Using SAT Checkers to Solve The Logic Minimization Problem

Samir Sapra
School of Computer Science
Carnegie Mellon University

Advisor: Edmund Clarke
Post-Doc Advisor: Michael Theobald
May 3, 2003

DRAFT

Abstract

We investigate new techniques for solving the two-level logic minimization problem: Given a Boolean function, find the smallest OR-of-ANDs expression that represents it. This is a problem of both theoretical and practical interest. It arises in several fields of Computer Science, such as digital design, reliability analysis and automated reasoning.

This project explores a new approach that involves the use of SAT checkers: programs that accept as input a Boolean formula f using only ANDs, ORs and NOTs, and determine whether the given formula f is satisfiable or not. To some, this approach might seem counter-intuitive; the SAT problem is NP-complete, and so it may appear inefficient to base our techniques on it. Thus, it is a surprise to learn that the SAT-based approach has shown considerable promise in our work.

As a by-product, our research has yielded a number of ideas and observations that can be explored in other contexts, such as hardware verification, software verification, and QBF checking (satisfiability checking for a class of formulae that is 'harder' than what SAT checkers can handle).

Contents

Chapter 1

Introduction

Recent advances in satisfiability checkers have induced breakthroughs in various areas of computer science, such as digital design and verification. It is interesting to consider how these advances may be exploited to improve the performance of logic minimization algorithms, which themselves have several applications in computer science. Studies like Coudert's [4] have investigated the use of other techniques (like BDD-manipulation algorithms) to solve the two-level logic minimization problem. We focus on applying satisfiability checkers.

The two-level logic minimization problem can be informally stated as follows: Given a Boolean function, find the smallest OR-of-ANDs expression that represents it. This problem is encountered in several practical areas of computer science, such as in the fields of logic synthesis, artificial intelligence, and reliability analysis [4, 10]. As an example, consider its application to logic synthesis. Every combinational circuit C implements a Boolean function on n variables. Further, every Boolean function can be realized in OR-of-ANDs form (also known as sum-of-products form) — informally, the form $f = \sum l_1 l_2 ... l_n$, where each l_i is either a Boolean variable or its complement. By finding a minimum sum-of-products representation for f, we can produce an efficient combinational logic design.

The problem is also of theoretical interest. Umans [19] studied its computational complexity and established that when the input function is specified in sum-of-products form, then the problem is $\sum_2^p$ *−complete*. ($\sum_2^p$ *−complete* denotes a class of problems that are 'harder' than NP-complete problems. An example of a $\sum_2^p$ *−complete* problem is to determine the satisfiability of a quantified Boolean formula with two alternating levels of quantification: $\exists X_1 \forall X_2 \phi$ where X_1 and X_2 are disjoint sets of variables and ϕ is in sum-of-products form.)

As mentioned previously, our approach to the two-level logic minimization problem involves the use of *SAT checkers*. A SAT checker is a program that accepts as input a Boolean formula f using only ANDs, ORs and NOTs, and determines whether the formula f is satisfiable or not. To some, this approach might seem counter-intuitive. The SAT problem is NP-complete, and so it may appear inefficient to base our techniques on it. However, in practice, SAT checkers perform very well and are able to solve 'real-world' formulae containing hundreds of thousands of variables within minutes. This high performance has resulted from recent breakthroughs in the strategies and heuristics used by SAT algorithms [14, 17, 20]. These breakthroughs have motivated the application of SAT checkers in a number of other contexts (such as model checking), yielding positive results [12, 3].

We explore SAT-based approaches for both *exact* and *heuristic* minimization. (Since solving the two-level logic minimization problem can be computationally expensive, heuristic approaches have been developed as well.) State-of-the-art heuristic minimizers like Espresso-II are used world-wide. Given its practical appliability, our focus in terms of implementation and results is on heuristic minimization.

The minimization strategies used by Espresso-II almost always lead to (near-)minimum solutions in practice. However, for large problems (functions with over 200 input variables) Espresso-II takes a long time to execute. SAT checkers, on the other hand, have recently become capable of handling comparatively huge numbers of variables. We therefore try to combine the strengths of Espresso-II (quality of approximation) and SAT (speed on large problems) by adopting the same basic strategies as Espresso-II but performing them efficiently by developing algorithms that use (appropriately adapted) SAT checkers.

The remainder of this document is structured as follows: Chapter 2 introduces useful definitions and background material. Chapters 3 and 4 discuss our techniques for applying SAT to heuristic minimization, with Chapter 3 focusing more on the theory and algorithms and Chapter 4 focusing more on the experimental results. In Chapter 5, we develop exact minimization strategies that use SAT and QBF checkers (QBF checkers being algorithms that test the satisfiability of quantified Boolean formulae). Finally, in Chapter 6 we summarize the work done and draw conclusions.

`[TO-DO: Expand]`

Chapter 2

Background

This chapter first reviews a number of fundamental definitions in Boolean algebra and logic synthesis. Then, two-level logic minimization algorithms are surveyed. This background forms the basis for the new approaches presented in subsequent chapters.

Some concepts are crucial to the understanding of specific chapters (but not others). Introduction of such concepts has been deferred to the appropriate chapters.

2.1 Basic Logic Synthesis - Definitions

The following definitions are taken from Theobald [18], Rudell [16], and standard textbooks [9, 8] with small modifications.

Let $\mathbb{B} := \{0, 1\}$ be the set of binary values. $\mathbb{B}^n$ can be modeled as a binary n-cube, and each element $e = (e_1, \ldots, e_n) \in \mathbb{B}^n$ is called a **minterm**. Figure 2.1(a) shows $\mathbb{B}^3$, the three-dimensional Boolean space, and its minterms. Note that the well-known binary Boolean algebra is given by the the set $\mathbb{B}$ together with the operations $+$ (also called disjunction, sum, OR) and $\cdot$ (conjunction, product, AND).

A **Boolean function** f of n variables, $x_1, \ldots, x_n$, is a mapping $f : \mathbb{B}^n \rightarrow \{0, 1, *\}$. Here, the symbol $*$ denotes a *don't care* condition, i.e. the value of the function does not matter. Note that a minterm $(e_1, \ldots, e_n)$ indicates which values are assigned to the variables of a function, i.e. $x_1 = e_1, x_2 = e_2$, and so on.

The **ON-set** of a Boolean function f is defined as the set of minterms for which the function has value 1. Similarly, the **OFF-set** and **DON'T-CARE-set** are defined as the sets of minterms for which the function has value 0 and *, respectively.

Boolean functions as defined above are often referred to as *single-output* Boolean functions. A *multi-output* Boolean function is a mapping $f : \mathbb{B}^n \rightarrow \{0, 1, *\}^m$. Note that each

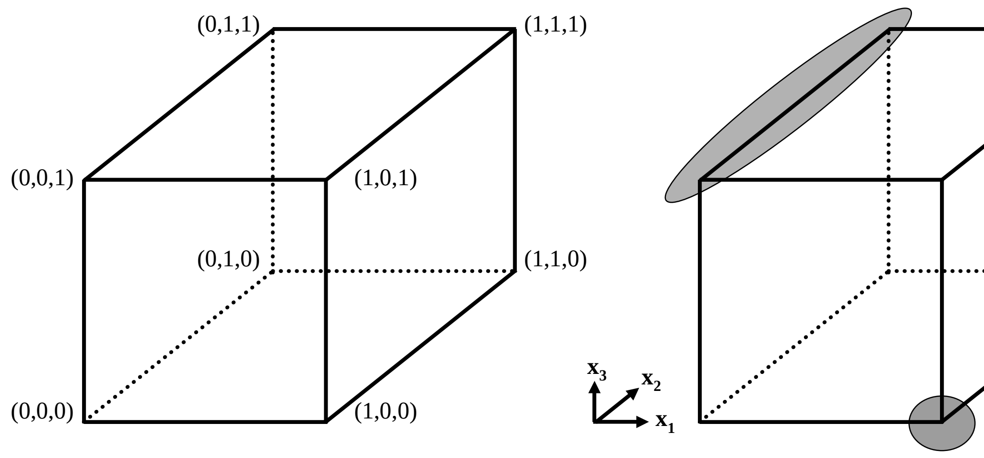


Figure 2.1: (a) Binary cube $\mathbb{B}^3$ and its minterms; (b) Minterm $x_1\overline{x}_2\overline{x}_3$ and cube $\overline{x}_1x_3$

of the output functions $f_1, \ldots, f_m$ has its own ON-set, OFF-set, and DON'T-CARE-set associated with it. For the sake of simplicity of presentation, only single-output functions are considered in the remainder of this section.

Each variable x_i has two **literals** associated with it: an *uncomplemented* (or *positive*) literal x_i, and a *complemented* (or *negative*) literal $\overline{x}_i$ or x_i'. The literal x_i ($\overline{x}_i$) represents a Boolean function which evaluates to 1 (0) for minterms with $e_i = 1$, and to 0 (1) for minterms with $e_i = 0$.

A **product** term is a Boolean product (AND) of literals. That is, a product evaluates to 1 for a minterm e, if each literal *included* in the product evaluates to 1 for the minterm e. Otherwise, the product evaluates to 0. In the former case, the product is said to *contain* minterm e. Note that each minterm corresponds to a product that only contains the given minterm. More specifically, the minterm $e = (e_1, \ldots, e_n)$ corresponds to the product $x_1^{e_1} \cdots x_n^{e_n}$, where $x_i^{e_i}$ denotes the positive (negative) literal of x_i if $e_i = 1(0)$. For example, the minterm $e = (1, 0, 1)$ corresponds to the product $x_1\overline{x_2}x_3$, which is often used as a convenient abbreviation.

Since a product corresponds to a set of adjacent minterms in the binary n-cube, a product is also referred to as a **cube**. Figure 2.1(b) shows a cube and a minterm in $\mathbb{B}^3$.

A cube α is *contained in* a cube β ($\alpha \subseteq \beta$) if each minterm contained in α is also contained in β. The *intersection* of cubes α and β ($\alpha \cap \beta$) is the uniquely defined cube which contains those minterms contained in both cubes. The *supercube* of cubes α and β, denoted *supercube*(α, β), is the uniquely defined smallest cube that contains both cubes. For example, if $\alpha = x_1\overline{x}_2$, and $\beta = \overline{x}_1\overline{x}_2x_3$, then *supercube*$(\alpha, \beta) = \overline{x_2}$. In general, to compute the supercube each literal is considered. A literal is included in the supercube of two cubes if and only if it is included in both cubes.

A **sum-of-products** is a Boolean sum (OR) of products. That is, a sum-of-products evaluates to 1 for a given minterm if some product contains the minterm.

An **implicant** of a Boolean function is a cube which contains no minterm in the OFF-set. A **prime implicant** is an implicant contained in no other implicant of the function. An **essential prime implicant** is a prime implicant containing at least one ON-set minterm which is not contained in any other prime implicant.

A **cover** of a Boolean function is a set of implicants interpreted as a sum-of-products, which evaluates to 1 for all the minterms of the ON-set, and none of the OFF-set. We use the term *prime cover* to refer to a cover containing only prime implicants.

The **complement** of a Boolean function f is denoted by $\overline{f}$, and evaluates to 1 (0) if f evaluates to 0 (1).

2.2 2-Level Logic Minimization

The *two-level logic minimization problem* is to find a 'sum-of-products' representation for f that minimizes a given cost function. Or equivalently, the problem is find a cover of f that minimizes a given cost function. In digital design, such a cover can be implemented as a minimum-cost OR-of-ANDs (two-level) circuit. Here, the cost, or size, of a cover is often defined as the number of cubes in the cover, which will be used in the following presentation.

Figure 2.2 given an example of a digital design application of two-level logic minimization. Here we are given function $f = \overline{x}(x + y) + \overline{(\overline{x} + x\overline{z})} + xyz + \overline{x}yz$ and we want to find an efficient way of implementing it in hardware. A two-level logic minimizer would give us the sum-of-products representation $f = \overline{x}y + xz$, which is much smaller than the original representation.

This section surveys the most significant previous algorithms for solving the two-level logic minimization problem.

Quine-McCluskey

The classic Quine-McCluskey algorithm [11, 16] to solve the two-level minimization problem is based on the insight that the implicants in a minimum-cost cover can be restricted to prime implicants. (Assume a minimum-cost cover included a non-prime implicant, then the non-prime implicant could be replaced by a prime implicant covering it.)

The Quine-McCluskey algorithm consists of two steps:

1. generate the set of all prime implicants;

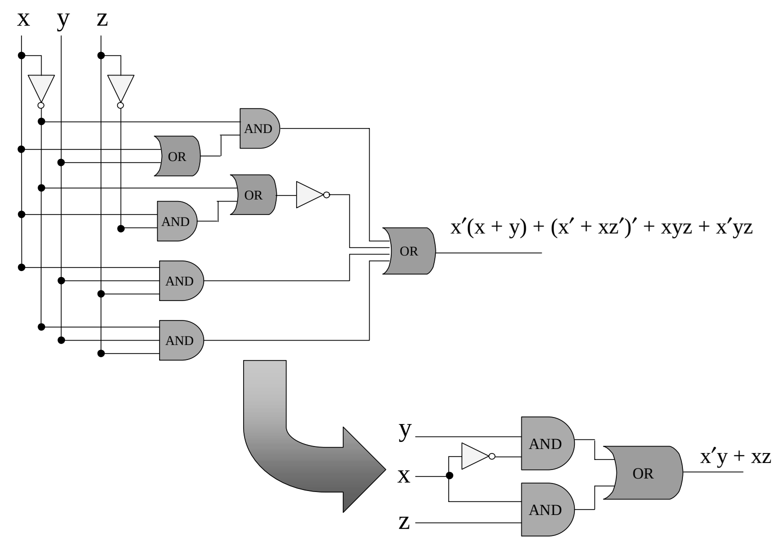


Figure 2.2: An example of how two-level logic minimization might be applied in the field of digital design

2. select a minimum number of prime implicants such that each ON-set minterm is contained.

The first step — to generate the set of prime implicants — starts from the set of ON-set minterms, and it iteratively computes larger implicants by removing literals. The prime implicants are then those implicants from which no more literals can be removed.

The second step is a so-called *unate set covering problem* [8]: Given two sets X and Y, and a relation $R \subseteq X \times Y$, find a minimum-cardinality subset S of Y such that

$$\forall x \in X \; \exists y \in S \; ((x, y) \in R)$$

More recent algorithms for two-level logic minimization follow the QUINE-MCCLUSKEY algorithm but with a number of improvements.

SCHERZO

SCHERZO [4, 7, 5, 6] is currently the state-of-the-art exact two-level logic minimization algorithm. Using *implicit* minimization techniques, i.e. using data structures that facilitate the manipulation of a large number of objects simultaneously, SCHERZO is 10 to more than 100 times faster than the best previous minimization methods. In addition, SCHERZO has solved examples with 10^{20} prime implicants. These huge examples are by far out of the reach of previous minimization algorithms.

Espresso-II

Since solving the 2-level logic minimization problem involves computationally intractable problems, heuristic approaches have been developed as well.

Espresso-II [15, 2] is the state-of-the-art tool for *heuristic* two-level logic minimization. The output of Espresso-II is a cover, which in practice is almost always (near-)minimum in cardinality. The tool is very efficient and is used world-wide.

Chapter 3

Heuristic Minimization Using SAT Checkers

As mentioned previously, solving the two-level logic minimization problem can be computationally expensive. Hence, heuristic tools like ESPRESSO-II have been developed as powerful practical alternatives. While ESPRESSO-II almost always produces (near-)minimum solutions in practice, it takes a long time to solve large problems.

With regard to heuristic minimization, our aim is to achieve high-quality approximations efficiently for large problems. We try to combine the strengths of ESPRESSO-II (quality of approximation) and SAT (speed on large problems) by adopting the same basic strategies as ESPRESSO-II but by performing them efficiently using SAT-based algorithms.

ESPRESSO-II's strategies are implemented by various procedures called 'operators'. The major ESPRESSO-II operators are called EXPAND, IRREDUNDANT, REDUCE and ESSENTIALS (these are described in more detail in the following section). ESPRESSO-II's runtime profile was analyzed for some large examples that we were interested in, and our analysis indicated that the major bottlenecks were REDUCE, ESSENTIALS and IRREDUNDANT (in that order). The EXPAND operator wasn't a bottleneck and executed quickly even on large examples with 200 variables. Accordingly, our efforts were focused on developing SAT-based algorithms for the other three operators.

In this chapter, we begin by giving background on ESPRESSO-II, since its basic minimization strategy is similar to the one that we use. We then develop SAT-based algorithms to implement the IRREDUNDANT, REDUCE and ESSENTIALS operators.

3.1 Background on Espresso-II

Espresso-II, developed in the early 1980s, is a very powerful tool for heuristic two-level logic minimization, and the tool is used world-wide.

The input to Espresso-II is the Boolean function to be minimized, specified in terms of an arbitrary set of implicants (e.g. all ON-set minterms, or possibly larger cubes). This set of implicants represents an initial unoptimized cover, or solution. The output of Espresso-II is a cover, which is in practice almost always (near-) minimum in cardinality.

Espresso-II *iteratively* refines the cover by applying three ‘operators’ in its main loop. This iteration continues until no further improvement is possible:

- EXPAND enlarges each implicant of the current cover, in turn, into a prime implicant.
- IRREDUNDANT makes the current cover irredundant by deleting a maximal number of redundant implicants from the cover.
- REDUCE sets up a cover that is likely to be made smaller by the following EXPAND step. To achieve this goal, each cube in the current cover is maximally reduced, in turn, to a smaller cube such that the resulting set of cubes is still a cover.

Example. Figure 3.1 illustrates the EXPAND operator. As indicated in the example, it may be possible to expand an implicant along alternative dimensions. The actual algorithm uses heuristics to decide which direction is best.

Figure 3.2 illustrates the main loop of the algorithm. In Part A an initial cover of the function is given as its set of ON-set minterms. Part B presents the cover that results after EXPAND is applied to only one minterm, the one labeled 1. The new expanded cube includes other cubes of the cover, which are consequently deleted from the cover. Part C shows the cover as obtained after applying the EXPAND operator to the entire initial cover, resulting in a cover of size five. Next, the application of the operator IRREDUNDANT removes the one redundant cube from the cover, and the result is shown in Part D. The cover now consist of four cubes. Then, the operator REDUCE is applied to maximally reduce cubes in turn; the outcome is shown in Part E. Two of the four cubes were transformed into smaller cubes, yet the collection of cubes still covers all ON-set minterms. Finally, EXPAND is applied a second time. As the EXPAND operator considers one of the two smaller cubes, it is determined that this cube can be expanded to overlap the other smaller cube, thus reducing the size of the cover from four to three. The resulting cover is a minimum cover and no further application of the above operators will change it. Accordingly, the iteration stops. □

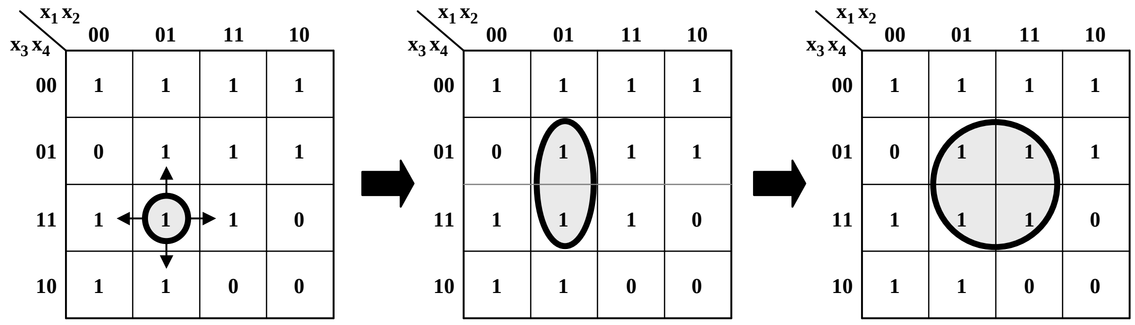


Figure 3.1: Expand in Espresso-II

Espresso-II also employs additional operators, such as ESSENTIALS and LAST_GASP, which can be quite powerful. ESSENTIALS is used to identify all essential prime implicants before the main loop is entered, in order to simplify the covering problem. LAST_GASP is applied after the main loop is exited, to try to escape a suboptimal local minimum; if successful the main loop is entered again.

One key reason for the efficiency of Espresso-II is the so-called *unate recursive paradigm*, i.e., to decompose operations recursively leading to efficiently solvable sub-operations on unate functions (i.e. functions that are unate in all of their variables). [1]

3.2 Irredundant

The IRREDUNDANT operator takes a cover produced by EXPAND and tries to reduce its cardinality to a local minimum. To understand this better, let us first distinguish between the notions of *irredundant cover* and *IRREDUNDANT operator*.

Definition 3.2.1 *An* **irredundant cover** *of a function is one that is not a proper superset of any cover of the same function.* For example, in Figure 3.3, (a) represents a redundant (i.e. not irredundant) cover of a logic function, whereas (b) and (c) represent irredundant covers.

It follows from the definition that the cardinality of an irredundant cover is at a local minimum. Removal of any implicant from an irredundant cover makes the cover invalid. Equivalently, in an irredundant cover, all cubes are *relatively essential*, where relative essentiality is defined as follows:

[1] A function is *unate in a variable* if changing the value of that variable from 0 to 1 either never changes the function's value from 0 to 1 or never changes the function's value from 1 to 0. A function would *not* be unate in a variable if changing the value of that variable from 0 to 1 sometimes changed the function's value from 0 to 1 and sometimes changed it from 1 to 0 depending on the values of the remaining variables.

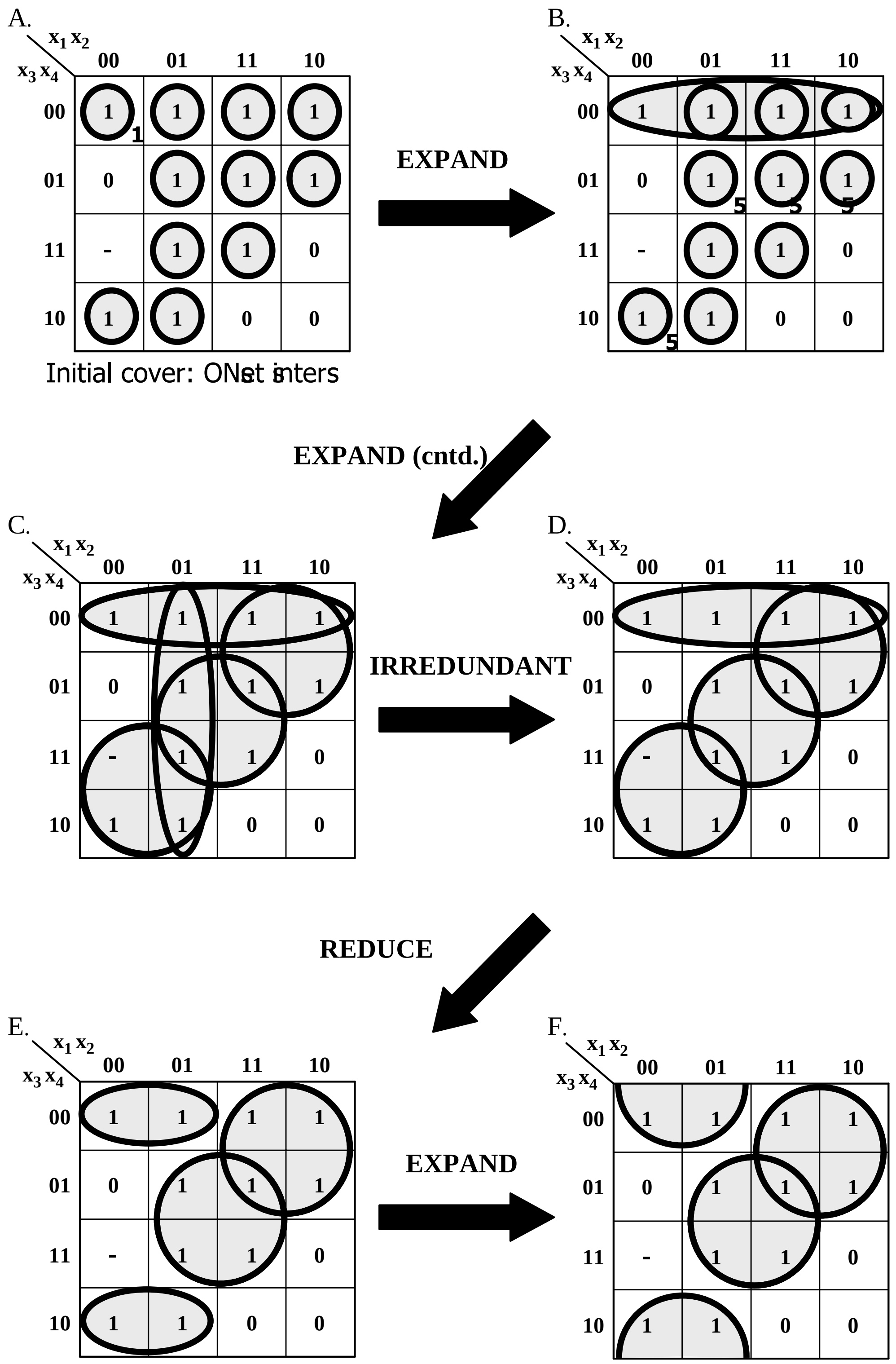


Figure 3.2: Espresso-II Example

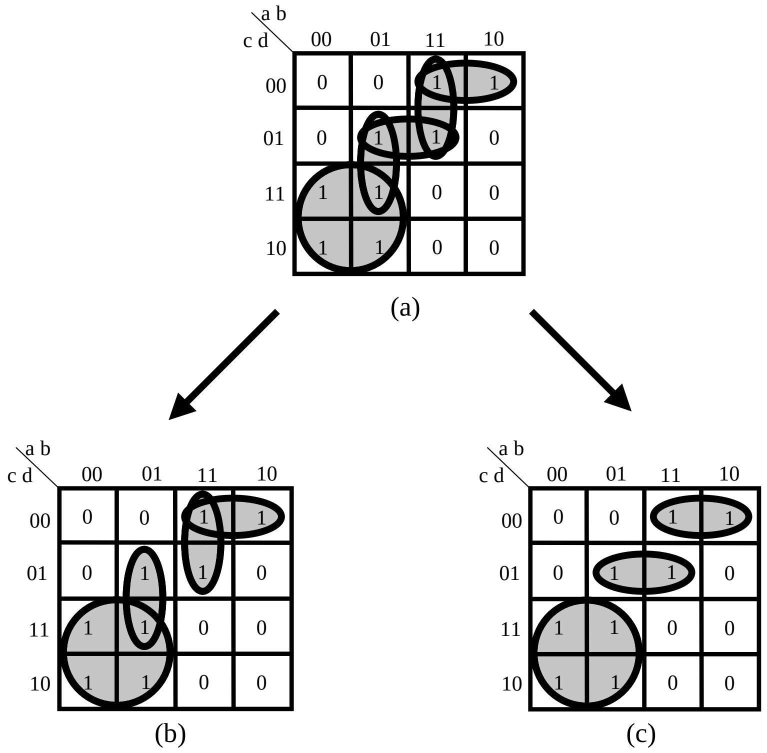


Figure 3.3: (a) Redundant cover of a logic function (b) Irredundant cover (c) Smallest irredundant cover

Definition 3.2.2 *Given a cover F of a Boolean function f, an implicant $\alpha \in F$ is said to be* **relatively essential** *if it contains an ON-set minterm of f that is not contained in any other implicant of F.* For example, in the irredundant covers (b) and (c) all of the implicants are relatively essential, but in the redundant cover (a) only the implicants $\overline{a}c$ and $a\overline{c}\overline{d}$ are relatively essential.

The **IRREDUNDANT operator**, on the other hand, is a procedure that makes a cover irredundant by removing some of its implicants. Given a redundant cover, there may be multiple ways of removing implicants from it, possibly yielding different irredundant covers (see Figure 3.3). Some of these irredundant covers may have lower cost than others, making covers like Figure 3.3(c) more desirable than those like Figure 3.3(b). To address this issue, Espresso-II's IRREDUNDANT operator sets up and solves an optimization problem to find a largest subset of implicants that can be removed from the given cover without making it invalid.

Currently, our implementation uses a simple-minded algorithm where we test each implicant α in a cover F for relative essentiality. Whenever we encounter an implicant α that fails the test, we delete it from F. This algorithm does produce an irredundant cover, but the resulting quality may be suboptimal. For example, if we are given Figure 3.3(a) and start by testing $b\overline{c}d$, we end up with Figure 3.3(b). `[TO-DO for final draft: In practice, what was the observed loss in qly vs speedup already due to SAT?]`. In the future, we might be able to develop an efficient SAT-based algorithm to handle the optimization problem as well.

In our algorithm, SAT is employed in the test for relative essentiality. For $\alpha \in F$ to be relatively essential, there must exist a witness ON-set minterm that is contained in α but not in any other implicant of F. That is, the following formula must be satisfiable:

$$\alpha \cdot \prod_{\substack{\beta \in F \\ \beta \neq \alpha}} \overline{\beta} \cdot \left(\sum_{\gamma \in F^{ON}} \gamma \right) \tag{3.1}$$

where α, β and γ are of course cubes over $x_1, x_2, \ldots, x_n$. It is straightforward to convert this formula into CNF and feed it to a SAT checker.

As an optimization, from the summation part of formula (3.1), we can exclude those γ that are disjoint from α. As mentioned previously, disjointness of implicants can be computed efficiently using bitwise operators.

3.3 Reduce

The purpose of the REDUCE operator is to modify the current cover so that its cardinality may be improved by the following EXPAND. Each implicant in a given cover is *maximally reduced* in size, i.e. reduced to the smallest cube such that the resulting set of implicants is still a cover.

Consider the example shown in Figure 3.4. Note that the end result of REDUCE depends on the order in which implicants are processed. In Figure 3.4, (b) was obtained from (a) by reducing $\overline{a}$ before $\overline{c}$, whereas (c) was obtained by reducing $\overline{c}$ before $\overline{a}$. Various heuristics have been developed to sort a cover before reducing its implicants. In ESPRESSO-II, implicants are weighted and then sorted in descending order of weight so as to first process those that are large and overlap many other implicants. Our implementation reuses the techniques adopted by ESPRESSO-II.

Let us now consider how to compute a maximally reduced cube. We are given a (sorted) cover F and an implicant $\alpha \in F$. Reducing α to the cube $\widetilde{\alpha}$ results in a new set of cubes $F' = (F - \{\alpha\}) \cup \{\widetilde{\alpha}\}$. We want to find the smallest cube $\widetilde{\alpha}$ that makes F' a cover. Any $\widetilde{\alpha}$ that makes F' a cover must contain all of the ON-set minterms of α that are not contained

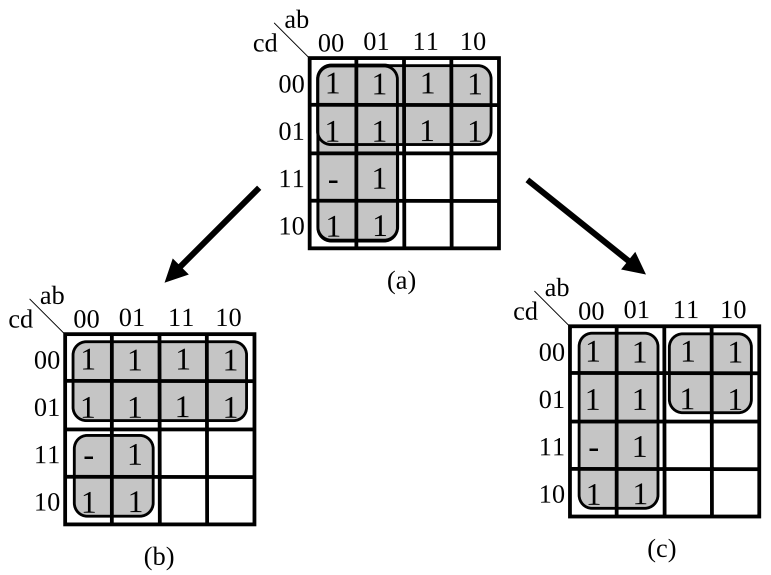


Figure 3.4: REDUCE is order-dependent

in any other implicant of F. The smallest $\widetilde{\alpha}$ that contains all of these minterms is simply their *supercube* (by definition).

Equivalently, we are looking for the supercube of all satisfying assignments of the following formula:

$$\alpha \cdot \prod_{\substack{\beta \in F \\ \beta \neq \alpha}} \overline{\beta} \cdot \left(\sum_{\gamma \in F^{ON}} \gamma \right) \tag{3.2}$$

It is straightforward to convert formula (3.2) into CNF. Note the similarity to (3.1).

Finally, it remains to be ensured that $\widetilde{\alpha}$ does not intersect the OFF-set. This follows from the fact that the smallest cube that contains a set of minterms (i.e. their supercube) is a subset of any other cube that contains those minterms. Thus, $\widetilde{\alpha}$ is guaranteed to be a subset of α, which itself does not intersect the OFF-set.

ESPRESSO-II computes maximally reduced cubes by using the aforementioned *unate reursive paradigm* (Section 3.1).

How might SAT be exploited to compute maximally reduced cubes? Formula (3.2) suggests one simple approach — find all the satisfying assignments of (3.2) using a SAT checker, and then compute their supercube

This approach, called MAXIMALLY-REDUCE-SIMPLE, is shown in Figure 3.5. The function SAT_CHECK() takes a CNF formula as its first parameter and determines whether it is SATISFIABLE. If so, a satisfying assignment is returned by modifying the second param-

```
MAXIMALLY-REDUCE-SIMPLE ($\alpha$, $F$, $F^{ON}$)
1 {
2         $\Phi \leftarrow$ toCNF $\left(\sum_{\gamma \in F^{ON}} \gamma\right)$
3         $\Psi \leftarrow \alpha \cdot \left(\prod_{\substack{\beta \in F \\ \beta \neq \alpha}} \overline{\beta}\right) \cdot \Phi$
4         $S \leftarrow \emptyset$
5         while (SAT_CHECK ($\Psi$, &$assignment$) == SATISFIABLE)
6         {
7                 $S \leftarrow S \cup assignment$
8                 $\Psi \leftarrow \Psi \cdot assignment'$
9         }
10         return SUPERCUBE($S$);
11 }
```

Figure 3.5: Simple method to compute maximally reduced cubes

eter (passed-by-pointer). The function SUPERCUBE computes and returns the supercube of its argument cubes. Cubes may be passed either together as a set or individually via the parameter list.

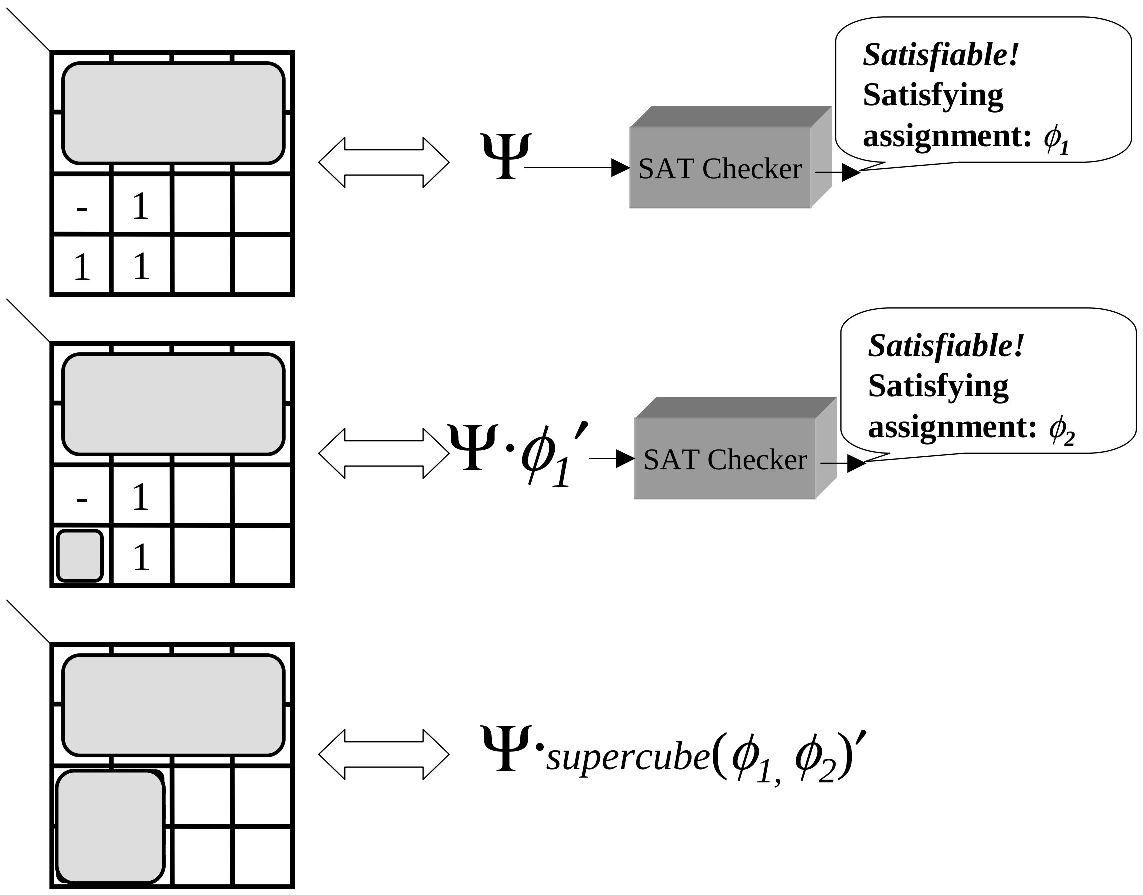
In each iteration of the while loop, we determine if Ψ is satisfiable. If it is, then satisfying assignment is returned via *assignment*, Ψ is modified to 'block' out *assignment* (line 8), and *assignment* is added to the collection S of assignments found so far (line 7). Continuing in this way, eventually all satisfying assignments are found, and we compute and return their supercube in line 10.

However, we can do exponentially faster! A modified algorithm (Figure 3.6) maintains a 'running total' supercube $\widetilde{\alpha}$ of all the assignments found so far, and blocks out each updated supercube $\widetilde{\alpha}$ from Ψ instead of each individual *assignment*. In the end the algorithm simply returns the running total so far.

To see the intuition behind this approach, see Figure 3.7.

```
MAXIMALLY-REDUCE-FASTER ($\alpha$, $F$, $F^{ON}$)
{
    $\Phi \leftarrow$ TOCNF $\left(\sum_{\gamma \in F^{ON}} \gamma\right)$
    $\Psi \leftarrow \alpha \cdot \left(\prod_{\substack{\beta \in F \\ \beta \neq \alpha}} \overline{\beta}\right) \cdot \Phi$
    $\widetilde{\alpha} \leftarrow 0$
    while (SAT_CHECK ($\Psi$, &$assignment$) == SATISFIABLE)
    {
        $\widetilde{\alpha} \leftarrow$ SUPERCUBE($\widetilde{\alpha}, assignment$)
        $\Psi \leftarrow \Psi \cdot \widetilde{\alpha}'$
    }
    return $\widetilde{\alpha}$;
}
```

Figure 3.6: Faster method to compute maximally reduced cubes



Satisfiable!
Satisfying assignment: ϕ_1
Ψ
SAT Checker
Satisfiable!
Satisfying assignment: ϕ_2
$\Psi \cdot \phi_1'$
SAT Checker
$\Psi \cdot supercube(\phi_1, \phi_2)'$
- 1
1 1
- 1
1

Figure 3.7: The new SAT-based REDUCE operator

3.4 Essentials

The ESSENTIALS operator is intended to simplify the minimization problem. Essential prime implicants must be present in any prime cover of the given function. Therefore, they can be identified at the outset so that the subsequent main loop of ESPRESSO-II only has to deal with non-essential primes.

Recall that for a prime implicant α of a function f to be essential, there must exist at least one *witness* ON-set minterm that is contained in α but not in any other prime implicant of f. Consequently, to compute the essentials of a function, it is sufficient to compute their witnesses. To get the actual essentials given the witnesses, we need only EXPAND each witness, removing duplicates from the result.

We note the following facts, which will help us identify witnesses (f denotes a given Boolean function):

Fact 3.4.1 *Every implicant of f is contained by some prime implicant of f (possibly itself).* This prime implicant need not be unique.

Fact 3.4.2 *An ON-set minterm e is a witness to essentiality if and only if there is exactly one prime implicant that contains it, namely the corresponding essential.* (Follows from definition of essential prime implicant.)

Suppose we have a minterm $e = l_1 l_2 \ldots l_n$. In n-dimensional Boolean space (i.e. in the Boolean n-cube), there are n minterms that are adjacent to it:

$$\overline{l_1} l_2 \ldots l_n, \qquad l_1 \overline{l_2} \ldots l_n, \qquad \ldots \qquad l_1 l_2 \ldots \overline{l_n}$$

each of which is obtained by negating one literal in e. Let us denote the set of minterms adjacent to e by $Adj(e)$. Then we define

$$\begin{aligned}
Adj^{ON}(e) &\equiv Adj(e) \cap \text{ON-set}(f) && \text{(ON-set minterms that are adjacent to } e\text{)}\\
Adj^{DC}(e) &\equiv Adj(e) \cap \text{DC-set}(f) && \text{(DC-set minterms that are adjacent to } e\text{)}\\
Adj^{OFF}(e) &\equiv Adj(e) \cap \text{OFF-set}(f) && \text{(OFF-set minterms that are adjacent to } e\text{)}
\end{aligned}$$

It holds that

$$Adj^{ON}(e),\ Adj^{DC}(e) \text{ and } Adj^{OFF}(e) \text{ are pairwise disjoint; and}$$
$$Adj^{ON}(e) \cup Adj^{DC}(e) \cup Adj^{OFF}(e) = Adj(e)$$

Given a cube α, we also define $Adj^{\{\alpha\}}(e)$ to be the set of all minterms contained in α that are adjacent to e. In this section, we denote the (uniquely defined) supercube of a set S of minterms by $supercube(S)$.

Fact 3.4.3 *If $A \subseteq B$, then $supercube(A) \subseteq supercube(B)$.*

Fact 3.4.4 *If α is a Boolean k-cube, e is a minterm in α, and then we can write*

$$\alpha = supercube(\{e\} \cup Adj^{\{\alpha\}}(e))$$

Since α has k dimensions, there are k minterms in α that are adjacent to e (i.e. $|Adj^{\{\alpha\}}(e)| = k$). Since the supercube under consideration must contain e and all of these k adjacent minterms, it itself must have dimensionality at least k. In fact, since α has k dimensions, the supercube must also have exactly k dimensions (by definition). Further, for the supercube to be uniquely defined, we must have that α is the supercube.

Fact 3.4.5 *Given an ON-set minterm e and a prime implicant α containing it, e is a witness of essentiality (with α its essential prime) if and only if all minterms in $Adj^{ON}(e) \cup Adj^{DC}(e)$ are contained in α.*

('only if'): Assume there is an $e' \in Adj^{ON}(e) \cup Adj^{DC}(e)$ that is *not* contained in α. Since Then e and e' are adjacent, they form a binary cube. Further, neither minterm is in the OFF-set. Hence, the set $\{e, e'\}$ represents an implicant, which we will denote as β for brevity. Let β' be a prime implicant containing β (Fact 3.4.1 guarantees the existence of β'). Then β' contains both e and e'. But this means β' is a prime implicant that contains e but is distinct from α (α doesn't contain e' by assumption). Hence, e is not a witness and α is not an essential.

('if'): Assume all minterms in $Adj^{ON}(e) \cup Adj^{DC}(e)$ are contained in α. Then we must have that $Adj^{ON}(e) \cup Adj^{DC}(e) \subseteq Adj^{\{\alpha\}}(e)$

Now let β be an arbitrary prime implicant containing e. Since β cannot intersect the OFF-set, every minterm in β must be either an ON-set minterm or a DC-set minterm. This implies that $Adj^{\{\beta\}}(e) \subseteq Adj^{ON}(e) \cup Adj^{DC}(e)$.

Hence we must have that $Adj^{\{\beta\}}(e) \subseteq Adj^{\{\alpha\}}(e)$. This in turn means that

$$\begin{aligned} \beta &= supercube(\{e\} \cup Adj^{\{\beta\}}(e)) \qquad \text{(Fact 3.4.4)} \\ &\subseteq supercube(\{e\} \cup Adj^{\{\alpha\}}(e)) \\ &= \alpha \end{aligned}$$

Thus, $\beta \subseteq \alpha$. But since β is also prime implicant and hence cannot be contained in any other implicant, this means that $\beta = \alpha$. Hence there is exactly one prime implicant that contains α, so by Fact 3.4.2, e is a witness to essentiality and α is an essential prime.

`[TO-DO for final draft:`
`- Complete characterization of witnesses as those (and only those) ON-set minterms` e `for whom` $\mathit{supercube}(\{e\} \cup \mathit{Adj}^{ON}(e) \cup \mathit{Adj}^{DC}(e))$ `does not intersect the OFF-set.`
`- Show how this lead to the procedure outlined below`
`- ''Encode'' that procedure into a single CNF formula]`

$\exists$ a minterm $l_1 l_2 \dots l_n$ such that following function gives 1:

1 must have $f(l_1 l_2 \dots l_n) = 1$

2 for each i in $1 \dots n$

2a Let $p_i = (l_1 l_2 ... l_{i-1} \overline{l_i} l_{i+1} \dots l_n \in$ some $\alpha \in F^{ON})$

3. Compute supercube with all $l_1 l_2 ... l_{i-1} \overline{l_i} l_{i+1} \dots l_n$ whose $p_i = 1$

4. Supercube must be covered by $F^{ON} \cup F^{DC}$

(Want to actually do 3 & 4 together)

- out optimization

- 3 levels of leniency or 'strength' of the notion of essentiality

Chapter 4

Experimental Results

`[For this draft only]`

Our final goal is to compare the overall runtimes of our approach with those of Espresso-II on large examples. However, due to a recently-discovered bug in a library function that we use, our implementation terminates abnormally on all the examples we have tried, so we are unable to report overall runtime on any example. We are working hard to resolve the issue. However, we have been able to conduct preliminary comparisons on an operator-by-operator basis. These partial results indicate that our approach is promising.

name	SAT-Espresso	Espresso-II
50x50	**Error!**	**Error!**
200x200	70.18	261.95

Figure 4.1: REDUCE (run-times reported in seconds). Here is where the bug first manifested itself

name	SAT-Espresso	Espresso-II
50x50	0.02	0.24
200x200	0.65	168.14

Figure 4.2: ESSENTIALS (run-times reported in seconds)

In our experiments we focus on the execution time required for the operators REDUCE, ESSENTIALS and IRREDUNDANT.

In our implementation, we reuse most of the code for Espresso-II, replacing only the operators REDUCE, ESSENTIALS and IRREDUNDANT, for simplicity of comparison. As

name	SAT-Espresso	Espresso-II
50x50	0.12	0.34
200x200	0.81	47.33

Figure 4.3: IRREDUNDANT (run-times reported in seconds)

our SAT checker we used a modified version of zChaff [14] from Princeton University. Our implementation involves a lot of redundant file I/O, so there is room for improvement in our performance.

Two examples are presented:

1. **'50x50'**: A `[five-output]` Boolean function over 50 variables. The input file was of type `fr`, meaning that it specified the ON-set and OFF-set, and the DC-set was assumed to be the complement of ON-set $\cup$ OFF-set. The initial cover had cardinality 50. This example was randomly generated.

2. **'200x200'**: A `[five-output]` Boolean function over 200 variables. The input file was randomly generated, of type `fr`, and specified an initial cover of cardinality 200.

Both examples were randomly generated under the restriction that the DC-set should constitute approx. 20% of the Boolean space.

All experiments were performed using a Linux workstation with an Intel processor. 'Espresso-II' denotes ESPRESSO-II and 'SAT-Espresso' denotes our implementation. `[Add more specifics about workstation environment.]`

Chapter 5

Exact Minimization Using QBF Checkers

In this section, we present techniques for solving the two-level logic minimization problem exactly by using QBF checkers. (QBF checkers are satisfiability testers for Quantified Boolean Formulae.)

Recall the classic QUINE-MCCLUSKEY algorithm used to solve this problem (presented in Chapter 2):

1. generate the set of all prime implicants;

2. *set covering problem:* select a minimum number of prime implicants such that each ON-set minterm is contained.

We follow the same basic approach. This means that we need to address two challenges:

1. Devising reasonably efficient ways of using QBF formulae to implicitly represent the set of prime implicants of a function.

2. Devising reasonably efficient ways of using the above technique along with SAT or QBF checkers to implicitly solve the set covering problem.

This chapter is organized around these two steps. We begin by introducing data structures to efficiently represent Boolean functions and sets of products.

5.1 Background on Decision Diagrams

This section introduces data structures to efficiently represent Boolean functions and sets of products, called *Binary Decision Diagrams* and *Zero-Suppressed Binary Decision Diagrams*.

These data structures have been used in Scherzo [4, 7, 5, 6], discussed previously, and will also be used in the new SAT-based exact two-level minimization techniques presented later in this chapter.

5.1.1 BDDs

Binary Decision Diagrams (BDDs) [1] are used to efficiently represent Boolean functions. An ordered BDD (OBDD) is a canonical representation of a function f which is obtained from the Shannon tree representation of f by reduction rules which (i) identify isomorphic subgraphs and (ii) delete each vertex that has the same left and right children.

Example 5.1.1 In Figure 5.1(a) the Shannon tree of the function $f = ab + c$ is shown. To find the function value for a specific assignment to the variables, one follows the path from the root node to a terminal node, taking the left (right) branch if the corresponding variable is assigned the value 0 (1). The corresponding BDD obtained by the above reduction rules is shown in part (b) of the figure. Note that the BDD of f is just a compact representation of the Shannon tree of f. In particular, the same algorithm can be used to evaluate the function for an assignment to the variables. □

Important properties of BDDs include canonicity of representation (if the variable ordering is fixed), and the efficiency of binary operators, e.g. the Boolean AND of two functions represented by BDDs can be efficiently computed in time proportional to the product of the number of nodes of the two BDDs.

5.1.2 ZBDDs

Zero-suppressed BDDs (ZBDDs) [13] are a variant of BDDs which were introduced to efficiently represent *sets of products*, e.g. the set of prime implicants of a function f. A ZBDD of a set of products is obtained from a tree representation of the set of products by reduction rules which (i) identify isomorphic subgraphs and (ii) delete each vertex whose right children points to 0 (i.e. the empty set). Note that to achieve small representations for sparse sets, the second reduction rule differs from the second reduction rule for BDDs. Another difference from BDDs is that a ZBDD makes decisions based on *literals* instead of *variables*.

Example 5.1.2 Consider Figure 5.1(c), which shows the tree representation of the given set of products $\{\overline{b}, \overline{a}, \overline{a}\overline{b}, a, a\overline{b}\}$. Here each path from the root node to a terminal 1 node corresponds to a product in the set. The product consists of those *literals* encountered on taking right branches on the path. Here, positive (negative) literals are denoted by a '+'

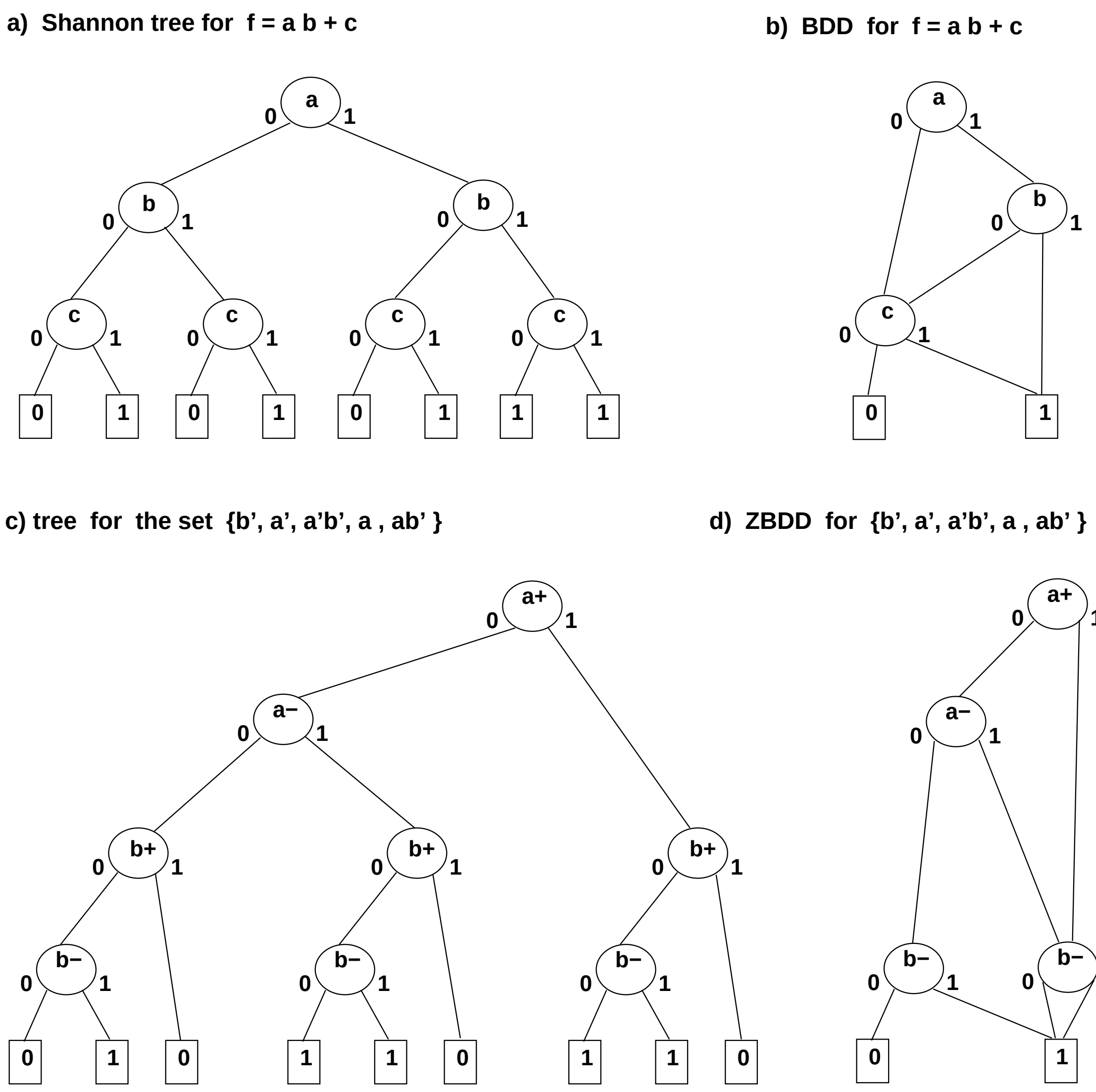

a) Shannon tree for f = a b + c
b) BDD for f = a b + c
c) tree for the set {b', a', a'b', a , ab' }
d) ZBDD for {b', a', a'b', a , ab' }
a
b
c
a+
a−
b+
b−
0
1

Figure 5.1: BDDs and ZBDDs

superscript (‘−’ superscript). The ZBDD for this set of products obtained by the above reduction rules is shown in part (d) of the figure. □

Important properties of ZBDDs include canonicity of representation and efficient computation of set-operations, such as union and intersection.

5.1.3 Application: Implicit Techniques

Based on Binary Decision Diagrams new efficient algorithms to solve standard problems in digital design have been developed. The main idea of *implicit* approaches is to use compact data structures such as BDDs and ZBDDs to eliminate the need to explicitly represent each object of interest (e.g. prime implicants, minterms). Such implicit approaches typically can

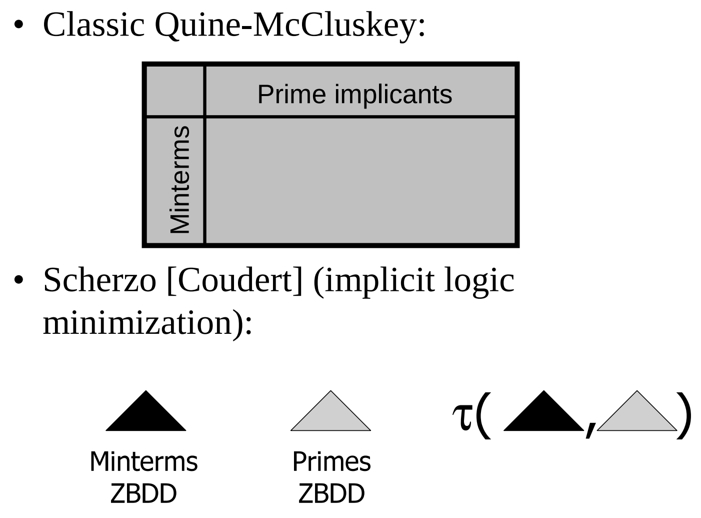

• Classic Quine-McCluskey:
Prime implicants
Minterms
• Scherzo [Coudert] (implicit logic minimization):
Minterms ZBDD
Primes ZBDD
τ(,)

Figure 5.2: Classic vs. Implicit Logic Minimization

solve much more complex problems than previous methods.

In fact, the use of implicit minimization techniques made Scherzo [4, 7, 5, 6] 10 to more than 100 times faster than the best previous exact two-level logic minimization methods. Scherzo has solved examples with 10^{20} prime implicants, which are by far out of the reach of classic minimization algorithms like the well-known Quine-McCluskey algorithm.

The new concepts of Scherzo are as follows:

- Scherzo uses data structures like BDDs [1] and ZBDDs [13] to represent Boolean functions and sets of products very efficiently. For example, consider how Scherzo computes the set of prime implicants for a Boolean function f. It first computes a BDD for the Boolean function f from an initial unoptimized cover (e.g. from the set of ON-set minterms). Then, it uses the BDD representation of f to directly generate a ZBDD that represents the set of prime implicants of f. The computation time to generate the ZBDD as well as the size of the ZBDD are both independent of the number of prime implicants. As mentioned before, this algorithm has been used to compute ZBDDs that represent up to 10^{20} prime implicants.

- Scherzo includes new algorithms that operate on these implicit data structures. For example, classic techniques are based on a 'covering' matrix where rows are labeled by the minterms (refer to Figure 5.2) and columns are labeled by the prime implicants. In contrast, Scherzo operates on two much more compact ZBDDs: one for the minterms and one for the prime implicants.

5.2 Implicitly Representing Prime Implicants

Actually, we can get by even if we can just represent the set of all implicants of a function f by an ordinary Boolean formula, say $\mathbf{I}(f)$.

$\mathbf{I}(f)$ is constructed so that there is a 1-to-1 and onto mapping between implicants of f, and satisfying instances of $\mathbf{I}(f)$.

There are several ways of expressing $\mathbf{I}(f)$. Here we focus on an approach to generating $\mathbf{I}(f)$ based on the BDD for f. One feature of this approach is that the structures of generated formulae are closely related to the structures of the corresponding BDDs.

$\mathbf{I}(f)$ is a Boolean formula over l_{x_1}, $l_{\overline{x_1}}$, l_{x_2}, $l_{\overline{x_2}}$, … , l_{x_n}, $l_{\overline{x_n}}$. Each assignment to the $2n$ variables corresponds to a product term such that: $\forall i$, $1 \leq i \leq n$, the positive (or negative) half-space of variable x_i is present in the product if and only if l_{x_i} (or $l_{\overline{x_i}}$) is true. If f_{x_1} and $f_{\overline{x_1}}$ are the cofactors of f wrt x_1, then we can see:

$$\mathbf{I}(f) \longleftrightarrow \Big[\Big(l_{x_1} + l_{\overline{x_1}}\Big)\Big(l_{\overline{x_1}} \rightarrow \mathbf{I}(f_{\overline{x_1}})\Big)\Big(l_{x_1} \rightarrow \mathbf{I}(f_{x_1})\Big)\Big] \tag{5.1}$$

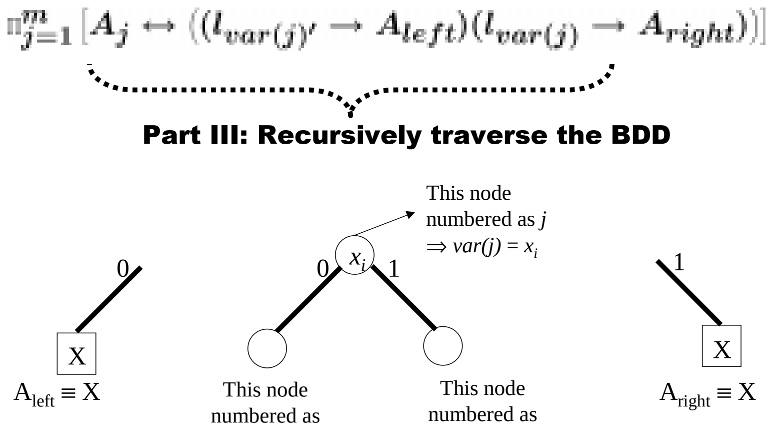
NOTE: $\mathbf{I}(f_{x_1})$ and $\mathbf{I}(f_{\overline{x_1}})$ are Boolean formulae over l_{x_2}, $l_{\overline{x_2}}$, … , l_{x_n}, $l_{\overline{x_n}}$.

This immediately suggests the following approach:

- Start with the BDD of f, where f is a Boolean function over n variables x_1, x_2, … , x_n.
- Uniquely number the BDD nodes (suppose there are m of them).
- Let $var(j)$ denote the variable associated with node j. Then construct the desired formula $\mathbf{I}(f)$ over
  - the $2n$ variables l_{x_1}, $l_{\overline{x_1}}$, l_{x_2}, $l_{\overline{x_2}}$, … , l_{x_n}, $l_{\overline{x_n}}$; and
  - m "implied" variables A_1, A_2, … A_m (one for each BDD node; we will shortly see why they are "implied")

  as follows:

$$\begin{aligned}\mathbf{I}(f) \;=\; & A_1 \\ \cdot\; & \prod_{i=1}^{n}(l_{x_i} + l_{\overline{x_i}}) \\ \cdot\; & \prod_{j=1}^{m}\Big[A_j \leftrightarrow \Big(\big(l_{var(j)'} \rightarrow A_{left}\big)\big(l_{var(j)} \rightarrow A_{right}\big)\Big)\Big]\end{aligned}$$


$\prod_{j=1}^{m}\left[A_j \leftrightarrow \left((l_{var(j)'} \rightarrow A_{left})(l_{var(j)} \rightarrow A_{right})\right)\right]$
Part III: Recursively traverse the BDD
This node numbered as j $\Rightarrow var(j) = x_i$
x_i
0
1
0
1
X
$A_{left} \equiv X$
This node numbered as
This node numbered as
X
$A_{right} \equiv X$

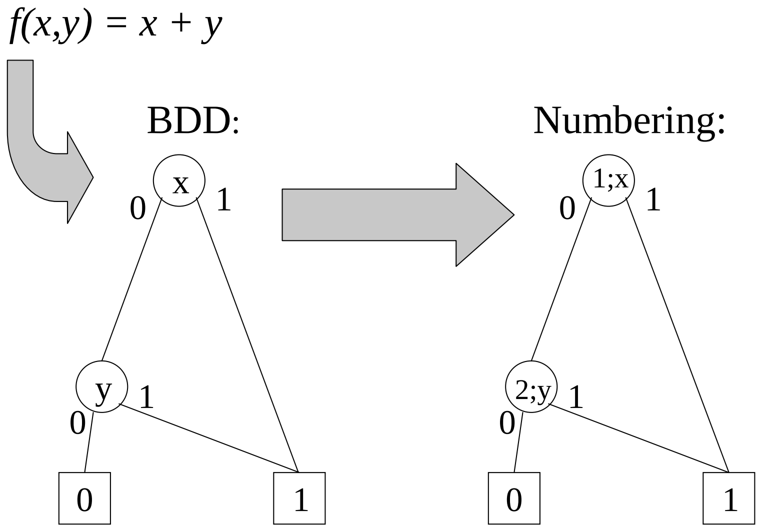


Figure 5.4: Constructing $\mathbf{I}(f)$ for $f(x,y) = x + y$

In the above formula, A_1 corresponds to the root node. $\prod_{i=1}^{n}(l_{x_i} + l_{\overline{x_i}})$ ensures that no variable appears more than once in the implicants characterized by $\mathbf{I}(f)$. Finally, each $\Big[A_j \leftrightarrow \Big(\big(l_{var(j)'} \rightarrow A_{left}\big)\big(l_{var(j)} \rightarrow A_{right}\big)\Big)\Big]$ 'fixes' the value of A_j for some BDD node numbered j. It is built by recursively traversing the BDD of f as shown in Figure 5.3.

As an example, consider the function $f(x,y) = x + y$. The BDD is represented and numbered as shown in Figure 5.4.

$$
\begin{aligned}
\text{Here,} \quad \mathbf{I}(f) \;&=\; A_1 \cdot (l_x + l_{\overline{x}}) \cdot (l_y + l_{\overline{y}}) \\
&\cdot \;\; \big[A_1 \leftrightarrow \big((l_{\overline{x}} \rightarrow A_2)(l_x \rightarrow 1)\big)\big] \\
&\cdot \;\; \big[A_2 \leftrightarrow \big((l_{\overline{y}} \rightarrow 0)(l_y \rightarrow 1)\big)\big]
\end{aligned}
$$

Satisfying instances $(l_{\overline{x}}, l_x, , l_{\overline{y}}, l_y, A_1, A_2)$: $\{(0,1,0,1,1,1)$, $(0,1,1,0,1,0)$, $(0,1,1,1,1,0)$, $(1,0,0,1,1,1)$, $(1,1,0,1,1,1)\}$

These correspond exactly to the implicants of f: $\{xy,\ x\overline{y},\ x,\ \overline{x}y,\ y\}$

5.3 Implicitly Solving the Set Covering Problem

We want to produce a Boolean formula $\Psi(f,i)$ that is satisfiable if and only if there is a set S of implicants of f, $(|S| \leq i)$, such that $\sum_{p \in S} p = f$.

E.g. $\Psi(f,3)$ should express:

$$\exists p_1 \exists p_2 \exists p_3 \text{ such that } \Big[(p_1, p_2, p_3 \text{ are implicants of } f) \text{ AND } (p_1 + p_2 + p_3 = f)\Big] \qquad (5.2)$$

so a likely form would be:

$$\begin{aligned}
&\exists l_{1,x_1} \exists l_{1,\overline{x_1}} \ldots \exists l_{1,x_n} \exists l_{1,\overline{x_n}} \\
&\exists l_{2,x_1} \exists l_{2,\overline{x_1}} \ldots \exists l_{2,x_n} \exists l_{2,\overline{x_n}} \\
&\exists l_{3,x_1} \exists l_{3,\overline{x_1}} \ldots \exists l_{3,x_n} \exists l_{3,\overline{x_n}} \\
&\quad [\ \ \mathbf{I_1}(f) \cdot \mathbf{I_2}(f) \cdot \mathbf{I_3}(f) \\
&\quad \cdot\ \ \forall x_1 \ldots \forall x_n (f \rightarrow \mathbf{p_1} + \mathbf{p_2} + \mathbf{p_3})\ ]
\end{aligned}$$

`[to add - description of formula]`

$$\text{where each } \mathbf{p_k} = \prod_{i=1}^{n} \Big[(l_{k,\overline{x_i}} + x_i)(l_{k,x_i} + \overline{x_i})\Big] \qquad (5.3)$$

Given an assignment of values to the $l_{k,\overline{x_i}}$'s and l_{k,x_i}'s, $\mathbf{p_k}$ gives the corresponding product term. E.g. If f is over two variables x and y, and $(l_{\overline{x}}, l_x, , l_{\overline{y}}, l_y) = (0,1,1,0)$, then

$$\begin{aligned}
\mathbf{p_1} &= (l_{1,\overline{x}} + x)(l_{1,x} + \overline{x})(l_{1,\overline{y}} + y)(l_{1,y} + \overline{y}) \\
&= (0 + x)(1 + \overline{x})(1 + y)(0 + \overline{y}) \\
&= x\overline{y}
\end{aligned}$$

Now, using a SAT checker, we keep testing the satisfiability of $\Psi(f,i)$ for $i = 1, 2, \ldots$ until we find an i that makes $\Psi(f,i)$ satisfiable. Any satisfying instance corresponding to this i gives a minimal sum-of products (SOP) representing f. Each of the products in the minimal SOP is given by one of the i copies of l_{x_1}, $l_{\overline{x_1}}$, l_{x_2}, $l_{\overline{x_2}}$, ... , l_{x_n}, $l_{\overline{x_n}}$.

Chapter 6

Conclusion

With regard to heuristic minimization: We have observed that state-of-the-art heuristic minimizers produce high-quality approximate solutions, but are computationally expensive on large examples. We have presented an approach to efficiently achieve high-quality approximations on large problems, by combining the strengths of ESPRESSO-II (quality of approximation) and SAT checkers (speed on large problems). Preliminary experiments show promising results.

With regard to exact minimization: We have observed that traditional exact minimizers suffer from the problem of explicit representation. This can be computationally expensive when there are an exponential number of prime implicants and minterms that need to be explicitly represented in memory. State-of-the-art exact minimizers use implicit techniques based on BDDs to tackle this problem. We have presented another kind of implicit approach, based on SAT and QBF (Quantified Boolean Formula) checkers.

The techniques presented in this documents can be explored in other interesting contexts, such as:

- QBF checking
- hardware verification
- software verification

`[TO-DO: Expand]`

Bibliography

[1] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[2] R. K. Brayton et al. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic, 1984.

[3] E. M. Clarke, P. Chauhan, S. Sapra, J. Kukula, H. Veith, D. Wang. Automated Abstraction Refinement for Model Checking Large State Spaces using SAT based Conflict Analysis. In *Proceedings of FMCAD'02*, pp. 33-51, 2002.

[4] O. Coudert. Two-level logic minimization: an overview. *Integration, the VLSI journal*, 17:97–140, 1994.

[5] O. Coudert. Doing two-level logic minimization 100 times faster. In *ACM-SIAM Symposium on Discrete Algorithms*, 1995.

[6] O. Coudert. On solving covering problems. In *Proceedings of the 33rd Design Automation Conference.* ACM, 1996.

[7] O. Coudert and J.C. Madre. New ideas for solving covering problems. In *Proceedings of the 32nd Design Automation Conference.* ACM, 1995.

[8] Giovanni De Micheli. *Synthesis And Optimization Of Digital Circuits.* McGraw-Hill, 1994.

[9] Srinivas Devadas, Abhijit Ghosh, and Kurt Keutzer. *Logic Synthesis.* McGraw-Hill, 1994.

[10] J. Hlavicka and P. Fiser.BOOM - A Heuristic Boolean Minimizer. In *Proceedings of the International Conference on Computer Aided Design (ICCAD'01)*, pp. 439-442, 2001.

[11] E.J. McCluskey. *Logic Design Principles.* Prentice-Hall, 1986.

[12] K. L. McMillan. Applying SAT methods in Unbounded Symbolic Model Checking. In *Proceedings of CAV'02*, pp. 250-264, 2002.

[13] S. Minato. Zero-Suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th Design Automation Conference*. ACM, 1993.

[14] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference (DAC'01)*, pp. 530-535, 2001.

[15] R. Rudell and A. Sangiovanni Vincentelli. Multiple valued minimization for PLA optimization. *IEEE Transactions on CAD*, CAD-6(5):727–750, September 1987.

[16] Richard Rudell. Logic synthesis for VLSI design. Technical Report UCB/ERL M89/49, Berkeley, 1989.

[17] J. P. Marques Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. Technical Report CSE-TR-292-96, Computer Science and Engineering Division, Department of EECS, Univ. of Michigan, April 1996.

[18] Michael Theobald. *Efficient Algorithms for the Design of Asynchronous Control Circuits*. PhD thesis, Columbia University, 2002.

[19] C. Umans. The Minimum Equivalant DNF Problem and Shortest Implicants. *Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 556-563, 1998.

[20] Hantao Zhang. SATO: An efficient propositional prover. In *Proceedings of the Conference on Automated Deduction (CADE'97)*, pp. 272-275, 1997.