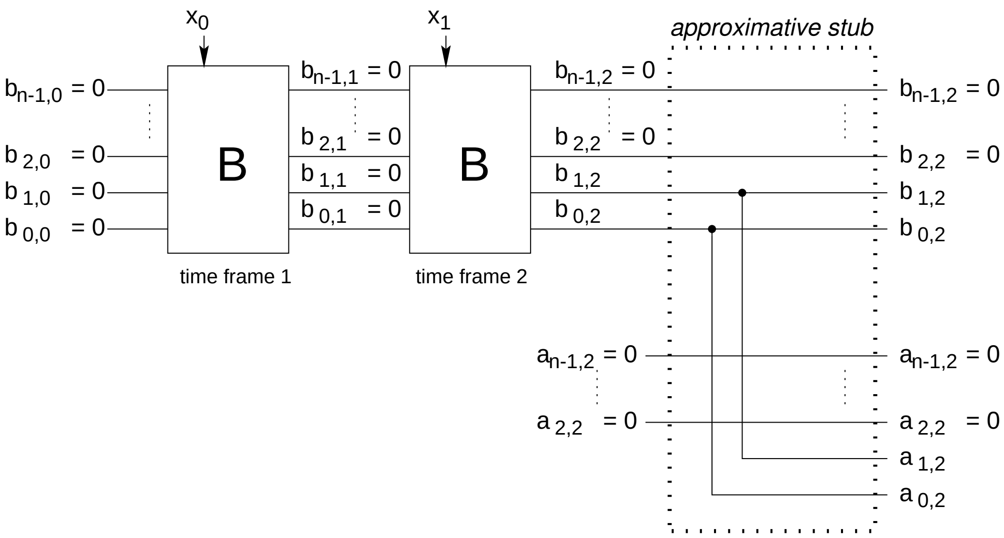


Figure 5.12: Expansion into 2 time frames and merging all logic

FST is $(b_{n-1}, \ldots, b_1, b_0,\ a_{n-1}, \ldots, a_1, a_0)$ and the set of reachable states $R(2)$ contains three elements:

$$\begin{array}{l} (0, \ldots, 0,0,0,\ \ 0, \ldots, 0,0,0) \\ (0, \ldots, 0,0,1,\ \ 0, \ldots, 0,0,1) \\ (0, \ldots, 0,1,0,\ \ 0, \ldots, 0,1,0) \end{array}$$

The merge frontier in Figure 5.12 is given by the fanout points corresponding to the signals $b_{0,\,2}$ and $b_{1,\,2}$ as well as to the constant signals $b_{n-1,\,2}, \cdots, b_{2,2}$ and the constant signals $a_{n-1,\,2}, \cdots, a_{2,2}$ which are all constant 0. This merge frontier marks the inputs of the approximate stub circuit.

After cutting at the merge frontier, the remaining stub circuit represents the following set of states:

$$\begin{array}{l} (0, \ldots, 0,0,0,\ \ 0, \ldots, 0,0,0) \\ (0, \ldots, 0,0,1,\ \ 0, \ldots, 0,0,1) \\ (0, \ldots, 0,1,0,\ \ 0, \ldots, 0,1,0) \\ (0, \ldots, 0,1,1,\ \ 0, \ldots, 0,1,1) \end{array}$$

The stub circuit represents 4 states because it has two independent input variables. The represented set of states corresponds to an expansion of the product FST into 3 (rather than 2) time frames. We have over-approximated the reachable state set $R(t)$ for $t = 2$ of the product FST. Note that this approximation does not add unreachable states to the state set of the product FST. The shown over-approximation is possible in every time frame. Therefore, for this example of an n-bit binary counter having $N = 2^n$ states, the structural FSM traversal can be completed in $\log_2 N = n$ steps. By way of contrast, conventional forward FSM traversal needs N steps, if no approximation techniques are used.

In general, if a design B was derived from a design A by logic synthesis which did not optimize with respect to the set of reachable states, the described approximation does not lead to false negatives.

In Example 5.2, it was possible to identify equivalent state variables. For this special case, methods like [41, 42] can also be useful as a preprocessing phase to our approach. If equivalent state variables can be determined then the corresponding substitutions can be added to the instruction queue for *every* time frame so that a fixed point is reached much faster. For example, in the special case where all state variables are known to be equivalent the problem is combinational and our technique would produce a “working” instruction queue after only one iteration.

5.3 Experimental Results

As pointed out earlier, state encoding techniques are used rarely in commercial synthesis tools, but the state encoding of circuits can change in practice because of structural modifications of the netlist including retiming. At the RTL level, changes in the pipelining often lead to repositioning of registers which has a similar effect as retiming on the gate-level. An important application of the proposed approach is to verify circuits after synthesis including retiming and related techniques.

A prototype of the described approach has been incorporated into the HANNIBAL package. The technique was evaluated by verifying circuits of the ISCAS89 benchmark set against the optimized and retimed circuits. The circuits were optimized by kernel extraction (using *fx* in SIS [84]). After optimization retiming is performed (using *retime* in SIS). The resulting circuits were verified against the original ones. For the original circuits we assumed an initial state of 0 for all registers.

A difficulty in our experimental evaluation comes from the fact that for larger circuits SIS cannot compute the initial state after retiming, because it uses symbolic FSM traversal for this task. Therefore, we developed a simple retiming algorithm that moves all registers as far as possible into forward direction. The new initial state can simply be calculated by forward implications of the old initial state. In this way, we ensured that the encoding for all optimized circuits differs drastically from the encoding of the unoptimized original circuits and no simple relationships exist between state variables.

Table 5.3 shows some statistics about the benchmark circuits before and after synthesis and retiming. As can be seen, in most cases the number of latches differs greatly for the original and the transformed versions of each circuit. This greatly influences the performance of a standard reachability analysis as it is implemented in UC Berkeley formal verification tool VIS [9]. The command *compute_reach* in VIS uses BDD-based FSM traversal to calculate the set of reachable states. It can be successfully applied only to the smaller benchmark circuits including S1238. For the larger circuits, the state explosion problem is encountered. The BDD of the set of states or intermediate BDDs in image computation blow up, so that the traversal has to be aborted. (For example, to the best of our knowledge, circuit S1423 has never been completely traversed to this date).

Table 5.4 shows some statistics about the state transition graph and standard FSM traversal for the original and the transformed circuits. In most cases the number of states is not altered by the synthesis and retiming steps applied to the circuits. Exceptions are S344, S349, S820, S832 and S953. Except for S820 and S832 the sequential depth of the state transition graphs is not altered, either. However, the fact that the encoding of states is changed has a drastical effect on the BDD size representing the reachable state sets. In most cases, the BDDs become significantly larger if the number of latches increases. The last column of Table 5.4 shows how many iterations were needed for *record_and_play()* to reach the fixed point when traversing the product machine composed of the original and the transformed circuits. This column clearly shows the beneficial effect which the approximation discussed in Example 5.2 (page 113) has on performance. Extreme cases are the circuits S420 and S838 which exhibit a counting behaviour. Standard FSM traversal has to iterate through all 65536 states of circuit S420 before completion. The resulting BDD consists of only the terminal node labelled ‘1’. However, together with the altered state encoding after synthesis and retiming, it was not possible to complete reachability analysis for the transformed S420 within 1 hour of CPU time on a 450 MHz AMD K6-2 machine running Linux. The same holds for S838. Without approximations, this circuit can be traversed

Circuit	# inputs	# outputs	# latches	
			orig.	transf.
s208	10	1	8	46
s298	3	6	14	42
s344	9	11	15	42
s349	9	11	15	42
s382	3	6	21	57
s386	7	7	6	78
s420	18	1	16	96
s444	3	6	21	58
s510	19	7	6	118
s526	3	6	21	75
s635	2	1	32	95
s641	35	24	19	19
s713	35	23	19	19
s820	18	19	5	185
s832	18	19	5	192
s838	34	1	32	196
s953	16	23	29	121
s1196	14	14	18	30
s1238	14	14	18	31
s1423	17	5	74	245
s5378	35	49	179	411
bs1512	29	21	57	258
bs3271	26	14	116	194
bs3330	40	73	132	270
bs3384	43	26	183	585
bs4863	49	16	104	430
bs6669	83	55	239	411

Table 5.3: Circuit statistics of original and transformed circuits

Circuit	*compute_reach* (VIS-1.3) for original circuit			*compute_reach* (VIS-1.3) for transformed circuit			*record_and_play()* (HANNIBAL) for product machine
	# states reached	BDD size	sequ. depth	# states reached	BDD size	sequ. depth	# iterations til fixed point
s208	256	1	255	256	3639	255	15
s298	218	76	18	218	751	18	10
s344	2625	991	6	2385	11203	6	9
s349	2625	995	6	2385	10674	6	9
s382	8865	152	150	8865	4449	150	16
s386	13	8	7	13	828	7	9
s420	65536	1	65535	*time-out*			27
s444	8865	224	150	8865	4592	150	16
s510	47	8	46	47	3227	46	12
s526	8868	275	150	8868	8171	150	21
s635	*time-out*			*time-out*			37
s641	1544	136	6	1544	133	6	9
s713	1544	136	6	1544	134	6	9
s820	25	10	10	16	43273	8	17
s832	25	10	10	16	47539	8	16
s838	*time-out*			*time-out*			51
s953	504	549	10	716	25607	10	11
s1196	2616	1044	2	2616	2745	2	6
s1238	2616	1405	2	2616	2542	2	6
s1423	*out of memory*			*out of memory*			14
s1512	*out of memory*			*out of memory*			16
s3271	*out of memory*			*out of memory*			19
s3330	*out of memory*			*out of memory*			9
s3384	*out of memory*			*out of memory*			17
s4863	*out of memory*			*out of memory*			8
s5378	*out of memory*			*out of memory*			36
s6669	*out of memory*			*out of memory*			11

Table 5.4: STG statistics

neither in the original nor in the transformed version. With its 34 latches, $1.7 \cdot 10^{10}$ iterations are needed for a complete traversal of the FSM. This was not possible within the chosen time limit of 1 hour. Using the approximations given by the *record_and_play()* procedure, approximate FSM traversal needs far less iterations. The number of iterations needed to traverse a counter grows linearly with the number of latches.

Table 5.5 shows the experimental results for sequential equivalence checking based on the approximate structural FSM traversal given by routine *record_and_play()*. These experiments were performed on a SUN Sparc 5 with 96 MBytes main memory. Note that in [41, 42] a different notion of equivalence is used and no results are shown for circuits that are both optimized and retimed. Therefore we compare our techniques only with the conventional verification approach by symbolic FSM traversal. The results for the command *seq_verify* in VIS-1.3 are shown in the right column of Table 5.5. The results show the feasibility and great potential of the proposed approach to verify circuits after synthesis and retiming. With our technique, the verification could be completed within acceptable CPU times for many cases where the conventional approach failed.

Circuit Name	*record_and_play()* (HANNIBAL)		*seq_verify* (VIS-1.3)
	# iterations till fixed point	CPU time h:min:sec	CPU time h:min:sec
s208	15	0:00:08	0:00:04
s298	10	0:00:09	0:00:03
s344	9	0:00:11	0:00:12
s349	9	0:00:11	0:00:12
s382	16	0:00:17	0:01:55
s386	9	0:00:48	0:00:03
s420	27	0:00:43	unable
s444	16	0:00:18	0:01:59
s510	12	0:00:35	0:00:26
s526	21	0:00:35	0:01:35
s635	37	0:01:32	unable
s641	9	0:00:12	0:00:04
s713	9	0:00:12	0:00:03
s820	17	0:36:50	unable
s832	16	0:26:37	unable
s838	51	0:08:13	unable
s953	11	0:01:09	unable
s1196	6	0:00:40	0:00:10
s1238	6	0:00:46	0:00:11
s1423	14	0:03:31	unable
s1512	16	0:04:09	unable
s3271	19	0:21:17	unable
s3330	9	0:11:33	unable
s3384	17	0:31:24	unable
s4863	8	0:36:52	unable
s5378	36	0:55:23	unable
s6669	11	0:47:15	unable

Table 5.5: Verification of optimized and retimed circuits

Chapter 6

Future Work

The techniques explored in this thesis have concentrated on two major issues: firstly, setting up a framework of reasoning techniques which can be applied in synthesis-based verification methods, and secondly, developing the foundations of a structural approach to FSM traversal which is the key problem in formal verification of sequential systems. In both areas there is a lot of room for further development. The following sections outline ideas for future research in some domains.

6.1 AND/OR Reasoning Graphs

The classical OR-type variable enumeration technique can be represented as a Shannon tree. Often, a Shannon tree can be significantly reduced in size by sharing isomorphic subtrees so that a binary decision diagram is obtained. It is interesting to investigate whether such a "BDD effect" can also occur for AND/OR trees. The following example shows how an AND/OR tree can be reduced by sharing of isomorphic subtrees.

Example 6.1 Consider the circuit in Figure 6.1 The AND/OR tree for the initial

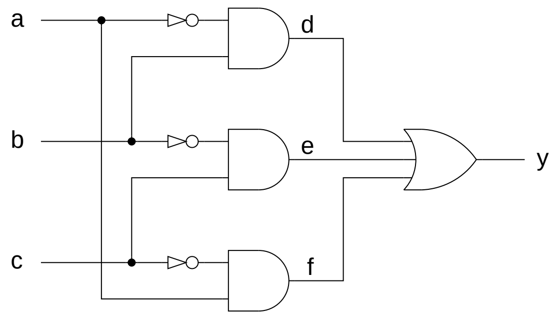


Figure 6.1: Circuit example for sharing isomorphic AND/OR subtrees

assignment $y = 0$ can be constructed using routine *and_or_enumerate()* of Table 3.2. If isomorphic subtrees are shared we obtain the AND/OR graph of Figure 6.2.

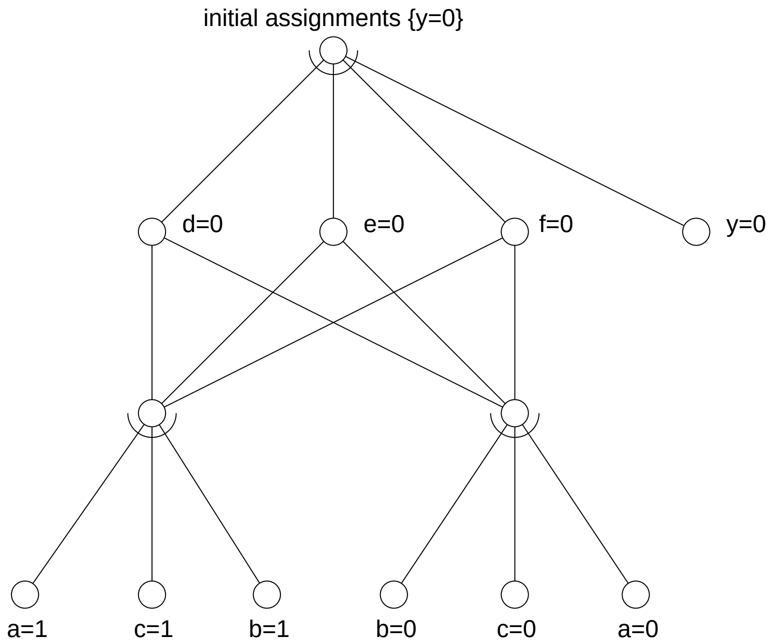


Figure 6.2: AND/OR graph for circuit in Figure 6.1

In general it is desirable to share isomorphic subtrees. So far we have only extracted information from AND/OR trees by enumerating them. If the AND/OR trees are actually built and reduced as shown in the example, a significant amount of enumeration may be saved. Similarly, just as OBDDs provide substantially more power than exhaustive simulation it can be expected that appropriate trade-offs between time and memory (using hashing and cacheing techniques) can further improve the performance of AND/OR graph-based methods.

Besides hashing of isomorphic subtrees there exist many other possibilities to improve the efficiency of the AND/OR search. Similar concepts as have been developed by [20] for branch-and-bound methods can also be used to improve the efficiency of AND/OR search. Future work may investigate this with particular focus on finding appropriate trade-offs between time and memory.

6.2 Structural FSM Traversal

We have introduced structural FSM traversal as a fixed point iteration of circuit transformations yielding a structural representation of reachable state sets of sequential circuits. A goal of future research is to apply the developed techniques to the field of model checking. A number of problems need to be addressed.

1. Modeling properties

   The properties to be examined by a model checker need to be formulated in some language.

If structural techniques are to be used, it must be explored how properties can be translated into a form which fits well with the proposed approach to FSM traversal.

For checking combinational properties, i.e., properties that are not a function of time, it is close at hand to use representations as introduced in Section 3.8. A cap circuit can be used to represent a certain relationship between signal values in the design. The cap function evaluates to 1 for every combination of signal values that fulfills a given property. A cap circuit can also be used for the formulation of certain class of properties often called *safety properties*. Safety properties correspond to conditions that must be true in every reachable state. Sequential equivalence is an instance of such a property, which is formulated for a product machine of two designs A and B to be compared.

For checking sequential properties, i.e., properties that include the notion of time, structural representations of these properties have to be developed. A possible approach related to traditional *language containment* techniques [56] is the use of finite automata as a representation of the properties to be checked.

Example 6.2 Consider a sequential circuit controlling some resource such as memory or access to a bus (Figure 6.3). Whenever the controller receives an

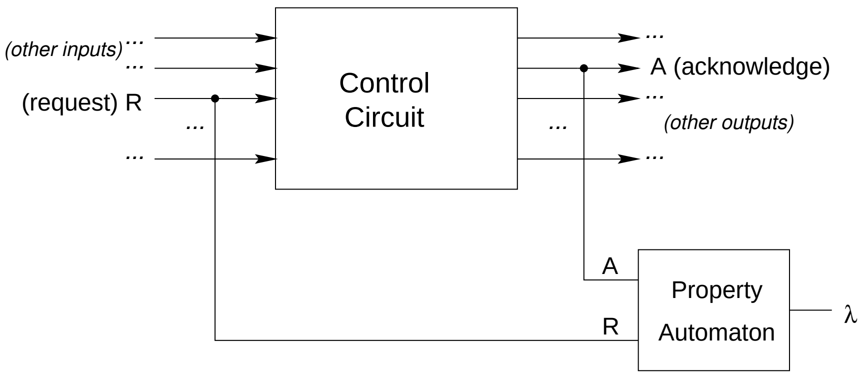


Figure 6.3: Circuit with property automaton

external request in form of a logic 1 at its input R, it issues an acknowledge signal A in the next clock cycle. The property to be checked is whether this is always the case. This property is formulated as a finite automaton as shown in Figure 6.4. The automaton outputs a logic 1 in the states indicated by double circles. State S3 is reached if the property is invalidated (rejecting state). The output value λ of the automaton in this state is 0. The automaton is implemented as a sequential circuit and attached to the controller circuit as shown in Figure 6.3. The two sequential circuits together form a product machine. Checking the property amounts to checking whether the output λ is always 1.

Future work needs to examine how to adequately formulate properties in this fashion and

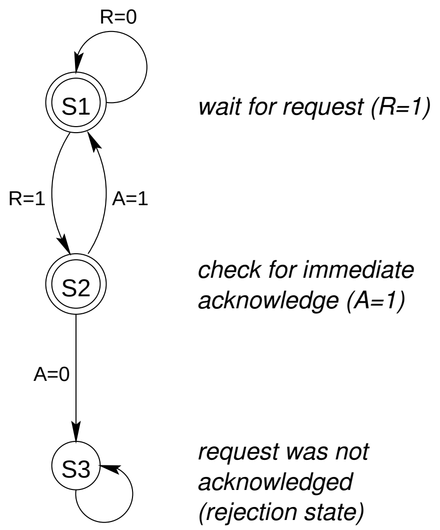


Figure 6.4: State transition graph of property automaton

how such a structural model checking methodology compares with traditional symbolic model checking as well as *bounded model checking* [6].

2. Approximations

   As can be seen in the above example, the product machine formed by combining a design under check and the properties to be analyzed are of different nature compared to a product machine for sequential equivalence checking. The latter usually contains many signal relationships between the two sub-circuits representing the designs to be compared because of structural similarities. This is not necessarily true for the former case. The approximation we have used in *record_and_play()* for sequential equivalence checking may not work for model checking. On the other hand, as can also be observed in the example, it can be expected that a property typically checks only selected aspects of a design. Besides the obvious reductions possible due to signals which are irrelevant to the considered property (like unmonitored outputs), it is necessary to explore how the product machine can be further simplified without affecting the correctness of the verification result. Just like in traditional model checking and language containment approaches, model *reduction* and *abstraction* techniques also play a central role in a structual approach to model checking.

3. *Exact* structural FSM traversal

   For product machines that cannot be further reduced, it may be necessary to conduct a reachability analysis that is *exact* because the property to be checked may be invalid for all unreachable states. For these cases, we need to develop practical techniques for synthesizing an exact stub circuit. The exact method which was described in Section 4.4 is based on calculating a 0-cover of the characteristic function $\chi_{\underline{x}}$ of the reachable state set. The prime implicants in this cover are expressed by the state variables. For determining these implicants, AND/OR reasoning needs to be performed to sufficiently deep recursion levels such that all possible implications from a state variable to the other state variables are considered. In practice, this may lead to unacceptable computation times already for mid-size circuits.

Instead of calculating the stub circuit in one step starting at the outputs of the tap circuit (the iterative circuit array), it is more efficient to begin at the primary inputs and construct the stub circuit level by level using successive cuts through the tap circuit. Each cut corresponds to an intermediate set of variables $\underline{x}$ for which an intermediate stub circuit is calculated. This methodology is similar to conventional cut-based range or don't-care set computations (see, e.g., [67, pp. 384–389]).

Example 6.3 Consider the logic network of Figure 6.5. (It is an example from [67]). Instead of calculating implicants of the output variables d and e imme-

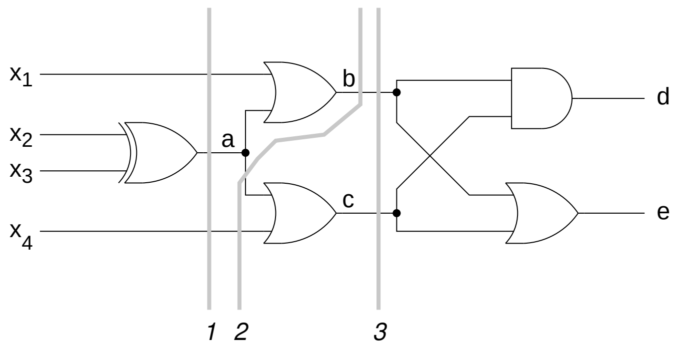


Figure 6.5: Cuts in network of Example 6.3

diately by constructing AND/OR graphs, we place successive cuts through the network as indictated by *1, 2* and *3.* For each cut, an intermediate stub circuit is calculated.

Cut 1 is given by the variables $\{x_1, a, x_4\}$. Obviously, for each variable no implicants in terms of other cut variables can be found. The stub circuit is created by cutting the XOR gate off and leaving signal a as a primary input.

Cut 2 is given by the variables $\{b, a, x_4\}$. The impossible patterns at this set of variables are (0,1,0) and (0,1,1). They can be represented by b as a 0-implicant of a. The corresponding stub circuit is shown in Figure 6.6.

The variables of cut 3 are $\{b, c\}$. Again, no implicants can be found for these variables in terms of each other. This leads to the stub circuit shown in Figure 6.7.

In each step, the circuit is transformed into a circuit with a different input-output behaviour but with the same set of output vectors. The last cut is given by the output variables d and e. Pattern (1,0) is the only impossible pattern. It can be

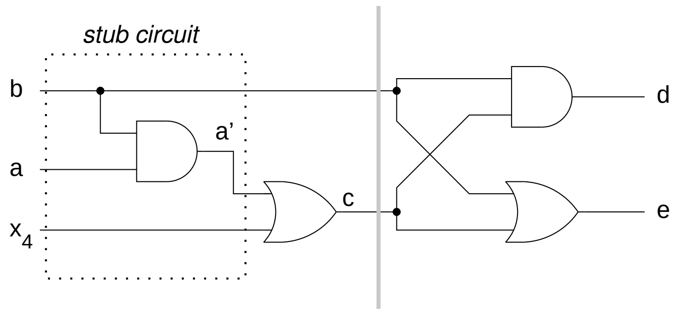


Figure 6.6: Stub circuit for cut 2

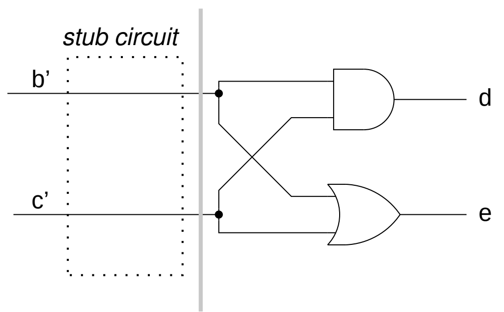


Figure 6.7: Stub circuit for cut 3

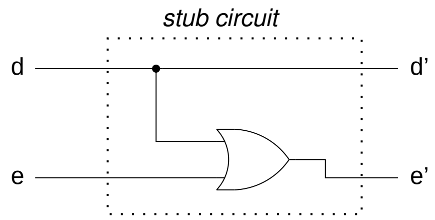


Figure 6.8: Stub circuit for network outputs

represented by d as a 1-implicant for e. The corresponding stub circuit is given in Figure 6.8.

The shown stub circuit calculation is based on a network traversal which strictly iterates the steps of propagating a cut frontier, calculating an intermediate stub for the frontier and performing existential quantification by cutting. We can view equivalence-preserving transformations (as the implicant-based transformations described in Chapter 3) and existential quantification by cutting as special instances of *range-preserving circuit transformations*. An interesting topic for research would be to develop a less strict framework of local range-preserving circuit transformations aiming at the efficiency of a non-canonical set representation. As an example, reconsider the stub circuit in Figure 4.9 of Example 4.2 (page 98). The OR gate in this stub circuit can be cut off and replaced by a primary input without altering the range of the stub circuit. A research goal for a stub synthesis framework could be to develop heuristics exploiting situations of this kind. Implicant-based transformations have to be selected such that large portions of logic can be cut off, thus minimizing the stub circuit representation.

Chapter 7

Summary

This thesis has explored how *structural techniques* can be applied to the problem of formal verification for sequential circuits. Algorithms for formal verification which operate on non-canonical gate netlist representations of digital circuits have certain advantages over the traditional techniques based on canonical representations as BDDs. They allow to exploit problem-specific knowledge because they can take into account structural properties of the designs being analyzed. This allows us to break the problem down into sub-problems which are (hopefully) easier to be solved. However, in the past, the main application of such structural techniques was in the field of combinational equivalence checking. One reason for this is that the behaviour of a sequential system does not only depend on its inputs but also on its internal states, and no concepts had been developed to-date allowing structural methods to deal with large sets of states.

An important goal of this research was therefore to develop structural, non-canonical forms of representing the reachable states of a finite state machine and to develop methods for reachability analysis based on such representations. In order to reach this goal, two steps were taken. Firstly, a framework for manipulating Boolean functions represented as gate netlists has been established. Secondly, using this framework, a structural method for FSM traversal was developed serving as the basis for an equivalence checking algorithm for sequential circuits.

The framework for manipulating Boolean functions represented as multi-level combinational networks is based on a new concept of an implicant in a multi-level network and on an AND/OR-type enumeration technique which allows us to derive such implicants. This concept extends the classical notion of an implicant in two-level circuits to the multi-level case. Using this notion, arbitrary transformations in multi-level combinational networks can be performed.

The multi-level network implicants can be determined from *AND/OR reasoning graphs*, which are associated with an AND/OR reasoning technique operating directly on the gate netlist description of a multi-level circuit. This reasoning technique has the important property that it is complete, i.e. the associated AND/OR trees contain *all* prime implicants of a Boolean function at an arbitrary node in a combinational circuit. In other words, AND/OR graphs constructed for a network function serve as a representation of this function. A great advantage over BDDs is that AND/OR graphs, besides representing the logic function, also represent some structural properties of the analyzed circuitry. This permits to develop heuristics that are specially tailored for certain applications such as logic optimization or verification.

Another advantage which is especially useful for logic optimization is the fact that the proposed AND/OR enumeration scheme is not restricted to the use of a specific logic alphabet such as $B_3 = \{0, 1, X\}$. By using Roth's D-calculus based on $B_5 = \{0, 1, X, D, \overline{D}\}$ *permissible implicants* can be determined. Transformations based on permissible implicants exploit observability don't-care conditions in logic synthesis by creating permissible functions at internal network nodes.

In order to evaluate the new structural framework for manipulating Boolean functions represented as gate netlists, several experiments with implicant-based optimization of multi-level circuits were performed. The results show that implicant-based circuit transformations lead to significantly better optimization results than traditional synthesis techniques.

Next, based on the proposed structural methods for Boolean function manipulation, techniques for representing and manipulating the set of states of a sequential circuit have been developed. The concept of a "stub circuit" was introduced which implicitly represents a set of state vectors as the range of a multi-output function given as a gate netlist. The stub circuit is the result of an existential quantification operation which is obtained by functional decomposition using implicant-based netlist transformations and a network cutting procedure.

Using this existential quantification operation, a new structural FSM traversal algorithm was formulated which performs a fixed point iteration on the set of reachable states represented by the stub circuit. The proposed approach performs a reachability analysis of the states of a sequential circuit. It operates on gate netlists and naturally allows to incorporate structural properties of a design under consideration into the reasoning. Therefore, structural FSM traversal is an interesting alternative to traditional symbolic FSM traversal, especially in those applications of formal verification, where structural properties can be exploited.

Structural FSM traversal was applied to the problem of sequential equivalence checking. Here, structural similarities between the designs to be compared can effectively reduce the complexity of the verification task. The FSM to be traversed is a special product machine called sequential miter. The special structural properties of this product machine have made it possible to formulate an approximate algorithm for structural FSM traversal, called *record_and_play()*. This algorithm uses an approximation on the reachable state set represented by the stub circuit which is very beneficial for performance. Instead of calculating the stub circuit using the exact algorithm, implicant-based transformations directly using structural design similarities are performed. These transformations, together with existential quantification implemented by the cutting procedure, lead to an over-approximation of the reachable state set. By this over-approximation, only such unreachable product states are added to the set of states represented by the stub circuit which are unreachable at the current point in time but which are nevertheless equivalent. Therefore, more product states are added to the set of reachable states sometimes leading to drastic acceleration of the traversal, i.e. the fixed point is reached in much fewer steps.

The algorithm *record_and_play()* was applied to the problem of checking the equivalence of a circuit with its optimized and retimed version. Retiming is a form of sequential circuit optimization which can radically alter the state encoding of a circuit. Traditional FSM traversal techniques often fail because the BDDs needed to represent the reachable state set and the transition relation of the product machine become too large. Experiments were conducted to evaluate the performance of *record_and_play()* on a standard set of sequential benchmark circuits. The algorithm was capable of proving the equivalence of optimized and retimed circuits with their original versions, some of which (to our knowledge) have never before been verified using traditional techniques like symbolic FSM traversal. The experimental results are very promising. Future research will therefore explore how structural FSM traversal can be applied to model checking.

Appendix A

Proofs of Theorems

Theorem 3.3 (Page 28) *Let f_i be a node in a combinational network C_i. Further, let p_i be a multi-level network implicant according to Def. 3.1, such that*

1. *the transformation of node f_i into f_{i+1} given by*
   (a) $f_{i+1} = f_i + p_i$ *if p_i is a 1-implicant of f_i*
   (b) $f_{i+1} = f_i \cdot \overline{p_i}$ *if p_i is a 0-implicant of f_i*

   followed by

2. *redundancy removal (with appropriate fault list)*

generates a combinational network C_{i+1}. For an arbitrary pair of equivalent combinational networks C and C' there exists a sequence of equivalent combinational networks $(C_1, C_2, \ldots, C_k)$ such that $C \equiv C_1$ and $C' \equiv C_k$.

Proof:

The proof is along the lines of the proof of Theorem 3.1 of [55]. Switching algebra is isomorphic to two-valued Boolean algebra. A combinational network can be mapped onto a set of interdependent Boolean formulae. Each signal in the network corresponds to a variable in a formula. Each gate corresponds to a Boolean operation (disjunction, conjunction, negation). The proof is based on the fact that for any Boolean formula one can obtain any equivalent formula by applying the laws of Boolean algebra. We show that for each of these laws there exist corresponding network transformations as described in the theorem.

Note that each of the two sub-steps in the transformation of Theorem 3.3 can be considered a legal transformation for itself. For each direction of the following laws, we show the corresponding network transformation. Since each law comes in two forms dual to each other, we show the proof only for one of them.

1. *Idempotency:* $a + a = a, \quad a \cdot a = a$
   Let $f_i = a + a$. Redundancy elimination yields $f_{i+1} = a$.

Let $f_i = a$. We choose $p_i = a$ which is a 1-implicant of f_i. Changing the cover of f_i yields $f_{i+1} = a + a$.

2. *Commutativity:* $a + b = b + a, \quad a \cdot b = b \cdot a$

   These laws are fulfilled by construction (definition) of primitive AND- and OR-gates.

3. *Associativity:* $a + (b + c) = (a + b) + c, \quad a \cdot (b \cdot c) = (a \cdot b) \cdot c$

   Let $f_i = a \cdot (b \cdot c)$. We choose $p_i = \overline{c}$, which is a 0-implicant of f_i. Changing the cover of f_i yields $f_{i+1} = a \cdot (b \cdot c) \cdot c$. Redundancy elimination yields $f_{i+1} = (a \cdot b) \cdot c$.

   Proof is analogous for opposite direction.

4. *Absorption:* $a \cdot (a + b) = a, \quad a + (a \cdot b) = a$

   Let $f_i = a \cdot (a + b)$. Redundancy elimination yields $f_{i+1} = a$.

   Let $f_i = a$. We choose $p_i = \overline{a + b}$ which is a 0-implicant of f_i. Changing the cover of f_i yields $f_{i+1} = a \cdot (a + b)$.

5. *Distributivity:* $a \cdot (b + c) = a \cdot b + a \cdot c, \quad a + (b \cdot c) = (a + b) \cdot (a + c)$.

   Let $f_i = a \cdot (b + c)$. We choose $p_i = a \cdot b$ which is a 1-implicant of f_i. Changing the cover of f_i yields $f_{i+1} = a \cdot (b + c) + a \cdot b$. Redundancy elimination yields $a \cdot b + a \cdot c$.

   Let $f_i = a \cdot b + a \cdot c$. We choose $p_i = \overline{a}$ which is a 0-implicant of f_i. Changing the cover of f_i yields $f_{i+1} = (a \cdot b + a \cdot c) \cdot a$. Redundancy elimination yields $(b + c) \cdot a$.

6. *Universal bounds:* $0 + a = a, \quad 0 \cdot a = 0, \quad 1 + a = 1, \quad 1 \cdot a = a$

   Let $f_i = 0 + a$. Redundancy elimination yields $f_{i+1} = a$.

   Let $f_i = a$. We choose $p_i = 0$ which is a 1-implicant of f_i. Changing the cover of f_i yields $f_{i+1} = 0 + a$.

   Let $f_i = 0 \cdot a$. Redundancy elimination yields $f_{i+1} = 0$.

   Let $f_i = 0$. We choose $p_i = \overline{a}$ which is a 0-implicant of f_i. Changing the cover of f_i yields $f_{i+1} = 0 \cdot a$.

7. *Unary operation:* $a \cdot \overline{a} = 0, \quad a + \overline{a} = 1$

   Let $f_i = a \cdot \overline{a}$. Redundancy elimination yields $f_{i+1} = 0$.

   Let $f_i = 0$. We choose $p_i = a \cdot \overline{a}$ which is a 1-implicant of f_i. Changing the cover of f_i yields $f_{i+1} = 0 + a \cdot \overline{a}$. Redundancy elimination with carefully selected fault list yields $f_{i+1} = a \cdot \overline{a}$.

In order to complete the proof it must be shown that the two-step methodology given in the theorem also allows arbitrary sharing of logic. This follows easily from the following construction. Let C be the original network and C' be the target network. Further, let C_{tree} denote a network that has tree structure and results from C if all sharing of logic is removed by duplication. Similarly, let C'_{tree} denote the tree version of the target network. Consider the following construction: Remove all sharing of logic between the different output cones of the original network so that we obtain C_{tree}. It is easy to derive C_{tree} using the given transformation scheme. Let y be some internal fanout point and assume it is the output of an AND gate with input signals a and b. By choosing an implicant $p = ab$, adding this implicant and performing redundancy elimination with an appropriate fault list, the fanout point is moved to the inputs of the AND gate. For other gate types the procedure is analogous. This process is repeated until no more internal fanout points exist and C_{tree} has been obtained. After all sharing of logic has been removed, each output cone is isomorphic to a Boolean expression that can be manipulated arbitrarily as shown using the above axioms. Therefore, it is also possible to obtain the network C'_{tree} using the given transformation. The target network C' results if the duplicated logic is removed. This can be accomplished if equivalent nodes are substituted. If node y is to be substituted by y' this can be accomplished by selecting $p = y'$ and performing the given transformation. This process can be repeated for well-selected nodes in C'_{tree} until network C' is reached. □

Theorem 3.7 (Page 43) *Let y be the output signal of a two-level combinational circuit in SOP form. The AND/OR tree for the assignment $y = 0$ (tautology test) has only two levels if the SOP expression is unate.*

Proof:

After the value assignment $y = 0$, the output signals of the implicants in the SOP become unjustified lines. The justifications for the unjustified lines reach the primary inputs. The implications from the justifications may cause events at other implicants (unjustified lines). However, since the SOP is unate, the implications from the justifications will produce values that justify these unjustified lines so that no new AND nodes can be created. □

Theorem 3.8 (Page 46) *Let y be an arbitrary node in a combinational network and T be the AND/OR enumeration tree for an initial set of value assignments $S = \{y = 0\}$. Consider a product term $p = l_1 \cdot l_2 \cdot \ldots \cdot l_k$ where l_i is a literal corresponding to a variable f_i or its complement in the combinational network. Further, consider an IST of T with a set of leaves, L.*

If there is a one-to-one mapping between the literals l_i of p and the elements $(f_i = V_i)$ of L such that $V_i = 0$ if l_i represents the uncomplemented variable f_i and $V_i = 1$ if l_i represents the complemented variable $\overline{f_i}$, then p is a 1-implicant of y. Analogously, p is a 0-implicant of y if the IST is a subtree of the enumeration tree with the initial assignment $S = y = 1$.

Proof:

The theorem is "obvious" due to the structure of the AND/OR tree. Nevertheless, for reasons of completeness it is proved formally by noting that the AND/OR tree is isomorphic to a Boolean expression of the recursive form described below.

Let n be the *level* index associated with the nodes of the AND/OR tree and o and a be Boolean expressions associated with the OR nodes and AND nodes such that the $o_{n,j}$ are the children of the $a_{n,i}$ and the $a_{n+1,k}$ are the children of the $o_{n,j}$:

$$a_{n,i} = \prod_j o_{n,j} \tag{A.1}$$

$$o_{n,j} = \begin{cases} \sum_k a_{n+1,k} & \text{for non-terminal nodes} \\ \overline{f} & \text{for terminal nodes if leaf corresponds to } f = 0 \\ f & \text{for terminal nodes if leaf corresponds to } f = 1 \end{cases} \tag{A.2}$$

By recursively applying Equations A.1 and A.2 to all levels n we obtain a Boolean expression $a_{0,1}$ with

$$y = 0 \longrightarrow a_{0,1} = 1 \tag{A.3}$$

The proof is by *induction*:

Consider an implication subtree (IST) of the AND/OR enumeration tree for the initial value assignment $S = \{y = 0\}$.

1. Let $n = f$ be the level index of the leaves of the IST. Take the leaves of the IST, $o_{f,j}$. For all leaves we set $o_{f,j} = 0$. This means that the product term t formed by the leaves of the IST as given in the theorem evaluates to 1. Then, since in the IST there exists at least one OR child of each AND node, for every $a_{f,i}$ there exists a $o_{f,j}$ such that $o_{f,j} = 0$.
2. Assumption:
   for every $a_{n,i}$ there exists a $o_{n,j}$ child such that $o_{n,j} = 0$.
3. Then, for all $a_{n-1,i}$ it is $a_{n-1,i} = 0$.
   Proof:
   given the above assumption,
   from Eq. A.1: for every $a_{n,i}$ in the IST it is $a_{n,i} = 0$,
   for a given $a_{n,i}$ all its siblings are also included and are 0, and
   with Eq. A.2: for every $o_{n-1,j}$ in the IST it is $o_{n-1,j} = 0$,
   from Eq. A.1: for every $a_{n-1,i}$ in the IST it is $a_{n-1,i} = 0$.

By induction we conclude that the Boolean expression $a_{0,1}$ belonging to the IST becomes 0 if the product term t formed as given in the theorem evaluates to 1, i.e.,

$$t = 1 \longrightarrow a_{0,1} = 0 \tag{A.4}$$

By contraposition of Eq. A.3 we conclude with Eq. A.4 that $t = 1 \longrightarrow y = 1$, hence t is a 1-implicant of y.

The proof for a 0-implicant is analogous. □

Theorem 3.9 (Page 47) *Let y be an arbitrary node in a combinational network and T be the AND/OR reasoning tree for an initial set of value assignments $S = \{y = V\}$, $V \in \{0, 1\}$. For every* prime *implicant of y there exists a minimal implication subtree (MIST) of T such that the leaves of the MIST correspond to the literals of the prime implicant as given in Theorem 3.8.*

Proof:

Recursive learning [53], which monitors the search performed by the algorithm *and_or_enumerate()*, has been proved to be complete: it can determine all logic implications including the indirect implications of $y = 0$. Viewed in an AND/OR tree, indirect implications correspond to an IST as defined in Definition 3.11 such that the IST has leaves labelled with identical value assignments. Since recursive learning is complete, *every single-literal implicant* can be associated with an IST of the AND/OR tree for the initial set of value assignments S.

The theorem is now proved for a prime 1-implicant using the following construction. Suppose there is a product term $p = l_1 \cdot l_2 \cdot \ldots \cdot l_m$ where the literals l_i correspond to variables in the combinational network in either complemented or uncomplemented form. Further, the combinational network is modified as follows. We add an AND gate with the output signal f_p as a new primary output of the combinational network. The inputs of the AND gate are the variables of the combinational network corresponding to the literals in p. Inverters are added for those variables whose complements correspond to the literal. In other words, we implement the product term p as an additional output of the combinational network. From the correctness and completeness of recursive learning and the definition of an implicant it follows that the assignment $y = 0$ implies $f_p = 0$ if and only if p is a 1-implicant of y. (By contraposition: $f_p = 1 \longrightarrow y = 1$). Hence, p is a 1-implicant if and only if for the above construction there exists an IST with leaves each labelled with the identical value assignment $f_p = 0$. To prove the theorem, we show that in absence of this construction an IST with leaves corresponding to the literals in p exists if in presence of the construction an IST exists with identical leaves labelled $f_p = 0$.

Note that in presence of the construction, the leaves of the IST must have siblings in the original AND/OR enumeration tree that correspond to the variables of p. This is guaranteed because $f_p = 0$ can only be implied from one of the inputs of the AND gate. Hence, each leaf $f_p = 0$ of the IST must have a sibling in the original tree corresponding to a literal (variable) in the implicant. Therefore, we can identify an IST of the original tree which only contains literals of the implicant as leaves. Since p is a prime implicant, *all* literals of p must be contained int the IST as leaves. The proof of the theorem is completed by observing that for any IST we can obtain a MIST as a subtree of the IST. The MIST is also an IST and hence its leaves must correspond to an implicant. Since the considered implicant is prime the MIST obtained from the IST must still correspond to the same implicant p. □

Theorem 3.10 (Page 50) *Let y be an arbitrary node in a combinational network and T be the D-AND/OR enumeration tree for the fault y stuck-at-1. Consider a product term $p = l_1 \cdot l_2 \cdot \ldots \cdot l_k$ where l_i is a literal corresponding to a variable f_i or its complement in the combinational network. Further, consider an IST of T with a set of leaves, L, such that in the combinational network the nodes f_i* cannot be reached by the fault effect.

If there is a one-to-one mapping between the literals l_i of p and the elements $f_i = V_i$ such that $V_i = 0$ if l_i represents the uncomplemented variable f_i, and $V_i = 1$ if l_i represents the complemented variable $\overline{f_i}$, the product term p is a permissible 1-implicant of y. Analogously, p is a permissible 0-implicant of y if the IST is a subtree of the enumeration tree for the fault y stuck-at-0.

Proof:

The proof is analogous to that of Theorem 3.8. Equations A.1, A.2 and A.4 are still valid, because the terminal nodes $o_{f,i}$ are solely composed of variables which cannot be reached by the fault effect and thus can only take values $V \in \{0, 1\}$. The Boolean expression of Eq. A.3 needs to be extended towards

$$(y = 0) \wedge (y \text{ observable at a primary output}) \longrightarrow a_{0,1} = 1$$

Its contraposition states that if the Boolean expression evaluates to $a_{0,1} = 0$, y is not observable at any primary output or y must be 1. We can therefore conclude that if p evaluates to 1, y is 1 or not observable. Hence, p is a permissible 1-implicant of y. The proof for permissible 0-implicants is analogous. □

Theorem 3.11 (Page 50) *Let y be an arbitrary node in a combinational network and T be the D-AND/OR enumeration tree for the fault y stuck-at-V, $V \in 0, 1$. For* every permissible prime implicant *at a node y there exists a minimal implication subtree (MIST) of T such that the leaves of the MIST correspond to the literals of the prime implicant as given in Theorem 3.10.*

Proof:

The following is along the lines of the proof for Theorem 3.11. The two algorithms *make_all_implications()* and *complete_unique_sensitization()* being part of the recursive learning procedure [53] can identify all necessary assignments for single stuck-at fault detection at a node y in a combinational network. These correspond to permissible single-literal implicants of y. This is accomplished by recursively checking whether all consistent justifications and sensitizations contain the same implied value assignments. Viewed in the AND/OR tree these indirect implications of fault injection correspond to an IST as defined in Definition 3.11 such that the IST has leaves labelled with identical value assignments. Since recursive learning is complete, *every permissible single-literal implicant* can be associated with an IST of the D-AND/OR enumeration tree for the initial set of value assignments S.

The theorem is now proved for a permissible prime 1-implicant using the following construction. Suppose there is a product term $p = l_1 \cdot l_2 \cdot \ldots \cdot l_m$ where the literals l_i correspond to variabels in the combinational network in either complemented or uncomplemented form. Further, the combinational network is modified as follows. We add an AND gate with the output signal f_p as a new primary output of the combinational network. The inputs of the AND gate are the variables of the combinational network corresponding to the literals in p. Inverters are added for those variables whose complements correspond to the literal. In other words, we implement the product term p as an additional output of the combinational network. From the correctness and completeness of recursive learning and the definition of an implicant it follows that the assignment $y = 0$ and the requirement that y be observable implies $f_p = 0$ if and only if p is a permissible 1-implicant of y. (By contraposition: $(f_p = 1) \longrightarrow (y = 1 \wedge y$ not observable)). Hence, p is a permissible 1-implicant if and only if for the above construction there exists an IST with leaves each labelled with the identical value assignment $f_p = 0$. It remains to be shown that in absence of this construction an IST with leaves corresponding to the literals in p exists if in presence of the construction an IST exists with identical leaves labelled $f_p = 0$. This can be proved in the same way as in the proof of Theorem 3.9. □

Theorem 3.12 (Page 66) *Let f be a function of a node in a combinational network and $p_1 \ldots p_n$ a set of product terms with literals belonging to complemented or uncomplemented variables of the network. Let d be the disjunction of the product terms: $d = p_1 + p_2 + \ldots + p_n$. Then, f is functionally equivalent to d if, and only if, the p_i are 1-implicants of f and $\overline{d} = \overline{p_1}\,\overline{p_2} \ldots \overline{p_n}$ is a 0-implicant of f. In this case, d is called a* 1-cover *of f.*

Proof:

"$\longrightarrow$":

f and d are functionally equivalent iff $(d = 0) \longrightarrow (f = 0)$ and $(d = 1) \longrightarrow (f = 1)$.

1. Proof for $(d = 0) \longrightarrow (f = 0)$:
   This is true because $\overline{d}$ is a 0-implicant of f.
2. Proof for $(d = 1) \longrightarrow (f = 1)$:
   If $d = 1$ then at least one of the p_i in the disjunction must be 1. The p_i are 1-implicants and therefore, $f = 1$ also.

"$\longleftarrow$":
If f is functionally equivalent to d, then it follows directly from $d = p_1 + p_2 + \ldots + p_n$ that the p_i are 1-implicants of f and that $\overline{d}$ is a 0-implicant of f. □

Lemma 3.14 (Page 73) *Consider a function $\tau_{\underline{x}}$ with n primary outputs x_i and a range set A for the output vectors, a corresponding function $\chi_{\underline{x}}$ of n primary inputs x_i which represents the characteristic function of the set A, and a product term p of k literals corresponding to complemented or uncomplemented variables x_i. Every such product term which is a prime 0-implicant of $\chi_{\underline{x}}$ corresponds to one prime 0- or 1-implicant of each of the k outputs x_j of $\tau_{\underline{x}}$ appearing in p. Analogously, every product term which is a 0- or 1-implicant of an output x_j of $\tau_{\underline{x}}$ corresponds to a 0-implicant of $\chi_{\underline{x}}$.*

Proof:

Let $p = l_1 \cdot l_2 \cdot \ldots \cdot l_k$ be a product term consisting of k literals. Each literal corresponds to a complemented or uncomplemented variable x_i which is a primary output of $\tau_{\underline{x}}$ and a primary input of $\chi_{\underline{x}}$. If p is a 0-implicant of the characteristic function $\chi_{\underline{x}}$ then there exists no combination of value assignments in the combined circuitry of Figure 3.35 such that $p = 1$, because no vector yielding $p = 1$ can be produced by $\tau_{\underline{x}}$. Consider now a product term q that is obtained by removing any single one of the literals, l_j, from p, such that $p = q \cdot l_j$. Because p is a prime implicant of $\chi_{\underline{x}}$, q is not an implicant of $\chi_{\underline{x}}$. Any combination of value assignments that makes $q = 1$ yields $l_j = 0$, because otherwise p would be 1 which is impossible. This means that q is a 0-implicant of literal l_j. If l_j represents an uncomplemented variable x_j, then q is a 0-implicant of x_j. If l_j represents a complemented variabe x_j, then q is a 1-implicant of x_j. This holds for all k literals of p.

The reverse relationship given in the lemma is shown analogously. □

Lemma 4.1 (Page 89) *Consider an arbitrary state s lying on a cycle of the state transition graph of length q. Furthermore, let s have a recurrence period p. Then, after a finite number of time steps, state s also has a recurrence period p_g which is the greatest common divisor of p and q.*

Proof:

Let t_0 be the first time that s is in the reachable state set. Then, $s \in R(t_0 + m \cdot p + n \cdot q)$, for all integers $m \geq 0$ because of given periodicity p, and for all integers $n \geq 0$

because of cycle length q. Let T be the set of integers $k,\ k = t - t_0 = m \cdot p + n \cdot q$. Set T is (obviously) closed under addition, i.e., for all $k_1, k_2 \in T$ it is $(k_1 + k_2) \in T$. For $T \setminus \{0\}$ the following number-theoretic result holds (see e.g. [47], Theorem 1.4.1): *A set of positive integers that is closed under addition contains all but a finite number of multiples of its greatest common divisor.* Since T contains only multiples of p and q, the greatest common divisor of all positive integers in T is the greatest common divisor, p_g, of p and q. Because there is only a limited number of multiples of p_g which are not in T there is a $t_{trans} \in T$ such that $t = k \cdot p_g$ for all times $t \geq t_{trans}$. □

Lemma 4.2 (Page 89) *After a finite transition time, the smallest recurrence period, p_{SCC}, is the same for all states in an SCC. This period p_{SCC} is given by the greatest common divisor of all cycle lengths in the SCC and of all recurrence periods for states in the SCC that have been inherited from predecessor states outside the SCC.*

Proof:

All states in an SCC are reachable from all other states in the SCC. Hence, every state inherits all recurrence periods that develop according to Lemma 4.1, including a smallest one which is the greatest common divisor of all of them. □

Lemma 4.4 (Page 90) *If a finite state machine has a synchronizing sequence, then the fixed point recurrence period of all its states is 1.*

Proof:

Let l be the length of the synchronizing sequence. If the FSM is synchronizable, there is a walk of length l from each state s to a given state s_0. In particular, there is a closed walk $(s_0, s_1, ..., s_{l-1}, s_0)$ of length l. If $l = 1$ the greatest common divisor of all cycles in the SCC is 1. If $l > 1$ there is also a cycle of length $l + 1$: Consider state s_1 reachable from s_0 in one step. Since there exists a synchronizing sequence of length l there is a walk from s_1 to s_0 of length l. Appending this walk with one step from s_0 to $_1$ closes the cycle of length $l + 1$. The greatest common divisor of l and $l + 1$ is 1. Hence, by Lemma 4.2 the fixed point recurrence period of all states is 1, also. □

Lemma 4.5 (Page 90) *If a finite state machine has a synchronizing sequence of length l and if d is the sequential depth of the machine, then it takes at most $l + d$ time steps until all states of the machine are in $R(t)$ and have a recurrence period of 1.*

Proof:

The synchronization state s_0 is reachable in l time steps from every state of the FSM, including itself. Hence, state s_0 is an element of $R(t)$ for every $t \geq l$, i.e., from this time on it has a recurrence period of 1. Every other state in the STG is reachable from s_0 in at most d time steps. Therefore, it takes at most d time steps until all states have inherited the recurrence period 1 from s_0. □

Theorem 4.6 (Page 92) *Let P be the set of all recurrent states of a finite state machine. There always exists a cover $\{C_0, C_1, ..., C_{T-1}\}$ of P with $C_k \in 2^P$, and there is a time $0 \leq t_{\text{fix}} \leq \infty$, such that*

$$R(t_{\text{fix}} + n \cdot T + k) = C_k, \quad 0 \leq n, \quad 0 \leq k < T$$

T is equal to the least common multiple of the recurrence periods of the entry SCCs *of the FSM. T is called* fixed point oscillation period.

Proof:

Let p be the recurrence period of an entry SCC E. Let S be the set of states of the entry SCC and all succeeding SCCs. All states in S inherit p, so the recurrence period r of any state in S must be a proper divisor of p. According to Lemma 4.3, the set of ESCC states, E, can be partitioned into p disjoint subsets. Let $K_i \in S$ be the union of such a subset with states from succeeding SCCs such that K_i contains all states of S occurring simultaneously in the reachable state set $R(t)$. Obviously, states with a period $r < p$ will be in $m = p/r$ distinct subsets K_i. This means that the K_i represent a cover of S: $S = \bigcup_i K_i$. The subsets K_i of the cover describe all possible combinations of states in S which can be in $R(t)$ simultaneously.

Without loss of generality, we consider an STG with two entry SCCs E_1 and E_2, having recurrence periods p_1 and p_2, respectively. Let S_1 be the set of states in E_1 and all its successor SCCs. Let S_2 be the set of states in E_2 and all its successor SCCs.

There exists a cover of S_1: $S_1 = \bigcup_i K_{1,i}$ consisting of p_1 subsets of S_1, and a cover of S_2: $S_2 = \bigcup_j K_{2,j}$ consisting of p_2 subsets of S_2. The subsets $K_{1,i}$ of the cover describe all possible combinations of states in S_1 which occur in $R(t)$ simultaneously. The subsets $K_{2,j}$ of the cover describe all possible combinations of states in S_2 which occur simultaneously in $R(t)$. Let us pick one set $K_{1,i}$ and one set $K_{2,j}$ such that both sets occur simultaneously in $R(t)$. $K_{1,i}$ will reoccur after p_1 time steps, $K_{2,j}$ will reoccur after p_2 time steps. The number of time steps it takes until both reoccur simultaneously is equal to T, the least common multiple of p_1 and p_2. Therefore, there are T different combinations $C_k = K_{1,i} \cup K_{2,j}$, $0 \leq k < T$ in which subsets of the cover of S_1 and subsets of the cover of S_2 occur simultaneously in $R(t)$. The states in S_1 occur in $(T/p_1) \geq 1$ combinations, the states in S_2 occur in $(T/p_2) \geq 1$ combinations. Hence, the C_k cover all recurrent states of the STG.

The generalization of this argument from two ESCCs to n ESCCs with recurrence periods $p_1, p_2, ..., p_n$ is straightforward. $\square$

Bibliography

[1] M. Abramovici, M. Breuer, and A. Friedman, *Digital Systems Testing and Testable Design*. Piscataway, New Jersey: IEEE Press, 1994.

[2] S. B. Akers, "A Logic System for Fault Test Generation," *IEEE Transactions on Computers*, vol. C-25, pp. 620–630, June 1976.

[3] S. B. Akers, "Binary Decision Diagrams," *IEEE Transactions on Computers*, vol. C-27, pp. 509–516, June 1978.

[4] R. L. Ashenhurst, "The Decomposition of Switching Functions," in *Proc. of an Intl. Symposium on the Theory of Switching held at Comp. Lab. of Harvard University*, pp. 74–116, 1959.

[5] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Algebraic Decision Diagrams and their Application," in *Proc. Intl. Conference on Computer-Aided Design (ICCAD-93)*, pp. 188–191, 1993.

[6] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic Model Checking using SAT Procedures instead of BDDs," in *Proc. Intl. Design Automation Conference (DAC-99)*, pp. 317–320, June 1999.

[7] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient Implementation of a BDD Package," in *Proc. Design Automation Conference (DAC-90)*, (Orlando, FL), pp. 40–45, June 1990.

[8] D. Brand, "Verification of Large Synthesized Designs," in *Proc. Intl. Conf. on Computer-Aided Design (ICCAD-93)*, pp. 534–537, 1993.

[9] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa, "VIS: A system for Verification and Synthesis," in *Proc. of the 8th Intl. Conference on Computer-Aided Verification, Springer Lecture Notes in Computer Science* (R. Alur and T. Henzinger, eds.), vol. 1102, (New Brunswick, NJ), pp. 428–432, July 1996.

[10] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: Multi-Level Interactive Logic Optimization System," *IEEE Transactions on Computer-Aided Design*, vol. 6, pp. 1062–1081, Nov. 1987.

[11] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[12] F. M. Brown, *Boolean Reasoning (The Logic of Boolean Equations)*. Kluwer Academic Publishers, 1990.

[13] R. Bryant and Y. A. Chen, “Verification of Arithmetic Functions by Binary Moment Diagrams,” in *Proc. Design Automation Conference (DAC-95)*, pp. 535–541, 1995.

[14] R. Bryant, “Graph-based Algorithms for Boolean Function Manipulation,” *IEEE Transactions on Computers*, vol. 35, pp. 677–691, August 1986.

[15] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill, “Sequential Circuit Verification using Symbolic Model Checking,” in *Proc. Intl. Design Automation Conference (DAC-90)*, pp. 46–51, June 1990.

[16] J. R. Burch and V. Singhal, “Robust Latch Mapping for Combinational Equivalence Checking,” in *Proc. Intl. Conference on Computer-Aided Design (ICCAD-98)*, 1998.

[17] G. Cabodi, P. Camurati, L. Lavagno, E. Macii, M. Poncino, S. Quer, and E. Sentovich, “Enhancing FSM Traversal by Temporary Re-Encoding,” in *Proc. Intl. Conference on Computer Design (ICCD-96)*, 1996.

[18] G. Cabodi, P. Camurati, L. Lavagno, and S. Quer, “Disjunctive Partitioning and Partial Iterative Squaring: an effective approach for symbolic traversal of large circuits,” in *Proc. Design Automation Conference (DAC-97)*, (Anaheim, CA), pp. 728–733, 1007.

[19] G. Cabodi, P. Camurati, and S. Quer, “Improved Reachability Analysis of Large Finite State Machines,” in *Proc. Intl. Conf. on Computer-Aided Design (ICCAD-96)*, pp. 354–360, November 1996.

[20] X. Chen and M. Bushnell, *Efficient Branch and Bound Search with Application to Computer-Aided Design*. Boston, MA: Kluwer Academic Publishers, 1996.

[21] H. Cho, G. D. Hachtel, E. Macii, M. Poncino, and F. Somenzi, “A Structural Approach for State Space Decomposition for Approximate Reachability Analysis,” in *Proc. Intl. Conference on Computer Design (ICCD-94)*, pp. 236–239, 1994.

[22] E. M. Clarke and E. Emerson, “Synthesis of synchronization skeletons for branching time temporal logic,” *Lecture Notes in Computer Science*, vol. 131, 1981.

[23] E. M. Clarke, M. Fujita, P. McGeer, K. L. McMillan, J. Yang, and X. Zhao, “Multi-Terminal Binary Decision Diagrams: an Efficient Data Structure for Matrix Representation,” in *Proc. Intl. Workshop on Logic Synthesis*, pp. (P6a) 1–15, 1993.

[24] O. Coudert and J. Madre, “Implicit and Incremental Computation of Primes and Essential Primes of Boolean Functions,” in *Proc. Design Automation Conference (DAC-92)*, pp. 36–39, 1992.

[25] O. Coudert, J. Madre, and H. Fraisse, "A New Viewpoint on Two-Level Logic Minimization," in *Proc. Design Automation Conference (DAC-93)*, pp. 625–630, 1993.

[26] O. Coudert, C. Berthet, and J.-C. Madre, "Verification of Synchronous Sequential Machines Based on Symbolic Execution," *Lecture Notes on Computer Science*, vol. 407, pp. 365–373, June 1989.

[27] H. A. Curtis, "A Generalized Tree Circuit," *Journal of the Association of Computing Machinery*, pp. 484–496, August 1961.

[28] M. Dagenais, V. Agarwal, and N. Rumin, "McBOOLE: A New Procedure for Exact Logic Minimization," *IEEE Transactions on CAD*, vol. CAD-5, pp. 229–232, January 1986.

[29] D. Defoe, *The Life and Adventures of Robinson Crusoe*. Edinburgh: Cadell, Davies, Strand and Blackwood, 1820.

[30] R. Drechsler, B. Becker, and S. Ruppertz, "K*BMDs: A New Data Structure for Verification," in *Proc. European Design & Test Conference*, pp. 2–8, 1996.

[31] R. Drechsler, B. Becker, A. Sarabi, M. Theobald, and M. Perkowski, "Efficient Representation and Manipulation of Switching Functions Based on Ordered Kronecker Functional Decision Diagrams," in *Proc. Design Automation Conference (DAC-94)*, pp. 415–419, 1994.

[32] L. A. Entrena and K. T. Cheng, "Sequential Logic Optimization by Redundancy Addition and Removal," in *Proc. Intl. Conference on Computer-Aided Design (ICCAD-93)*, pp. 310–315, November 1993.

[33] E. Fabricius, *Modern Digital Design and Switching Theory*. CRC Press, 1992.

[34] T. Filkorn, *Symbolische Methoden für die Verifikation endlicher Zustandssysteme*. PhD thesis, Institut für Informatik der Technischen Universität München, 1992.

[35] M. Fujita, H. Fujisawa, and N. Kawato, "Evaluation and improvements of Boolean Comparison Method Based on Binary Decision Diagrams," in *Proc. Intl. Conference on Computer-Aided Design*, (Santa Clara, CA), pp. 2–5, November 1988.

[36] M. Fujita, Y. Matsunaga, and T. Kakuda, "On Variable Ordering of Binary Decision Diagrams for the Application of Multi-Level Logic Synthesis," in *Proc. European Design Automation Conference (EDAC-91)*, pp. 50–54, March 1991.

[37] G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Probabilistic Analysis of Large Finite State Machines," in *Proc. 31st ACM/IEEE Design Automation Conference (DAC-94)*, pp. 270–275, 1994.

[38] G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Markovian Analysis of Large Finite State Machines," *IEEE Transactions on Computer-Aided Design*, vol. 15, pp. 1479–1493, Dec. 1996.

[39] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*. Boston: Kluwer Academic Publishers, 1996.

[40] A. J. Hu and D. L. Dill, "Reducing BDD Size by Exploiting Functional Dependencies," in *Proc. 30th ACM/IEEE Design Automation Conference (DAC-92)*, pp. 522–525, 1992.

[41] S. Huang, K. Cheng, K. Chen, and U. Gläser, "An ATPG-Based Framework for Verifying Sequential Equivalence," in *Proc. Intl. Test Conference (ITC-96)*, 1996.

[42] S. Huang, K. Cheng, K. Chen, and U. Gläser, "On Verifying the Correctness of Retimed Circuits," in *Proc. Great Lakes Symposium on VLSI*, 1996.

[43] N. Ishiura, H. Sawada, and S. Yajima, "Minimization of Binary Decision Diagrams Based on Exchanges of Variables," in *Proc. Intl. Conference on Computer-Aided Design (ICCAD-91)*, pp. 472–475, 1991.

[44] J. Jain, R. Mukherjee, and M. Fujita, "Advanced Verification Techniques Based on Learning," in *Proc. 32nd ACM/IEEE Design Automation Conference (DAC-95)*, pp. 420–426, June 1995.

[45] S.-W. Jeong, B. Plessier, G. D. Hachtel, and F. Somenzi, "Variable Ordering for FSM Traversal," in *Proc. Intl. Workshop on Logic Synthesis*, (MCNC, Research Triangle Park, NC), May 1991.

[46] U. Kebschull, E. Schubert, and W. Rostenstiel, "Multi-Level Logic Based on Functional Decision Diagrams," in *Proc. European Design Automation Conference (EDAC-92)*, pp. 43–47, 1992.

[47] J. G. Kemeny and J. L. Snell, *Finite Markov Chains*. New York: Springer-Verlag, 1976.

[48] Z. Kohavi, *Switching and Finite Automata Theory*. McGraw-Hill, 1978.

[49] T. Kropf and H. Wunderlich, "A Common Approach to Test Generation and Hardware Verification Based on Temporal Logic," in *Proc. Intl. Test Conference (ITC-91)*, pp. 57–66, 1991.

[50] A. Kühlmann and F. Krohm, "Equivalence Checking Using Cuts and Heaps," in *Proc. Design Automation Conference (DAC-97)*, pp. 263–268, Nov. 1997.

[51] W. Kunz, "An Efficient Tool for Logic Verification Based on Recursive Learning," in *Proc. Intl. Conference on Computer-Aided Design (ICCAD-93)*, pp. 538–543, Nov. 1993.

[52] W. Kunz and P. Menon, "Multi-Level Logic Optimization by Implication Analysis," in *Proc. Intl. Conf. on Computer-Aided Design (ICCAD-94)*, San Jose, pp. 6–13, November 1994.

[53] W. Kunz and D. Pradhan, "Recursive Learning: A New Implication Technique for Efficient Solutions to CAD Problems: Test, Verification and Optimization," *IEEE Transactions on Computer-Aided Design*, vol. 13, pp. 1143–1158, Sep. 1994.

[54] W. Kunz and D. Stoffel, *Reasoning in Boolean Networks - Logic Synthesis and Verification Using Testing Techniques*. Boston: Kluwer Academic Publishers, 1997.

[55] W. Kunz, D. Stoffel, and P. Menon, "Multi-Level Logic Optimization and Equivalence Checking by Implication Analysis," *IEEE Transactions on Computer-Aided Design*, vol. 16, pp. 266–281, March 1997.

[56] R. P. Kurshan, *Computer-Aided Verification of Coordinating Processes — The Automata-Theoretic Approach*. Princeton, New Jersey: Princeton University Press, 1994.

[57] Y. T. Lai and S. Sastry, "Edge-Valued Binary Decision Diagrams for Multi-Level Hierarchical Verification," in *Proc. Design Automation Conference (DAC-95)*, pp. 254–260, 1995.

[58] E. L. Lawler, "An Approach to Multilevel Boolean Minimization," *Journal of the ACM*, vol. 11, pp. 283–295, July 1964.

[59] H. K. Lee and D. S. Ha, "An Efficient Forward Fault Simulation Algorithm Based on the Parallel Pattern Single Fault Propagation," in *Proc. Intl. Test Conference (ITC-91)*, pp. 946–955, October 1991.

[60] C. Leiserson and J. Saxe, "Retiming Synchronous Circuitry," *Algorithmica*, vol. 6, pp. 5–35, 1991.

[61] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli, "Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment," in *Proc. Intl. Conference on Computer-Aided Design (ICCAD-88)*, pp. 6–9, November 1988.

[62] Y. Matsunaga, "An Efficient Equivalence Checker for Combinational Circuits," in *Proc. Design Automation Conference (DAC-96)*, pp. 629–634, June 1996.

[63] E. McCluskey, "Minimization of Boolean Functions," *Bell System Technical Journal*, vol. 35, pp. 1417–1444, 1956.

[64] E. McCluskey, *Logic Design Principles*. Englewood Cliffs, NJ: Prentice-Hall, 1986.

[65] P. McGeer, J. Sanghavi, R. Brayton, and A. Sangiovanni-Vincentelli, "ESPRESSO-SIGNATURES: A New Exact Minimizer for Logic Functions," in *Proc. Design Automation Conference (DAC-93)*, pp. 618–621, 1993.

[66] K. McMillan, *Symbolic Model Checking*. Boston: Kluwer Academic Publishers, 1993.

[67] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[68] S. Muroga *et al.*, "The Transduction Method – Design of Logic Networks Based on Permissible Functions," *IEEE Transactions on Computers*, vol. C-38, pp. 1404–1424, Oct. 1989.

[69] P. Muth, "A Nine-Valued Logic Model for Test Generation," *IEEE Transactions on Computers*, vol. C-25, pp. 630–636, June 1976.

[70] C. Pixley, "A Theory and Implementation of Sequential Hardware Equivalence," *IEEE Transactions on Computer-Aided Design*, vol. 11, pp. 1469–1478, Dec. 1992.

[71] C. Pixley, V. Singhal, A. Aziz, and R. K. Brayton, "Multi-level Synthesis for Safe-Replaceability," in *Proc. Intl. Conference on Computer-Aided Design (ICCAD-94)*, pp. 442–449, 1994.

[72] S. Quer, G. Cabodi, P. Camurati, L. Lavagno, E. Sentovich, and R. K. Brayton, "Incremental Re-Encoding for Symbolic Traversal of Product Machines," in *Proc. European Design Automation Conference (EDAC-96)*, 1996.

[73] W. Quine, "The Problem of Simplifying Truth Functions," *American Mathematical Monthly*, vol. 59, pp. 521–531, 1952.

[74] J. Rajski and J. Vasudevamurthy, "Testability Preserving Transformations in Multi-Level Logic Synthesis," in *Proc. Intl. Test Conference (ITC-90)*, pp. 265–273, 1990.

[75] K. Ravi and F. Somenzi, "High Density Reachability Analysis," in *Proc. Intl. Conference on Computer-Aided Design (ICCAD-95)*, pp. 154–158, 1995.

[76] S. Reddy, W. Kunz, and D. Pradhan, "A Novel Verification Framework Combining Structural and OBDD Methods in a Synthesis Environment," in *Proc. Design Automation Conference (DAC-95)*, pp. 414–419, June 1995.

[77] E. Rich, *Artificial Intelligence*. McGraw-Hill, 1983.

[78] J. P. Roth, "Diagnosis of Automata Failures: A Calculus and a Method," *IBM Journal of Research and Development*, vol. 10, pp. 278–291, July 1966.

[79] J. P. Roth and R. M. Karp, "Minimization over Boolean Graphs," *IBM Journal of Research and Development*, vol. 10, pp. 278–291, July 1966.

[80] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," in *Proc. Intl. Conference on Computer-Aided Design (ICCAD-91)*, pp. 42–47, 1993.

[81] R. Rudell and A. Sangiovanni-Vincentelli, "Multiple-valued Minimization for PLA Optimization," *IEEE Transactions on Computer-Aided Design*, vol. CAD-6, pp. 727–750, September 1987.

[82] H. Savoj, R. K. Brayton, and H. Touati, "Extracting Local Don't-Cares for Network Optimization," in *Proc. Intl. Conference on Computer-Aided Design (ICCAD-91)*, pp. 514–517, November 1991.

[83] M. Schulz, E. Trischler, and T. Sarfert, "SOCRATES: A Highly efficient automatic test pattern generation system," in *Proc. Intl. Test Conference (ITC-87)*, pp. 1016–1026, 1987.

[84] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," Tech. Rep. Memorandum No. UCB/ERL M92/41, Electronics Research Laboratory, University of California at Berlekey, 1992.

[85] C. Shannon, "A Symbolic Analysis of Relay and Switching Circuits," *Transactions AIEE*, vol. 57, pp. 713–723, 1938.

[86] D. Stoffel and W. Kunz, "AND/OR Reasoning Graphs for Determining Prime Implicants in Multi-level Combinational Networks," in *Proc. Asia and South Pacific Design Automation Conference (ASPDAC-97)*, pp. 529–538, January 1997.

[87] D. Stoffel and W. Kunz, "Logic Equivalence Checking by Optimization Techniques," in *Proc. International Workshop on Computer-Aided Design, Test, and Evaluation for Dependability*, (Peking, China), pp. 85–90, July 1996.

[88] D. Stoffel and W. Kunz, "Record & Play: A Structural Fixed Point Iteration for Sequential Circuit Verification," in *Proc. Intl. Conference on Computer-Aided Design (ICCAD-97)*, pp. 394–399, Nov 1997.

[89] D. Stoffel and W. Kunz, "Structural FSM Traversal — Theory and a Practical Algorithm," Tech. Rep. 005/1997, Dept. of Computer Science, Universit of Potsdam, Germany, Nov. 1997.

[90] P. Tafertshofer, A. Ganz, and M. Henftling, "A SAT-Based Implication Engine for Efficient ATPG, Equivalence Checking, and Optimization of Netlists," in *Proc. of the Intl. Conference on Computer-Aided Design (ICCAD-97)*, pp. 648–655, November 1997.

[91] H. J. Touati, H. Savoj, B. Lin, and R. K. Brayton, "Implicit State Enumeration of Finite State Machines using BDDs," in *Proc. of the Intl. Conference on Computer-Aided Design (ICCAD-90)*, pp. 130–133, 1990.

[92] C. van Eijk, *Formal Methods for the Verification of Digital Circuits*. PhD thesis, Eindhoven University of Technology, 1997.

[93] C. van Eijk, "Sequential Equivalence Checking without State Space Traversal," in *Proc. Conference on Design, Automation and Test in Europe (DATE-98)*, (Paris, France), pp. 618–623, March 1998.

[94] I. Wegener, *The Complexity of Boolean Functions*. Stuttgart: B. G. Teubner, 1987.

Curriculum Vitae

Dominik Stoffel, born April 2nd, 1966, in Xenia, OH, USA

1972 – 1975	Elementary School Sulz am Eck
1975 – 1978	Hermann-Hesse-Gymnasium Calw (Junior High School)
1978 – 1979	Junior High School, Wappinger Falls, NY, USA
1979 – 1982	Otto-Hahn-Gymnasium Furtwangen (Senior High School)
1982 – 1985	Fürstenberg-Gymnasium Donaueschingen (Senior High School)
26/6/85	Abitur (graduation from high school)
1985 – 1992	Undergraduate/graduate studies in Electrical Engineering at the University of Karlsruhe, Germany.
5/89 – 8/89	Summer internship with Johnson Controls, Inc., Milwaukee, WI, USA. Development of an LPCVD system for producing silicon nitride films to be used in silicon microsensors.
4/92 – 6/93	Project work and master's thesis at FZI Research Center for Information Technologies at the University of Karlsruhe, Dept. for Microcomputer Technology (head: Prof. Dr. Klaus Bender). Development of a testing tool for distributed intelligent microsystems.
23/12/92	Graduation from university with the academic degree of Diplom-Ingenieur in Electrical Engineering.
8/93 – 9/94	Employment with Mercedes-Benz, Sindelfingen. Development of testing equipment for automotive electronics.
10/94 – 5/98	Ph.D. student and research assistant with Max Planck Fault Tolerant Computing Group (since 1/98 part of the Dept. of Computer Science, head: Prof. Dr. Michael Gössel) at the University of Potsdam.
since 6/98	Ph.D. student and research assistant with Prof. Dr. Wolfgang Kunz, Design Automation Group, Dept. of Computer Science, University of Frankfurt, Germany. Research topics: Computer-Aided Design (CAD) for VLSI circuits, logic synthesis and formal hardware verification.