AN ALGORITHM FOR THE PLA EQUIVALENCE PROBLEM

DISSERTATION

Presented to the Graduate Council of the

University of North Texas in Partial

Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

By

Gyo Sik Moon, B.E., M.S.

Denton, Texas

December, 1995

AN ALGORITHM FOR THE PLA EQUIVALENCE PROBLEM

DISSERTATION

Presented to the Graduate Council of the
University of North Texas in Partial
Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

By

Gyo Sik Moon, B.E., M.S.

Denton, Texas

December, 1995

Moon, Gyo Sik, An Algorithm for the PLA Equivalence Problem. Doctor of Philosophy (Computer Science), December, 1995, 118 pp., 24 tables, references, 157 titles.

The *Programmable Logic Array* (PLA) has been widely used in the design of VLSI circuits and systems because of its regularity, flexibility, and simplicity. The equivalence problem is typically to verify that the final description of a circuit is functionally equivalent to its initial description. Verifying the functional equivalence of two descriptions is equivalent to proving their *logical equivalence.* This problem of pure logic is essential to circuit design. The most widely used technique to solve the problem is based on *Binary Decision Diagram* or BDD, proposed by Bryant in 1986. Unfortunately, BDD requires too much time and space to represent moderately large circuits for equivalence testing.

We design and implement a new algorithm called the Cover-Merge Algorithm for the equivalence problem based on a divide-and-conquer strategy using the concept of *cover* and a *derivational method.* We prove that the algorithm is sound and complete. Because of the NP-completeness of the problem, we emphasize simplifications to reduce the search space or to avoid redundant computations. Simplification techniques are incorporated into the algorithm as an essential part to speed up the the derivation process. Two different sets of heuristics are developed for two opposite goals: one for the proof of equivalence and the other for its disproof. Experiments on a large scale of data have shown that big speed-ups can be achieved by prioritizing the heuristics and by choosing the most favorable one at each iteration of the Algorithm.

Results are compared with those for BDD on standard benchmark problems as well as on random PLAs to perform an unbiased way of testing algorithms. It has

been shown that the Cover-Merge Algorithm outperforms BDD in nearly all problem instances in terms of time and space. The algorithm has demonstrated fairly stabilized and practical performances especially for big PLAs under a wide range of conditions, while BDD shows poor performance because of its memory greedy representation scheme without adequate simplification.

ACKNOWLEDGMENTS

I would like to thank my advisor, Frank Vlach, for his advice, encouragement and support during my years at the University of North Texas. He motivated me to look at research topics in logic and verification area. Without his patient guidance and advice, this work would not have been possible.

I also thank my other committee members, Chao-Chih Yang, Roy Thomas Jacob, and Benedict Wong, for their comments and suggestions.

I am very grateful to my wife, Young-Sook, for her endurance. My deepest gratitude goes to my mother in Korea who always has given me love and support sacrificially, and to my father in memory.

TABLE OF CONTENTS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Preliminary

We define terminologies and notations which will be used throughout this dissertation.

We use truth-functional propositional connectives as usual (Table 1):

Connectives	*Symbols*
logical-and	'$\wedge$' or a dot or the absence of a connective
logical-or	'$\vee$' or '$+$'
logical-not	'$\sim$' or a prime or a bar
exclusive-or	'*exor*' or '*xor*' or '$\oplus$'
logical-implication	'$\rightarrow$' or '$\Rightarrow$'
logical-equivalence	'$\Leftrightarrow$' or '$\equiv$'

Table 1: Truth-Functional Logical Connectives

Logical-and (logical-or, logical-not, logical-implication) is also called *conjunction* (*disjunction*, *negation*, *material implication*), respectively. The first three connectives are adequate for expressing any truth function over the Boolean logic.[1] However, other connectives are useful in describing logical statements succinctly. Boolean expressions can be described by standard form in the propositional logic or LISP expression form. For example, the Boolean expression $(\phi \wedge \psi) \vee (\bar{\phi} \wedge \psi)$ can also be expressed in a LISP

[1]Other sets of connectives are also adequate. For example, the set $\{\sim, \Rightarrow\}$ is adequate for expressing any truth function [70].

form (**or** (**and** ϕ ψ) (**and** (**not** ϕ) ψ)).

Truth and falsity are called *truth values* and nothing else is a truth value. '*T*' or '*True*' or '*true*' denotes the truth value truth. '*F*' or '*False*' or '*false*' denotes the truth value falsity. Sometimes we use '0' for falsity and '1' for truth. We define *individual variables* as $x_1, \cdots, x_n$ each of which value is either true or false. A *literal* is a variable or the negation of a variable. A *clause* (*term*) is a disjunction (conjunction) of literals. A formula (or expression) F is said to be in a *conjunctive normal form* (CNF) (*disjunctive normal form* (DNF)) if and only if F has the form $F_1 \wedge \cdots \wedge F_m$ $(F_1 \vee \cdots \vee F_m), m \geq 1$, where each of $F_1, \cdots, F_m$ is a disjunction (conjunction) of literals. A *maxterm* (*minterm*) is a disjunction (conjunction) of n variables with each variable being negated or non-negated.

Hereafter we abbreviate 'if and only if' to 'iff'.

A formula (or Boolean expression) is said to be a *tautology* iff it is true under all its interpretations. A formula is said to be a *nontautology* iff it is not a tautology. A formula is said to be *unsatisfiable* iff it is false under all its interpretations. A formula is said to be *satisfiable* iff it is not unsatisfiable. Two formulas F and G are said to be *equivalent*, denoted by $F \equiv G$ or $F \Leftrightarrow G$, iff the truth values of F and G are the same under every interpretation of F and G.

A *programmable logic array* (PLA) [48, 66, 101] is a class of two-level logic circuit, which provides an efficient way of synthesizing arbitrary combinational as well as sequential logic circuits. Because of their regularity, flexibility, and simplicity, PLAs have been extensively used in the design of VLSI circuits and systems. A PLA typically consists of AND and OR planes which are initially fabricated with links among them. The physical structure of a PLA is a two-dimensional array of simple switching elements. Specific Boolean functions can be realized in sum of products form by appropriately connecting the switching elements. Major disadvantage of PLAs is that most PLAs use only small portion of switching elements and leave the rest unused, resulting in a significant waste of silicon area. So, reducing the area complexity of PLAs is a challenging research area. Recently, different types of PLAs have been proposed to enhance testability and area complexity of PLAs [13, 127, 130].

The *cofactor* of a function f with respect to a variable x is the function resulting from replacing x by a constant b, denoted by $f|_{(x=b)}$. Then, for variables $x_1, \cdots, x_n$, $f|_{(x_i=b)}(x_1, \cdots, x_n) = f(x_1, \cdots, x_{i-1}, b, x_{i+1}, \cdots, x_n)$. The *Shannon expansion* [129] of a function around variable x_i is given by: $f = x_i \cdot f|_{(x_i=1)} + \bar{x}_i \cdot f|_{(x_i=0)}$. The *composition* of two functions f and g is the function resulting from replacing x_i of f by g, denoted by $f|_{(x_i=g)}$. That is, $f|_{(x_i=g)}(x_1, \cdots, x_n) = f(x_1, \cdots, x_{i-1}, g(x_1, \cdots, x_n), x_{i+1}, \cdots, x_n)$.

1.2 Outline of the Dissertation

Verification is an essential part of any logic design domain such as logic minimization, synthesis, etc [13, 71, 121, 128, 150, 156]. The main purpose of verification is to certify the *logical equivalence* of two logic circuits. Verification for a logic design is typically used to prove that the final description of a circuit is logically equivalent to its initial description. It also may take place at any stage of the design process to make sure that any structural changes, mainly caused by a logic minimizer or synthesizer, have not altered the meaning of the initial description.

Simulation [12, 17, 16, 74] has been extensively used as the standard method for verification of logic circuits. As the size of the computer logic grows bigger, the technique experiences severe drawbacks. Obviously, the number of tests required to perform a satisfactory coverage grows exponentially with the number of input variables. Research is still going on in this area to get a better coverage of tests using cost-effective simulation techniques. However, *formal methods* [42, 79, 100, 138, 154] as opposed to simulation are getting more attention from the community.

In Chapter 2, we formally describe the equivalence problem and review related works that have been developed in this area so far. The equivalence testing of two PLAs is to ascertain that the two are logically equivalent. To do that, the traditional and most widely used approach is to prove that either

(1) $\phi \equiv \psi$ or

(2) $\phi \oplus \psi \equiv \mathit{false}$

is a tautology for any two Boolean expressions ϕ and ψ in *disjunctive normal form* or DNF. Thus, the equivalence problem is now translated into a tautology checking problem, which is a well-known NP-complete problem [60]. Almost all algorithms for equivalence testing use the same interpretation as stated above, taking both ϕ and ψ as input to a tautology checker.

In Chapter 3, we introduce a new approach based on a *divide-and-conquer* strategy using the concept of *cover* and a *derivational method* for the equivalence problem and investigate its theoretical aspects. Instead of taking whole expressions into consideration, we divide the problem into as many smaller problems as the number of terms of each expression by translating the notion of logical equivalence into the notion of *cover* (Definition 3.3):

> Define a PLA as the set of terms occurred in a Boolean expression in DNF. Let ϕ and ψ be two PLAs ($|\phi|, |\psi| \geq 1$). ϕ *covers* ψ, denoted by $\psi \in^c \phi$, iff for each term $\beta \in \psi$, any assignment that satisfies β also satisfies a disjunction of some terms of ϕ. □

The concept of cover is the theoretical basis for the divide-and-conquer strategy for the equivalence problem:

$\phi \equiv \psi$

iff $\psi \in^c \phi$ and $\phi \in^c \psi$

iff each term $\alpha \in \phi$ is covered by some terms of ψ and each term $\beta \in \psi$ is covered by some terms of ϕ. □

The above statement is justified by Theorem 3.1. Thus, the original problem is transformed into a series of relatively small cover testing problems. Because of the difficulty of the problem, we emphasize simplification techniques in order to reduce the search space or problem size. This simple divide-and-conquer strategy would enable more simplifications than the other approach, resulting in more reduction of the search space. The overall performance of equivalence testing is directly determined by the efficiency of each individual cover testing, which might be a hard problem in general although each subproblem would be smaller than the original.

Let ϕ be a PLA as defined in Definition 3.3 and β be a term. Then, the cover testing problem is to decide whether or not $\beta \in^c \phi$. Simplifications can be achieved by removing each term disjoint (Definition 3.4) with β from ϕ, preserving the notion of cover:

If $\Phi_1 = \phi - \{\alpha \,|\, \alpha \in \phi, \alpha \text{ is disjoint with } \beta\}$,

then $\beta \in^c \phi$ iff $\beta \in^c \Phi_1$. □

The above simplification is computationally cheap and justified by Theorem 4.4. The problem size may be further reduced by performing the projection operation on Φ_1. The projection operation is defined as follows:

Let α and β be two terms.

Let $\alpha[i]$ be the value of a variable x_i such that

$$\alpha[i] \ (1 \leq i \leq n) = \begin{cases} 1 & : \ x_i \text{ occurs in } \alpha \\ 0 & : \ \bar{x}_i \text{ occurs in } \alpha \\ d & : \ \text{neither } \bar{x}_i \text{ nor } x_i \text{ occurs in } \alpha. \end{cases}$$

The *projection* of α with respect to β, denoted by $\alpha^{\Pi}|_{\beta}$, is $\alpha^{\Pi}[j]|_{\beta} = \alpha[i]$, where i is the jth position of β whose value is d.

The *projection* of a set of terms Γ with respect to β, denoted by $\Gamma^{\Pi}|_{\beta}$, is the set of projections $\{\alpha_1^{\Pi}|_{\beta}, \cdots, \alpha_m^{\Pi}|_{\beta}\}$, where $\alpha_i^{\Pi}|_{\beta}$ is the projection of α_i with respect to β for each $\alpha_i(\neq \beta) \in \Gamma$. We abbreviate $\beta^{\Pi}|_{\beta}$ to β^{Π}. $\Box$

And it is shown that $\Phi_1^{\Pi}|_{\beta}$ can be extracted from Φ_1 in $O(mn)$ time, where $m = |\Phi_1|$, n is the number of input variables. By performing simplification based on projection, a simplified set Φ_2 may be obtained from Φ_1, which is justified by Theorems 3.3 and 4.4:

Let Φ_2 be $\Phi_1^{\Pi}|_{\beta}$.

Then, $\beta \in^c \Phi_1$ iff $\beta^{\Pi} \in^c \Phi_2$. $\Box$

The concepts of *minimal cover* (Definition 3.10) and *uniqueness* (Definition 3.12) are the theoretical basis for simplification based on projection. The efficiency of projection reduction is directly determined by the number of literals of β. So, the more literals in β, the more simplifications would take place. Obviously, the operation always simplifies Φ_1 because β normally contains at least one literal. Since simplification based on projection is performed throughout the whole input set of terms, it often leads to big simplifications. The cover testing problem has now been reduced to the problem of testing whether or not $\beta^{\Pi} \in^c \Phi_2$.

We also investigate the theoretical aspects which can be observed from the derivational method: *minimal cover*, *uniqueness*, *mergeability*, and *losslessness*. Minimal cover and uniqueness essentially concern redundancies regarding a cover and essentiality of terms as well as minterms with respect to a cover. Mergeability shows the sufficient condition for the existence of merges for a set of terms. Losslessness addresses the issues of the side-effects caused by the derivation process concerning lost minterms and their recovery. We establish interrelationships among these concepts and develop simplification techniques based on these properties. Theorems developed in Chapter 3 are used to prove the correctness of the equivalence testing algorithm described in Chapter 4.

In Chapter 4, we present the Algorithm Cover-Merge which is the top level algorithm realizing the divide-and-conquer method for equivalence testing described in Theorem 3.1. Theorem 4.5 shows the validity of the Algorithm Cover-Merge. The top level algorithm performs a series of subproblems each of which is to solve a cover testing problem. Cover testing is performed by the Algorithm Cover which essentially invokes the Algorithm Merge-Loop for each cover testing. The Algorithm Cover is validated by Theorem 4.4.

The purpose of the Merge-Loop is to prove $\gamma \in^c \Gamma$ for a PLA Γ and a term γ, using the derivational method (Definition 3.8):

Δ is called a *derivation* of a term γ from a set of terms Γ iff

(1) Δ is a finite sequence of terms.

(2) the last term in Δ is γ.

(3) each term of Δ is either a member of Γ or an immediate consequence (Definition 3.6) by the Merge Rule (Definition 3.5) of two terms preceding it in the sequence. $\Box$

Let $\Gamma \mapsto \gamma$ denote a derivation of γ from Γ. $\Gamma \not\mapsto \gamma$ denotes that it is not the case that $\Gamma \mapsto \gamma$. The Algorithm Merge_Loop_1 realizes the paradigm of the Merge-Loop, which is proved to be sound and complete (Theorem 4.1):

Let Γ be a PLA and γ be a term. For any two terms α and β, β is said to be a *generalization* of α iff β is the result of replacing zero or more 0's and 1's in α by d's.

There is a derivation of $\tilde{\gamma}$ from Γ by the Merge_Loop_1 iff $\gamma \in^c \Gamma$, where $\tilde{\gamma}$ is a generalization of γ. $\Box$

Thus, we establish,

$\beta^{\Pi} \in^{c} \Phi_2$ iff the Merge_Loop_1 returns True. □

Although the Merge_Loop_1 is logically bound to give correct answers, it may contain many redundancies and unnecessary operations. So, it would not be practical unless simplifications take place adequately in order to remove such redundancies so that a combinatorial explosion could be avoided as much as possible. Because of the NP-completeness of the problem, we do not expect that this algorithm or *any* algorithm can always avoid such an explosion in terms of time or space (or both) on every instance of cover testing problems. This strongly indicates that simplification techniques or whatever methods that can reduce the search space should be extensively used as an essential part of the Merge-Loop. The Algorithm Merge_Loop_2 implements simplification techniques and heuristics developed in this direction. The Merge_Loop_2 greatly improves the performance of the Merge-Loop in terms of the length of derivation of β^{Π} from Φ_2 by incorporating simplification techniques and heuristics into the algorithm. Theorem 4.3 substantiates the validity of the Merge_Loop_2:

$\beta^{\Pi} \in^{c} \Phi_2$ iff the Merge_Loop_2 returns True. □

Simplification is the most important part of the algorithm as far as running time is concerned. Three different types of simplifications are extensively used for the Merge_Loop_2: *covered term reduction*, *nonessential term reduction*, and *nonmerge reduction*. Simplifications during the Merge-Loop can be done by excluding *degenerating merges* (Definition 4.2), justified by Theorem 4.1, and by removing covered terms due to *forward merges* (Definition 4.1). Nonessential term reduction removes nonessential terms with respect to a *nonredundant derivation* (Definition 4.4), justified by Theorem 4.2. Removal of nonmergeable terms is justified by Theorem 3.6. These simplification rules preserve the notion of cover.

Heuristics based on types of merges are developed to speed up the derivation process by allowing simplifications or by eliminating unnecessary operations. They are ordered by priorities so that the most favorable heuristic might be selected at each iteration of the algorithm. Heuristics are designed and implemented such that they can be performed within a reasonable amount of time. Experiments show that these heuristics are the main cause of big speed-ups.

Suppose $\beta^{\Pi} \notin^{c} \Phi_2$ for a term β^{Π}. Because the basic structure of the algorithm is designed to prove $\beta^{\Pi} \in^{c} \Phi_2$ by showing the existence of a derivation $\Phi_2 \mapsto \beta^{\Pi}$, the Merge-Loop will do an exhaustive search for β^{Π} until every possibility is exhausted. It may not be always possible to avoid this undesirable situation because of the nature of the derivation process. However, the situation may be improved considerably by introducing a heuristic which is specially designated to dynamically predict the outcome of a derivation, based on the following observation: A nonredundant derivation of β^{Π} can be viewed as the process of replacing 0's or 1's by d's (i.e. reducing the number of literals) of merges along the sequence of a derivation. However, a certain type of merges called *backward merges* (Definition 4.1) always increases the number of literals along a derivation sequence as the result of performing such merges. The heuristic H7 (Section 4.2) says that if only backward merges are possible for a set of terms Φ_2, then it is most likely that $\beta^{\Pi} \notin^{c} \Phi_2$. If the condition for the heuristic is met, then $\beta^{\Pi} \notin^{c} \Phi_2$ is assumed and falsification heuristics are initiated to search a minterm of β^{Π} which would not be covered by Φ_2. Prediction is made at each iteration of the Merge-Loop so that better chances of detection can be obtained. Experiments show that almost all cases have been successfully detected within a few iterations of the Merge-Loop.

Since the problem of proving that $\beta^{\Pi} \in^{c} \Phi_2$ and the problem of disproving it are equally hard in general, any technique for disproof is not expected to be *complete* because otherwise it might spend too much time on some unfavorable cases. Instead, we invent *falsification heuristics* which run in $O(nm^2)$ time, where n is the number of variables and m is the number of terms of Φ_2. The main objective of the heuristics is to produce a counterexample by constructing a minterm of β^{Π} which is not covered by Φ_2. These heuristics are based on greedy methods which would seek to maximize the number of terms falsified by the assignment made at each step of the falsification process. Empirical results show that nearly all cases have been successfully falsified by the heuristics.

In Chapter 5, we prepare and test a wide variety of parameterized random sets of problem instances to give an unbiased way of testing the algorithms and techniques developed so far. We address various issues of constructing randomized problem instances and analyze their probabilistic properties.

Experimental results follow in Chapter 6. Experimental results on a wide variety of circuits show that the heuristics and techniques presented in Chapter 4 are found to be quite effective in simplification and reduction of proof procedures, achieving big speed-ups in most cases. In comparison with the most widely used technique called *Binary Decision Diagram* or BDD, the algorithm proposed outperforms BDD in nearly all cases of input circuits including standard benchmark problems. Finally, concluding remarks are found in Chapter 7.

CHAPTER 2

THE EQUIVALENCE PROBLEM AND RELATED WORK

In this chapter, we first discuss the equivalence problem in more precise terms and then we take other closely related problems into consideration. A review of related works points out important concepts and features that have been developed so far.

2.1 Introduction

The main purpose of the equivalence problem is to verify that two Boolean expressions have the same functionality. In other words, the problem is to decide, given two Boolean expressions ϕ_1 and ϕ_2, whether ϕ_1 and ϕ_2 are logically equivalent. The meaning of the problem is to check whether or not each set of assignments which satisfies ϕ_1 also satisfies ϕ_2 and vice versa. Exhaustive search by enumerating all such satisfying instances cannot be a solution because it always requires an exponential computing time in general as the number of variables increases. The difficulty arises from the fact that the problem involves Boolean satisfiability problems or tautology checking problems which are known to be NP-complete [60].

Much of the work on the equivalence problem has concentrated on applications in the verification of Boolean circuits. Circuit designers are mostly concerned about the correctness of logic minimizers (or logic optimizers) during the process of circuit design. The correctness of each design level requires that the minimized (or optimized) version has not changed the original meaning of the circuit (i.e., for the same set of input values, the two circuits must produce the same output). So, a verification process is an essential part of circuit design. Other application domains can be found in artificial intelligence, databases, etc., where information or knowledge can be described in Boolean expressions and searching (or matching) is involved.

It is obvious that the inherent nature of the equivalence problem is closely related to the satisfiability problem because both problems rely on the existence of a satisfying instance: Testing tautologyhood of a circuit and testing satisfiability are dual because ϕ is a tautology $\Leftrightarrow \bar{\phi}$ is unsatisfiable, where $\bar{\phi}$ is the negation of ϕ. So, a tautology checker and a satisfiability checker are essentially doing the same task. Another related problem that we can think of is the problem of testing whether certain properties are realized in a Boolean circuit ϕ, which can be solved by first expressing the properties in a Boolean expression, ψ, and then testing the implication $\phi \Rightarrow \psi$. The implication can be tested by checking its tautologyhood.

The equivalence problem has been treated in many different ways. However, we can categorize them into a few groups depending on the type of algorithms used. Here is the list of groupings of algorithms which have contributed significantly to the area.

- Algorithms based on divide-and-conquer and backtracking: Traditionally, this technique has been used most widely in this area. To prove $\phi \equiv \psi$, one has to prove either (1) $\phi \equiv \psi$ is a tautology or (2) $\phi \oplus \psi \equiv$ *false* is a tautology. So, the equivalence problem is interpreted directly as a tautology checking problem. And then a divide-and-conquer type algorithm along with backtracking is invoked to prove the tautologyhood. A variety of algorithms have been de signed along this line, sharing a similar paradigm, with different strategies of backtracking and branching techniques.
- Graph-based algorithms: Algorithms based on *binary decision diagram* (BDD) proposed by Bryant [18] in 1986 are essentially different from divide-and-conquer

type tautology checkers in many respects and have become standard methods for Boolean function verifications. The central part of graph-based algorithms is to construct directed-acyclic graphs (DAGs) which represent a Boolean expression, without ambiguity, without duplication, and in a consistent, and systematic manner. After construction of such graphs, most of the Boolean operations including equivalence testing can be performed by simply using appropriate graph algorithms already developed.

- Algorithms based on logic programming: Simonis and others [27, 43, 131] proposed a new algorithm to solve Boolean circuit verification problems using an extended logic programming language named CHIP. Boolean satisfiability is tested by performing *Boolean unification*, which would show a set of variable bindings if a Boolean circuit is satisfiable.
- Algorithms based on counting: Iwama [73] computes the total number of distinct solutions for a Boolean expression by excluding overlapped solutions for each clause.

None of these works efficiently for every problem instance because of the difficulty of the problem. However, each algorithm can demonstrate a good performance under certain restrictions and problem characteristics. We will discuss the merits and demerits for some of the important works in later sections.

2.2 Satisfiability Problems

We briefly discuss satisfiability problems as well as their variations and classifications. We then review some of the important works done in the area. Finally, we examine some of the algorithms which play important roles in solving the problems.

This section concentrates on describing Boolean satisfiability problems, and the approaches that have been presented to solve the problems. Various types of approaches are reviewed, analyzed and compared with each other. The most popular type of algorithm is *backtracking*. State-of-the-art techniques in backtracking for satisfiability problems are discussed and analyzed in later sections.

A Boolean expression has been the standard representation for describing any decision problem with discrete variables. A typical satisfiability problem for an expression could be to obtain a set of assignments which satisfies the expression or to test its tautologyhood. Problems of this type can be found in many important application areas such as circuit testing, tautology checking, combinatorial problems, etc. Unfortunately, most of these problems are NP-complete.

The *satisfiability problem* or *SAT* is to determine if a Boolean formula is true for some assignment of truth values to the variables. *CNF-Satisfiability* is the satisfiability problem for CNF formulas. Also, this problem is known as a *Constraint Satisfaction Problem* (CSP). The problem is to find an assignment for the variables that satisfies all of the constraints or to determine that the problem is unsatisfiable. The requirement is that all the constraints must be satisfied at once. So, an approximation technique is of little help because of the nature of the problem.

Traditional approaches were to develop algorithms for particular domain of applications in which domain specific knowledge can be built into the algorithms to exploit structural characteristics. The result is that an algorithm may work well on one type of problem, but badly on other types of problems. This type of ad-hoc approach to each individual problem has advantages and disadvantages. The disadvantages are the following: (1) Algorithms used for solving one type of problem might not be easily adapted to solve other problems. (2) It is very hard to combine them with other techniques used for different types of problems to design a more efficient algorithm for solving more general type of problems. (3) It is usually not easy to analyze such an algorithm. So, researchers have tried to find algorithms that can work well under a wide range of conditions. So far, the results have not been satisfactory in the sense that no generally applicable algorithms have been designed, although much progress have been reported in this area.

Parallelization for this area is quite new. Cook and Chen [31, 33] recently presented efficient parallel algorithms for solving *Parallel 2-CNF problem* in logarithmic time on a *concurrent-read concurrent-write parallel random access machine* (CRCW PRAM) which can be solved in linear time by a serial algorithm [45].

Intractable Problems

The satisfiability problem or *SAT* is the standard NP-complete problem, proved by Cook in 1971 [35], which draw much attention from theoretical computer scientists. Since the monumental work done by Cook, an enormous amount of research has been done on intractable problems.

3SAT or 3-CNF is the same as the satisfiability problem (*SAT*) except each clause has exactly three distinct literals. This type of problem is known to be NP-complete. Other satisfiability-related problems which have been proved to be NP-complete can be found in the famous book on intractable problems published by Garey and Johnson [60] in 1979.

Tractable Problems

Since satisfiability problems are hard in general, it became important to identify specific classes of problems which admit tractable solution techniques. Researchers have found some classes of satisfiability problems which can be solved in linear or polynomial time. These tractable classes can also be used to assist in the solution of more general problems.

The following are some of the problems that can be solved in linear or polynomial time.

- Dowling and Gallier [44] showed that *SAT* restricted to *Horn clauses* can be solved in linear time.
- Even *et al.* [45] showed that *SAT* restricted to binary clauses (*2-CNF*) can be solved in linear time.
- Yamasaki and Doshita [157] defined a class H_2, that strictly includes *Horn clauses*, for which Arvind and Biswas [3] proved that *SAT* can be solved in quadratic time.
- Gallo and Scutellà [59] have built a hierarchy, $\Gamma = \Gamma_0, \Gamma_1, \cdots$, of classes, where Γ_0 consists of only *Horn clauses* and Γ_1 is H_2, such that for each Γ_k, *SAT* can be solved in $O(nm^k)$ time, where n is the input size, and m is the number of atomic propositions.

2.3 Satisfiability (Tautology) Checker

In this section, we summarize state-of-the-art technologies in the area of *Satisfiability (Tautology) Checking.* Since so many techniques and algorithms have been developed, it is not helpful to put everything down here. Instead we take some of the significant works into consideration so that we can categorize them into a few or more groups of ideas from which many techniques can be designed.

We also review some of the important algorithms developed for tautology checkers, which have been widely used and have become standard in the area. Yet some techniques can be recognized as radically different from others in many respects. Interesting results have been reported in the area of satisfiability checking using techniques different from backtracking. Algorithms based on non-backtracking work independently and are hard to combine them with other techniques while algorithms based on backtracking are relativel; easy to combine so that more powerful algorithms can be devised by mixing and adapting various techniques. Performances of several algorithms have been measured on random predicates with parameters: v (the number of variables), t (the number of clauses), and p (the probability that each literal appears in a clause).

Tautology Checkers based on Binary Decision Diagram

Bryant [15, 18] proposed a graph-based representation of Boolean functions called *binary decision diagrams* (BDDs). Earlier graphical representations [1, 49, 92, 104, 108] similar to BDD lack in the abilities for practical applications to manipulate Boolean functions systematically and to represent many Boolean functions with reasonable size of graphs. BDD was presented as an improvement on the earlier works and has demonstrated nice properties and advantages over previous approaches. BDD follows the principle of representation based on directed acyclic graphs originated from Lee [92] and Akers [1]. As far as graphical representation is concerned, BDD is the most advanced representation and it has become the standard method for Boolean function manipulations.

Research has been done to improve the performance of BDD in many respects. Most efforts have been made to reduce the size of the graphs by putting various restrictions on their construction such as variable ordering [54, 55, 56]. Some have introduced new functionalities and features into the manipulation functions of BDD, while many other researchers have proposed variations of the diagram [4, 7, 26, 85, 94, 102, 152].

Two major contributions of BDD are:

- A canonical representation for a Boolean function can be realized as the result of constructing a directed-acyclic graph for the function, given a total ordering of input variables.

- Algorithms for combining two functions with a binary operator and for composing two functions are embodied in the system so that the construction of a graph can be done in a consistent and systematic way, which are the central part of BDD.

The most elegant feature of BDD is without doubt the canonicity, which was first observed by Fortune et al [49]. The canonicity means that there exists a *unique* graphical representation for any Boolean expression and BDD guarantees to produce such a graph for a Boolean function as the result of performing the construction process on the input graph. The cost of constructing a canonical graph is reasonable in terms of time and space for many widely used circuits. Experimental results show that BDD can construct a canonical form in nearly linear time proportional to the size of an input graph for a wide variety of circuits.

The canonicity provides a theoretical basis for solving tasks of interest such as testing for equivalence and testing for satisfiability as follows:

1. Testing for equivalence of two Boolean circuits is equivalent to testing whether the two graphs representing the circuits match exactly.
2. The problem of testing for satisfiability involves a depth first traversal of a graph with backtracking in order to search a terminal vertex having value 1 (or *true*). Testing for tautology can be done similarly.
3. The problem of testing whether certain properties are embodied in a Boolean function can be solved by testing the satisfiability of the graph resulting from applying the Boolean product of the original graph which is the canonical representation of the original function and the other graph which represents the properties in terms of a Boolean function in a canonical form.

Graph algorithms already known can directly be utilized to solve these problems and most of the problems have time complexity proportional to the size of the graphs. So, the algorithms based on BDD could be quite efficient provided that the graphs can be constructed in a reasonable amount of time and the size of the graphs is not big.

Because of the strict ordering of input variables, BDD acquires the ability to produce a unique canonical structure for a Boolean function. However, the same fixed ordering scheme may require too much space when the initial ordering is not good. In other words, the size of the final canonical graph for a Boolean function is highly sensitive to the ordering of input variables, which is the major drawback of BDD. That is, a poor choice of ordering may cause the system to require too much computing resources.

Example 2.1 For a Boolean function $(a \wedge b) \vee (c \wedge d) \vee (e \wedge f)$, the system requires 8 vertices with an ordering $a < b < c < d < e < f$ but 16 vertices with an ordering $a < d < b < e < c < f$. In general, the function $(x_1 \wedge x_2) \vee \cdots \vee (x_{2n-1} \wedge x_{2n})$ can be realized by a graph of $2n + 2$ vertices with the ordering $x_1 < x_2 < \cdots < x_{2n}$ but 2^{n+1} vertices with the ordering $x_1 < x_{n+1} < x_2 < x_{n+2} < \cdots < x_n < x_{2n}$. $\square$

For a simple circuit, it may not be hard to obtain a good ordering either manually or algorithmically, but it may become quite difficult as the size of input grows. In fact, an optimal ordering for a Boolean circuit for constructing a canonical graph with minimum number of vertices is known to be co-NP. Many algorithms have been designed to get a good ordering although it may not give an optimal solution. Some of the results show nice performances on a wide range of Boolean functions [54, 55, 56, 75, 96]. Yet we believe that no polynomial algorithm for computing such an ordering in general is feasible because of the NP-completeness of the problem. An optimal ordering can be extremely hard to obtain especially when the input circuit shows little symmetry or little hierarchy but much randomness.

There are some Boolean functions which are inherently hard to be represented with BDD graphs regardless of the ordering of variables. For example, integer multipliers cannot be represented in BDD as the number of vertices of the graph grows exponentially [18], while others report practical performances without using graphical representations [132, 147, 148].

Tautology Checkers based on Divide-and-Conquer

Algorithms based on divide-and-conquer are popular in this area and a variety of algorithms have been designed and implemented. However, there are some concepts which are shared by most of the algorithms. Yet we still can point out other distinctive techniques exclusively used in some algorithms, which have shown their merits.

The theoretical basis for divide-and-conquer algorithms is Shannon's expansion principle [129]. The order of variables involved in the expansion directly controls the behavior of such algorithms and severely affects the performance. Most of the tautology checkers of this kind emphasize choosing the best variable and simplifying so that the length of the proof or the size of the subproblems could be minimized. Since algorithms for finding an optimal ordering are costly, most algorithms use heuristics to choose a good variable which is likely to lead to smaller subproblems or big simplifications. It is natural to take advantage of the characteristics of problem domains for designing efficient heuristics [7, 136, 54, 90, 136, 149].

The first multilevel tautology tester was VERIFY, developed by Roth [125, 123], which is a specialization of *consistency* of the *D-algorithm* [126]. VERIFY seeks a counterexample that shows a discrepancy of two logic designs, starting with an output pair. If the program fails to find one, then the two designs are equivalent.

Smith *et al.* [136] developed a hardware design verification methodology for the IBM 3081 project in 1982. Boolean comparison was incorporated in the system to verify that two design representations — hardware flowcharts and the detailed hardware logic design — are logically equivalent. The algorithm uses domain-specific heuristics to choose one variable for reduction of the Boolean expression at the current branch point by Shannon expansion. A manual intervention is carried out to select a variable in case the heuristics fail. A simple backtracking scheme is used.

Haralick and Wu [65] developed a divide-and-conquer type theorem prover, as an improvement of resolution theorem proving programs and Wang's technique [151], not based on Shannon expansion but based on a set of *divide-rules* which divides a material implication into two subexpressions, preserving logical equivalence. The program uses fairly simple heuristics to choose the next expression to be divided.

Gelder [61] developed a divide-and-conquer satisfiability checker which has similar characteristics with previous ones. But it shows many superior qualities comparing with earlier systems in the sense that it emphasizes simplification rules and branching techniques. An initial transformation is required such that only leaf nodes in the search tree may contain negations, which is called an *AND-OR tree*. And by putting a few more restrictions an *AND-OR tree* can be transformed into a *succinct AND-OR tree*. The succinctness allows the system to use simplification rules based on *dominance* to collapse variables and other techniques to remove literals, preserving satisfiability. Simplification rules preserving satisfiability would give more chances to reduce the search space than those preserving logical equivalence. Another contribution is that the system employs heuristics for branching techniques depending on the number of occurrences and the type (mixed or symmetric) of a variable.

INSTEP, presented by Vlach [147, 148, 149], is a multilevel satisfiability checker designed for a verification of a commercial VLSI chip manufactured by Texas Instruments. Results on IFIP and ISCAS benchmarks are overall competitive with those for the widely used methods based on *binary decision diagrams* or BDDs. One of the major contributions of INSTEP is for the first time it shows the solution in polynomial time of certain benchmarks involving combinational multipliers, while any program based on BDDs runs in exponential time.

INSTEP has unique features in simplification strategies and backtracking techniques. The main focalization of the program is simplification which takes place in the beginning of a verification process before a backtracking procedure is initiated. Non-equivalence preserving rules are extensively used depending on the polarity and the number of occurrences of a variable. A special type of backtracking algorithm called *dependency-directed backtracking* (DDB) is designed for INSTEP. Experimental evidence shows that DDB reduces the search space substantially by keeping track of good branch expressions and by eliminating nonessential ones so that the procedure can find a short path to a contradiction.

The introduction of *higher-level operators* and nonstandard operators such as **if3+**, **if4+**, **adder**, **half-adder**, etc. is unique and proved to be very effective in verification. INSTEP shows that simplification alone is enough for many benchmarks without even having to call the backtracking procedure. The system will work most efficiently when a Boolean circuit demonstrates certain characteristics described as follows: (1) Hierarchical structures in terms of input and output (2) Sparsity of occurrence of variables (3) Repetition of similar elements (4) Occurrences of specific configurations such as half-adder, adder, etc.

Algorithms based on divide-and-conquer have the following advantages: (1) It is relatively easy to combine them with other techniques developed for simplification and backtracking. (2) It is also easy to apply various types of simplification rules at different places of algorithms. (3) Characteristics of each problem domain can be encoded in the system so that the performance might be maximized.

In conclusion, divide-and-conquer algorithms are widely used in the area and work quite efficiently in many cases. Their performance is heavily depended upon the effectiveness of simplification rules and backtracking techniques. Since most of the systems naturally incorporate domain-specific knowledge into the rules and heuristics, they may not work well for other problem domains.

Logic Programming

Using an extended logic programming language CHIP, developed by Dincbas *et al.* [43], whose main objective is to solve search problems with constraints in areas like Operations Research, financial planning or digital circuit design. Simonis *et al.* [27, 43, 131, 134, 135] presented a new method to solve Boolean satisfiability problems using *Boolean unification.* The program would show a set of variable bindings as the result of unification process if a Boolean circuit is satisfiable. Experimental results show that the program works efficiently on a number of benchmark circuits.

The major difference between CHIP and other ordinary logic programming languages lies in its internal representation of Boolean terms which holds the same characteristics of the canonicity of binary decision diagram (BDD). Performing circuit verifications using logic programming languages is an interesting approach and it should receive more attention from the community.

Counting Algorithm

The algorithm proposed by Iwama in 1989 [73] computes the number of solutions for each clause and then count the number of solutions for each pair of clauses that depend on different variables, and continue for triples, etc. For each step overlapping solutions are discarded by using the *inclusion-exclusion principle.* The efficiency of this algorithm clearly depends on the number of overlaps, which can be exponentially large in general.

Iwama's algorithm runs in polynomial time when $p > \sqrt{\frac{\ln t}{v}}$.[1] This algorithm is quite efficient when p is large. That is, the more literals in a clause, the more chances the algorithm works efficiently. The major advantage of this algorithm is that it works efficiently where backtracking algorithms work less efficiently because the more literals in a clause, the more branch points would occur in backtracking. So, the algorithm can work as complementary to backtracking, but how to combine the two schemes to design a better algorithm is not clear, since counting and backtracking are totally different schemes in nature. Later, Tanaka in 1991 [142] reported preliminary results on a refinement of Iwama's algorithm.

Algorithms for Better Upper Bounds

The BSAT algorithm, proposed by Monien and Speckenmeyer [103] in 1985, is based on branching techniques which can solve the *3SAT* in less than 2^n steps (1.62^n, where n is the number of variables), which establishes a better upper bound, although not much significance for practical purpose is expected. A general solution can be found in the paper for *k-SAT*, $k \geq 3$.

For non-clausal Boolean expressions, Gelder in 1988 [61] proved that satisfiability testing in the propositional calculus can be done with an upper bound of $2^{(0.25+\epsilon)L}$ for any positive ϵ, where L is the length of the input expression. According to his algorithm, the number of input variables is not an essential part of the complexity consideration.

2.4 Backtracking

A divide-and-conquer type tautology checker typically selects a variable to simplify input problems by assigning a truth value, which in turn generates two smaller

[1] v, t, and p are defined on page 17.

problems each of which is solved by employing the same assigning and simplifying technique recursively. A backtracking process will be invoked whenever there is no hope of reaching a solution from the current branch point and it repeats the process until every possibility is exhausted. The fundamental idea behind backtracking methods is *Shannon cofactoring* [129], which is a theoretical basis for many divide-and-conquer tautology checkers and has become a standard technique in the area.

The performance of such an algorithm heavily depends on strategies to select a variable to be assigned and to decide its truth value. In other words, an efficient algorithm should find a variable which can lead to a big simplification or to a short path to a solution (or a contradiction). Also, the algorithm should show the ability to choose the right truth value for the variable selected so that the best result can be achieved. Designing such an algorithm for all instances of problems may not be feasible. However, it is possible that algorithms can be designed so that they can run in polynomial *average time* within certain restricted ranges of control parameters [23, 62, 73, 111]. Unfortunately, none of these have a practical range of parameters in which algorithms may demonstrate a polynomial average time behavior.

Simple Backtracking

This scheme selects the variables in a fixed order. However, the algorithm does not provide any clue to how we order the variables to reduce the number of steps to reach a solution. The range of control parameters in which simple backtracking runs in polynomial average time is not practical at all [110].

Unit Clause Backtracking

This method [112, 113] basically concerns the choice of the next variable to simplify a propositional formula. The basic idea of choosing the next variable is the following: If some clause depends on only one of the unset variables, then select that variable, otherwise select the first unset variable and continue as in backtracking.

The theory behind this idea is the concept of *intermediate predicate* P_A defined over sets $A \subseteq S$, where S is the set of all variables. The intermediate predicate must have the values *true* for any assignment of values to the variables in A that can be extended to solution to P. When $P_A = 0$ (or *false*), there is no hope of extending the partial setting P_A to a solution. So, the tenet is to extend partial solutions unless the values of partial predicates are *false*. Subproblems can be reduced as early as possible by excluding those partial settings which would never give solutions to P. When there is no satisfying assignment of variables to a problem, the algorithm would backtrack until every possibility is exhausted. This algorithm is considered to be superior to simple backtracking because there is a region where simple backtracking takes exponential time while unit clause backtracking takes polynomial time.

Clause Order Backtracking

The algorithm [23] selects the first clause that is not always *true*. For the first variable that affects the value of this clause, each possible value is plugged in, the predicate simplified, and the resulting subproblem is solved by recursive application of the algorithm. Each solution of the subproblem (along with the partial assignments of values that lead to the subproblem) gives a solution to the original problem. If the original problem has no nontrivial clauses, then every assignment of values to the remaining variables results in a solution.

The average time for clause order backtracking is always less than that for simple backtracking. It leads to polynomial time under many conditions where simple backtracking uses exponential average time. *Clause order backtracking* runs in polynomial average time when

$p < \frac{1}{2v}$, and $p > \sqrt{\frac{\ln t + \frac{\ln v}{2}}{v}}$.

Also it is empirically shown that clause order backtracking with the unit clause rule is better than plain clause order backtracking.

Pure Literal Rule Algorithm

This algorithm [62] uses the basic idea of the *Davis-Putnam procedure*, which is a powerful technique for solving the satisfiability problem. The algorithm first searches for a unit clause and sets the variable belonged to the clause so that the clause is *true* and simplify the original predicate using the assignment. If any variable appears only negated or unnegated, set the truth value of it so that its literals are *true*. Simplify the original predicate based on that assignment. Then, the original predicate is satisfiable if the simplified predicate is. If no pure literal can be found in input predicate, apply the Shannon Cofactoring, which forms two predicates by setting the variable to each value and simplifying the resulting predicates. The original predicate is satisfiable if at least one of the simplified predicates is satisfiable. If each subpredicate happens to be simplified much, then the algorithm would be left with less search space. But for some types of predicates, we may not be able to simplify much and we would end up with fast growing number of subpredicates. This algorithm uses polynomial time when

$t < \frac{\ln v}{\epsilon}$ and $p > \epsilon$ or

$tp \leq \epsilon\sqrt{\frac{\ln v}{v}}$,

where ϵ is any fixed small number [111].

Dependency-Directed Backtracking

Whenever the current set of expressions is found to be inconsistent, INSTEP [148] initiates the *directed backtracking* algorithm. The major part of backtracking in INSTEP is *Dependency Directed Backtracking* (DDB), which keeps track of which branch expressions (i.e., expressions on which branching is done) are involved in the derivation of a contradiction. Such branch expressions are recognized as *essential branch expressions*. When a backtracking starts, nonessential branch expressions are excluded so that unnecessary branches cannot be generated subsequently. This would reduce the number of branch points exponentially. Experiments show that DDB reduces the search space substantially by removing nonessential branch expressions.

To efficiently detect essential branch expressions, INSTEP embodies the following observations:

- Branch expressions that immediately resulted in the contradiction are essential.
- Branch expressions that resulted in big simplifications of the current set of expressions are most likely essential.
- Branch expressions that led quickly to a contradiction previously are likely to be essential to the current derivation of the contradiction.

These potentially essential expressions will most likely be involved in the derivation of the contradiction. Based on these information, INSTEP assigns an appropriate priority to each branch expression. INSTEP achieves a very large speed-up in backtracking by removing nonessential expressions and by reordering branch expressions according to the priorities associated with branch expressions.

2.5 Simplification Techniques

Since a divide-and-conquer type algorithm for Boolean function manipulations requires binary tree traversal in depth-first, the search would soon be victimized by combinatorial explosion unless an efficient algorithm to reduce the search space is used. Many types of simplifications have been proposed and evolved since the birth of the mechanical theorem proving. In this section, we discuss and analyze some of the significant ones to outline its development and current techniques.

Almost all tautology checkers employ truth functional simplifications which preserve logical equivalence, introduced in [118]. Yet others adopt satisfiability preserving rules to allow more simplifications in addition to the truth functional simplifications. The idea of applying such rules to reduce the search space is due to Davis and Putnam [40]. They presented four simplification rules which preserve satisfiability. These are the *tautology rule, one-literal rule, pure-literal rule* and *splitting rule*, which have been extensively used and proved to be very effective in mechanical theorem proving. Researchers in verification area have embodied the rules in their systems with extensions and generalizations [14, 61, 65, 147]. Gelder [61] generalized the pure-literal rule of Davis-Putnam procedure to non-clausal expressions. A set of new simplifications of this kind (*Non-Equivalence Preserving* or NEP rules) depending on the polarity of variables is introduced in INSTEP [147]. The NEP simplifications are justified by the *Image Theorem* and they can achieve more reductions than the truth functional ones in many cases. The extensive use of such rules is the primary cause for INSTEP's ability to verify the multipliers in the IFIP benchmarks in polynomial time.

Extended Shannon cofactoring [61, 65, 125, 147] takes cases on complex formulas as well as variables, which provide a means to allow more simplifications. For example, Boolean expressions like **(or** ϕ ψ**)** or **(iff** ϕ ψ**)**, where ϕ and ψ may be complex formulas, can be branched into two sub-expressions by taking cases on the truth value of ϕ.

Example 2.2 Let Γ be a set of formulas and A, B be formulas.

$\Gamma \cup \{$(iff A B)$\}$ is satisfiable

iff either $\Gamma \cup \{$A, (iff T B)$\}$ or $\Gamma \cup \{$(not A), (iff F B)$\}$ is satisfiable

iff either $\Gamma \cup \{$A,B$\}$ or $\Gamma \cup \{$(not A), (not B)$\}$ is satisfiable. □

Nonstandard higher-level Boolean operators, as opposed to the standard Boolean operators such as **and**, **or**, **not**, etc., are effectively incorporated in INSTEP so that more simplified and succinct representation of circuit elements can be achieved, expressing the same truth values. Thus, the formula (**or** (**and** ϕ ψ) (**and** (**not** ϕ) ψ)) now can be equivalently described in a simple form:

(**if3** ϕ ψ χ),

which defines a three-place *if-then-else* expression. For further simplifications INSTEP introduces a set of new higher-level operators such as **if3+**, **if4+**. For example, the definitions of **if3+** and **if4+** are given as follows:

(**iff** (**if3+** ϕ ψ χ) (**if3** ϕ (**or** ψ χ) (**not** ψ))).

(**iff** (**if4+** ϕ ψ χ ξ) (**if3** ϕ (**and** ψ (**or** χ ξ)) (**or** (**not** ψ) (**not** ξ)))).

Some higher-level operators depict circuit elements at functional level rather than gate level. For example, **adder** and **half-adder** are defined by the following expressions:

(**iff** (**adder** ϕ ψ χ ξ ζ) (**and** (**iff** ϕ (**al2** χ ξ ζ)) (**even** ψ χ ξ ζ))).

(**iff** (**half-adder** ϕ ψ χ ξ) (**and** (**iff** ϕ (**and** χ ξ)) (**even** ψ χ ξ))).

A formula beginning with **al2** is true when at least two of its arguments are true. A formula beginning with **even** is true when even number of its arguments are true. ϕ and ψ of **adder** and **half-adder** represent carry and sum outputs, respectively. The remaining signals represent the inputs. These special operators are especially beneficial in the verification of a particular type of circuits in which they are components of the circuits and hierarchically structured. Tautology checking for some circuits becomes quite easy when higher-level operators are used in a simplification process. Such an example can be found in Vlach [147] for a parity checker:

(**iff** (**odd** A B C D) (**xor** (**xor** A B) (**xor** C D))).

INSTEP simplifies it to *true* in linear time for this type of circuit while a divide-and-conquer tautology checker without simplification may involve combinatorial search.

The notions of *control* and *dominance* [61, 147] enable the system to allow simplifications that can easily be identified such that the actual reduction process is computationally cheap.

CHAPTER 3

THE DERIVATIONAL METHOD

In this chapter, we first introduce basic terminologies and concepts regarding the *derivational method.* Next, we introduce the notion of *cover* for the equivalence problem and outline briefly how the concept can be used to solve the equivalence problem. The Merge Rule and other operations are also introduced. We propose a new deductive apparatus for formal verification: a *derivational approach* using the Merge Rule as the rule of inference.

Next, the concept of *minimal cover* is investigated in order to remove redundancies from input expressions, and also to provide a means of proof of the theorems essential to equivalence testing. The *uniqueness property* enables us to perform simplification based on *projection reduction*, which is extensively used in verification. Mergeability accounts for the condition of the existence of a merge and also for simplification based on *nonmerge reduction.*

Finally, we introduce the concept of *losslessness*, which deals with the consequences of merges and addresses those issues that a merge can cause. Our interest in losslessness is to study the effects of different types of merges as to what can be lost as the result of merges and how lost information can be recovered, etc.

3.1 Introduction

We assume that $x_1, \cdots, x_n$ are input variables and also assume that both x_i and $\bar{x}_i$ do not occur in a term. The symbols 1, 0, and d indicate True, False, and don't-care, respectively. A *term* (or *and-term*) can be represented by a string of symbols: $a_1 a_2 \cdots a_n$,

$$a_i \ (1 \leq i \leq n) = \begin{cases} 1 & : \ x_i \text{ occurs in the term} \\ 0 & : \ \bar{x}_i \text{ occurs in the term} \\ d & : \ \text{neither } \bar{x}_i \text{ nor } x_i \text{ occurs in the term.} \end{cases}$$

For example, $\bar{x}_2 x_3 \bar{x}_5$ is represented by $d01d0$, when $n = 5$. A string composed of $\{0, 1, d\}$ is to be regarded as a term or a disjunction of minterms for the term depending on the context (for example, $d01d0$ can be treated as a disjunction of four minterms: $00100 + 00110 + 10100 + 10110$). We use $a[i]$ to represent a_i. Note that the above representation is conceptual, but for the actual implementation the symbol d can be omitted by using a linked list storing only literals and their positional information.

We define two special types of terms: *unit term* and *complete term*.

Definition 3.1 A *unit term* is a term which contains a single literal in it. A *complete term* with n variables consists only of n occurrences of the symbol d, denoted by σ_n. σ_n is often used to represent a tautology. □

Unit terms are valuable for simplifications where the average number of literals in a term is very small (best if it is close to one).

Definition 3.2 The *cardinality* of a term α is the number of literals in it, denoted by $|\alpha|$ or $Lit(\alpha)$. The *cardinality* of a set of terms ϕ is the number of terms in it, denoted by $|\phi|$. □

Definition 3.3 Define a PLA as the set of terms occurred in a Boolean expression in DNF. Let ϕ_1, ϕ_2 be two PLAs ($|\phi_1|, |\phi_2| \geq 1$). ϕ_1 *covers* ϕ_2, denoted by $\phi_2 \in^c \phi_1$, iff for each term $\beta \in \phi_2$, any assignment that satisfies β also satisfies a disjunction of some terms of ϕ_1. □

We can rewrite it using the definition of *cover*:

ϕ_1 *covers* ϕ_2 iff each term $\beta \in \phi_2$ is covered by $\psi \subseteq \phi_1$.

A term can be treated as the set of minterms covered by the term. Likewise, a PLA can be treated as the set of minterms covered by the PLA. Therefore, we can rewrite the definition as follows:

ϕ_1 covers ϕ_2 iff ϕ_1, considered as a set of minterms, is a superset of ϕ_2.

Testing cover between two terms can be done easily by comparing literal values at each position without enumerating minterms. Formally, α *covers* β iff for each position i such that $\alpha[i] = 0$ or 1, $\beta[i] = \alpha[i]$. For example, $d01d0$ covers $d0110$, but not $dd110$.

We state the notion of *equivalence* in terms of *cover*, which basically divides the original problem into two subproblems.

Theorem 3.1 (Equivalence between two PLAs) ϕ_1 is *equivalent* to ϕ_2 iff ϕ_1 covers ϕ_2 and ϕ_2 covers ϕ_1. □

Example 3.1 Let $\phi_1 = x_4 + \bar{x_2}\bar{x_3} + \bar{x_1}\bar{x_4} + x_1\bar{x_2} + x_2\bar{x_4}$ and $\phi_2 = \bar{x_2}x_4 + \bar{x_1}x_4 + \bar{x_3}x_4 + x_1x_2x_3x_4 + \bar{x_3}\bar{x_4} + x_3\bar{x_4}$. Then, ϕ_1 and ϕ_2 are equivalent because $\phi_2 \in^c \phi_1$ and $\phi_1 \in^c \phi_2$. □

Note that each of the two subproblems of the same kind can in turn be divided into many smaller problems of deciding whether or not a set of terms in one side covers a single term in the other side. This simple *divide-and-conquer* strategy increases the probability of getting more simplifications.

The following terminologies are frequently used in this thesis in order to describe the relationship between two terms.

Definition 3.4 Let α, β be any arbitrary terms.

(1) *Complementary:* The position j of α, β is called a *complementary position*

if $\alpha[j] \neq d, \beta[j] \neq d$, and $\alpha[j] = 1 - \beta[j]$ (or, $\beta[j] = 1 - \alpha[j]$). α and β are called *complementary terms* if there is a complementary position j and for any other position k, $\alpha[k] = \beta[k]$. (e.g. $0d1d$ and $0d0d$ are complementary.)

(2) *Disjoint:* α, β are called *disjoint terms* if there is a position j such that $\alpha[j]$ and $\beta[j]$ are complementary. We define the *distance* of two disjoint terms as the number of complementary positions. (e.g. $0d1d$ and $110d$ are disjoint with distance = 2.)

(3) *Identical:* The position j of α, β is called an *identical position* if $\alpha[j] = \beta[j]$. α and β are called *identical terms* if $\alpha[j]$ is identical to $\beta[j]$ for each position j.

(4) *Inclusive:* The position j of α, β is called an *inclusive position* if *either* (a) $\alpha[j] = d$, $\beta[j] = 0$ or 1 or (b) $\alpha[j] = 0$ or 1, $\beta[j] = d$. α and β are called *inclusive* if one is covered by the other. $I(\alpha, \beta)$ denotes the number of *inclusive positions* between the two terms α, β. (e.g. $0d1d$ and $011d$ are inclusive.)

(5) *Shared:* α and β are called *shared* if α and β are not *disjoint*. (e.g. $0d1d$ and $01d0$ are shared.)

(6) *Common Literal Position:* A *common literal position* of two terms α and β is a position j in which $\alpha[j] = \beta[j] \neq d$. □

3.2 The Merge Rule and Derivation

The Quine-McCluskey method [99, 117] was the first algebraic approach to produce a simplified expression for a Boolean function. The method is based on tabulation which overcomes many drawbacks of the earlier method called the Karnaugh map [81]. The tabulation method starts from the list of minterms that specify a Boolean function and simplifies by combining two complementary terms. It repeats the process until no new simplification is possible. This method works efficiently for minimizing Boolean expressions of up to 10 variables. The Merge-Rule introduced in this section has similar idea of combining two terms. But two terms need not be complementary to be merged. So, the Merge-Rule can be considered as a generalization of the previous work.

The Merge Rule is the only inference rule employed for the derivation procedure. Later, we will show that the derivational method using the Merge Rule as its sole rule of inference is *sound and complete* in deciding whether or not a PLA covers a term.

Definition 3.5 A *merge* γ of two terms α and β is a term such that (1) α and β have exactly one complementary position and (2) for each position i, $\gamma[i]$ is defined by the *Merge Rule* shown in Table 2. The resulting γ is called the *merged term* or *merge* of α and β. □

$\alpha[i]$	0	0	0	1	1	1	d	d	d
$\beta[i]$	0	1	d	0	1	d	0	1	d
$\gamma[i]$	0	d	0	d	1	1	0	1	d

Table 2: The Merge Rule

Definition 3.6 γ is called an *immediate consequence* of two terms α and β if $\gamma[i]$ is the result of applying the Merge Rule to $\alpha[i]$ and $\beta[i]$ for each position i, denoted by $\{\alpha, \beta\} \mapsto \gamma$. A *merge position* (MP) for $\{\alpha, \beta\} \mapsto \gamma$ is the position j in which $\alpha[j]$ and $\beta[j]$ are complementary and $\gamma[j] = d$, denoted by MP(α, β). There is only one merge position for any merge. □

We observe the following simple facts which will be used in later sections for proving theorems.

Observation 3.1

1. If $\{\alpha, \beta\} \mapsto \gamma$, then $\alpha + \beta$ covers γ (because $\alpha \cup \beta$ is a superset of γ).

2. If $\{\alpha, \beta\} \mapsto \gamma$, then (a) α and γ are either *inclusive* or *shared* and (b) so are β and γ. □

There are two special types of merges: *unit merge* and *simple merge*. They are considered to be quite useful for simplification.

Definition 3.7 γ is said to be a *unit merge* of two terms α, β iff α, β are mergeable and at least one of them is a *unit term*. γ is said to be a *simple merge* of two terms α, β iff α, β are complementary. □

Definition 3.8 Δ is called a *derivation* of a term γ from a set of terms Γ iff

(1) Δ is a finite sequence of terms.

(2) the last term in Δ is γ.

(3) each term of Δ is either a member of Γ or an immediate consequence by the Merge Rule of two terms preceding it in the sequence. □

$\Gamma \mapsto \gamma$ denotes a derivation of a term γ from a set of terms Γ. $\Gamma \not\mapsto \gamma$ denotes that it is not the case that $\Gamma \mapsto \gamma$. In Section 4.1, we show how the derivational method can be used for cover testing. Since the derivation process may contain redundancies, it is crucial to exclude them for efficient derivation, which is discussed in Section 4.2.

The following theorem, a generalization of Observation 3.1 (1), says that a merged term is always covered by the base set of terms from which it is derived.

Theorem 3.2 If $\{\alpha_1, \cdots, \alpha_m\} \mapsto \gamma$ then $\{\alpha_1, \cdots, \alpha_m\}$ covers γ.

Proof: Let $s = s_1, \cdots, s_r$ be any sequence of derivation of γ from $\Gamma = \{\alpha_1, \cdots, \alpha_m\}$, where $s_r = \gamma$. When $r = 1$, there exists a merge: $\{\alpha_i, \alpha_j\} \mapsto \gamma$, $s_r = \gamma$. Then, it is obvious that $\{\alpha_i, \alpha_j\}$ covers γ. Assume that the theorem holds for $r \leq k (> 1)$. Suppose $r = k + 1$. If $s_r \in \{\alpha_1, \cdots, \alpha_m\}$, then $\{\alpha_1, \cdots, \alpha_m\}$ immediately covers s_r. Otherwise, s_r must be an immediate consequence of merging two terms s_p and s_q preceding s_r in the sequence, where $p, q \leq k$. Then, by the hypothesis, $\{\alpha_1, \cdots, \alpha_m\}$ covers s_p and $\{\alpha_1, \cdots, \alpha_m\}$ covers s_q. So, $\{\alpha_1, \cdots, \alpha_m\}$ covers $s_p \cup s_q$. But $s_p \cup s_q$

covers s_r. Hence, $\{\alpha_1, \cdots, \alpha_m\}$ covers s_r. $\square$

3.3 Operations

We define *union*, *intersection*, *difference*, and *projection* operations on terms, which will be frequently used in subsequent algorithms as basic functions. Operations on terms and operations on sets are essentially same except projection if we interpret a term as the set of minterms covered by it. Computing union, intersection and projection on terms is cheap, but computing difference is much more costly.

Definition 3.9 (1) *Union*: The *union* of two terms α and β, denoted by $\alpha \cup \beta$, is $\{t \mid t$ is a minterm such that $t \in^c \alpha$ or $t \in^c \beta\}$.

(2) *Intersection*: The *intersection* of two terms α and β, denoted by $\alpha \cap \beta$, is $\{t \mid t$ is a minterm such that $t \in^c \alpha,\ t \in^c \beta\}$. The intersection γ of α, β can be computed as follows:

(i) If α and β are disjoint, then $\gamma = \emptyset$.

(ii) Otherwise, for each position i, $\gamma[i]$ is defined by the Intersection Rule shown in Table 3.

(3) *Difference*: The *difference* of two terms α, β or *relative complement* of β *with respect to* α, denoted by $\alpha \ominus \beta$, is $\{t \mid t$ is a minterm such that $t \in^c \alpha,\ t \notin^c \beta\}$.

(4) *Projection, Complementary Projection*: We define a new operation called

$\alpha[i]$	0	0	1	1	d	d	d
$\beta[i]$	0	d	1	d	0	1	d
$\gamma[i]$	0	0	1	1	0	1	d

Table 3: The Intersection Rule

projection whose main purpose is to extract some portion of terms and rearrange their components. The *projection* is extensively used to prove some theorems as well as to simplify the verification process. Let α, β be two terms of Γ. The *projection* of α with respect to β, denoted by $\alpha^{\Pi}|_{\beta}$, is $\alpha^{\Pi}[j]|_{\beta} = \alpha[i]$, where i is the jth position of β whose value is d. The *projection* of Γ with respect to β, denoted by $\Gamma^{\Pi}|_{\beta}$, is the set of projections $\{\alpha_1^{\Pi}|_{\beta}, \cdots, \alpha_m^{\Pi}|_{\beta}\}$, where $\alpha_i^{\Pi}|_{\beta}$ is the projection of α_i with respect to β for each $\alpha_i (\neq \beta) \in \Gamma$. We abbreviate $\beta^{\Pi}|_{\beta}$ to β^{Π} which is equivalent to σ_r, where $r = n - |\beta|$. The *complementary projection* of α with respect to β, denoted by $\alpha^{-\Pi}|_{\beta}$, is exactly like $\alpha^{\Pi}|_{\beta}$ except that i is the jth position of β whose value is not d. Similarly, we define $\Gamma^{-\Pi}|_{\beta}$ and $\beta^{-\Pi}$. We abbreviate $\alpha^{\Pi}|_{\beta}, \Gamma^{\Pi}|_{\beta}, \alpha^{-\Pi}|_{\beta}, \Gamma^{-\Pi}|_{\beta}$ to $\alpha^{\Pi}, \Gamma^{\Pi}, \alpha^{-\Pi}, \Gamma^{-\Pi}$, respectively if there is no ambiguity. □

Example 3.2 Let $\Gamma = \{ddd11, 1ddd1, d001d, 0d1d1, 100dd\}$ and $\beta = d0d1d$. By applying the projection rule, we obtain, $\beta^{\Pi} = ddd$ and $\Gamma^{\Pi}|_{\beta} = \{dd1, 1d1, d0d, 011, 10d\}$. For complementary projections, $\beta^{-\Pi} = 01$ and $\Gamma^{-\Pi}|_{\beta} = \{d1, dd, 01, dd, 0d\}$. □

The following facts are immediately understood on the above operations:

Observation 3.2

1. Let Γ, Δ be sets of terms. If $\Gamma \mapsto \gamma$, then $\Gamma \cup \Delta \mapsto \gamma$.
2. If Γ is not a cover for a term β, then there is a set Δ such that $\Gamma \cup \Delta$ is a cover for β.
3. Two terms, α and β, are disjoint iff $\alpha \cap \beta = \emptyset$.
4. Let t be a minterm. If $t \cap \alpha \neq \emptyset$ and $t \cap \beta \neq \emptyset$, then $\alpha \cap \beta \neq \emptyset$. □

3.4 Minimal Cover

In this section, we introduce the concept of *minimal cover*. Suppose a set of terms Γ covers a term β. Then, it is possible that some terms of Γ may not be essential to the cover. We will show that removing this type of redundancy not only gives a means for simplification, but provides, more importantly, some of the crucial properties for the proofs of the main theorems for our equivalence testing algorithm.

Definition 3.10 A set of terms Γ is said to be a *minimal cover* for a single term β iff

(1) Γ covers β and

(2) no proper subset of Γ covers β. □

The following facts are observed regarding minimal cover (trivial proofs by contradiction).

Observation 3.3

1. If Γ is a minimal cover for β, then each term of Γ contains at least one minterm of β.
2. If Γ is a minimal cover for β, then there is no term $\alpha \in \Gamma$ such that α is disjoint with β. □

We now show that there exists a subset of Γ which is a minimal cover for a term β, provided that Γ covers β.

Theorem 3.3 (Existence of a minimal cover) If Γ covers β, then there exists a subset Γ' of Γ such that Γ' is a minimal cover for β.

Proof: Let $\Gamma^0 = \Gamma$ be the set of terms $\{\alpha_1, \cdots, \alpha_m\}$. Define a recursion as follows:

```
for $j = 1$ to $m$,
    if $(\Gamma^{j-1} - \{\alpha_j\})$ covers $\beta$,
        then $\Gamma^j = \Gamma^{j-1} - \{\alpha_i\}$;
        else $\Gamma^j = \Gamma^{j-1}$;
end.for;
```

The theorem follows immediately from a trivial proof by contradiction. □

Example 3.3 Let $\beta = d0d1$ and $\Gamma = \{d00d, 0dd1, 0d1d, 10dd\}$. Then, $\Gamma_1 = \{d00d,\ 0d1d,\ 10dd\}$ covers β and Γ_1 is a subset of Γ. However, any subset of Γ_1 does not cover β. So, Γ_1 is a minimal cover for β. □

A minimal cover is not necessarily optimal in terms of cardinality.

Definition 3.11 Let $\Gamma = \{\alpha_1, \cdots, \alpha_m\}$ cover β. Γ_{opt} is said to be an *optimal cover* for β if

(1) Γ_{opt} is a cover for β and

(2) any other cover has at least as many members as Γ_{opt} has. □

For example, $\Gamma_2 = \{0dd1, 10dd\}$ is an optimal cover for the previous example. It is obvious that an *optimal cover* is a *minimal cover* by the definition of *optimal cover*, but not vice versa. The actual computation of optimal cover seems too costly to be practical, even though it has some theoretical importance.

3.5 Uniqueness

The idea of minimal cover emerges from the observation that some terms of Γ may be redundant with respect to a cover. Likewise, we observe that some minterms of a term may not be essential to a cover. In this section, we introduce the concept of *uniqueness*, which tells about the essentiality of a minterm as well as a term. Next, we investigate the properties associated with uniqueness and establish the relationship between the two concepts: uniqueness and minimal cover. Later, we further study how the concepts of uniqueness, minimal cover, and merge are interrelated.

Definition 3.12 A *minterm* t of a term α is said to be *unique* with respect to a set of terms Γ (or simply, t of α is *unique* when Γ is assumed) iff $t \in^c \alpha$ and there is no $\beta \in \Gamma - \{\alpha\}$ such that β covers t. □

Example 3.4 Let $\Gamma = \{d0dd, dd1d, dd00, d1d1\}$. Then, $t = 0001 \in d0dd$ is *unique* with respect to Γ. But 0000 is not. □

$\check{U}(\alpha)$ denotes the set of all minterms of α that are unique with respect to Γ. For example, $\check{U}(d0dd) = \{0001, 1001\} = d001$ for the previous example.

We show the sufficient condition for the existence of a unique minterm for each term of Γ, which will be used in subsequent theorems.

Lemma 3.1 If Γ is a minimal cover for a term β, then each member of Γ has at least one unique minterm of β.

Proof: Suppose not. Then, there exists a term α such that $\Gamma - \{\alpha\}$ covers β. Hence, Γ is not minimal. □

Example 3.5 We use the Example 3.4. Γ is a minimal cover for σ_4. And $d0dd$ ($dd1d$, $dd00$, $d1d1$) has a unique minterm 0001 (0110, 0100, 0101, resp.). □

The concept of uniqueness can be extended to a term. Also we can define *redundancy* of a term in terms of the uniqueness.

Definition 3.13 A term $\alpha \in \Gamma$ is called *unique* with respect to Γ iff it contains a *unique minterm*. A term $\alpha \in \Gamma$ is called *redundant* with respect to Γ iff it is not *unique*. □

Example 3.6 In Example 3.4, every term of Γ is unique with respect to Γ. □

The following theorem justifies simplification based on projection called *projection reduction*, preserving the notion of minimal cover. The performance of *projection reduction* largely depends on the cardinality of β (The larger $|\beta|$, the more simplifications would take place). It reduces the number of input variables for a particular testing, which exponentially reduces the search space and would lead to a big simplification.

Theorem 3.4 (Projection Reduction) If $\Gamma = \{\alpha_1, \cdots, \alpha_m\}$ is a minimal cover for β, then $\Gamma^{\Pi}|_{\beta} = \{\alpha_1^{\Pi}|_{\beta}, \cdots, \alpha_m^{\Pi}|_{\beta}\}$ is a minimal cover for β^{Π}.

Proof: We first show that Γ^{Π} covers β^{Π}.

By the definition of minimal cover, we have,

(1) $\forall\Delta[\Delta$ is a proper subset of $\Gamma \rightarrow \exists t[t \in^c \beta, t \notin^c \Delta]]$, and

(2) $\forall t[t \in^c \beta \rightarrow \exists\alpha[\alpha \in \Gamma, t \in^c \alpha]]$.

Using the definition of projection and complementary projection, we get

(3) $\forall t[t \in^c \beta \rightarrow t^{-\Pi} = \beta^{-\Pi}, t^{\Pi} \in^c \beta^{\Pi}]$ and

(4) $\forall\alpha[\alpha \in \Gamma \rightarrow [\alpha^{\Pi} \in \Gamma^{\Pi}, \forall t[t \in^c \alpha \rightarrow t^{\Pi} \in^c \alpha^{\Pi}, t^{-\Pi} \in^c \alpha^{-\Pi}]]]$.

(2) and (4) implies that,

(5) $\forall t[t \in^c \beta \rightarrow \exists\alpha^{\Pi}[\alpha^{\Pi} \in \Gamma^{\Pi}, t^{\Pi} \in^c \alpha^{\Pi}]]$.

From (3) and (5), we derive,

$\forall t[t \in^c \beta \rightarrow t^{\Pi} \in^c \beta^{\Pi}, \exists\alpha^{\Pi}[\alpha^{\Pi} \in \Gamma^{\Pi}, t^{\Pi} \in^c \alpha^{\Pi}]]$, which implies $t^{\Pi} \in^c \beta^{\Pi} \rightarrow t^{\Pi} \in^c \alpha^{\Pi}$ for some $\alpha^{\Pi} \in \Gamma^{\Pi}$, provided that $t \in^c \beta$.

Hence,

(6) $\{\alpha_1^{\Pi}, \cdots, \alpha_m^{\Pi}\}$ covers β^{Π}.

Next, we show that no proper subset of Γ^{Π} covers β^{Π}.

Let $\Delta = \{\delta_1, \cdots, \delta_p\}$ be any proper subset of Γ. Let $\Delta^{\Pi} = \{\delta_1^{\Pi}, \cdots, \delta_p^{\Pi}\}$. Then, by the universal instantiation, (1) can be rewritten as

(7) $\exists t_1[t_1 \in^c \beta, t_1 \notin^c \delta_1, \cdots, t_1 \notin^c \delta_p]$.

Observation 3.3 (2) implies that for any $\delta \in \Delta$, δ is not disjoint with β. Then, $\delta^{-\Pi}$ is not disjoint with $\beta^{-\Pi}$, which further implies that

(8) $\beta^{-\Pi} \in^c \delta_1^{-\Pi}, \cdots, \beta^{-\Pi} \in^c \delta_p^{-\Pi}$.

From (3) and (8), we get,

(9) $\forall t[t \in^c \beta \rightarrow t^{\Pi} \in^c \beta^{\Pi}, t^{-\Pi} \in^c \delta_1^{-\Pi}, \cdots, t^{-\Pi} \in^c \delta_p^{-\Pi}]$.

(7) and (9) implies that $\exists t_1[t_1^{\Pi} \in^c \beta^{\Pi}, t_1^{\Pi} \notin^c \delta_1^{\Pi}, \cdots, t_1^{\Pi} \notin^c \delta_p^{\Pi}]$.

That is,

(10) for any proper subset Δ^{Π} of Γ^{Π}, $\exists t_1[t_1^{\Pi} \in^c \beta^{\Pi}, t_1^{\Pi} \notin^c \Delta^{\Pi}]$.

From (6), (10), and the definition of minimal cover, we conclude,

$\Gamma^{\Pi} = \{\alpha_1^{\Pi}, \cdots, \alpha_m^{\Pi}\}$ is a minimal cover for β^{Π}. □

Example 3.7 Let $\Gamma = \{ddd11, 1dd01, d00d1, 0d101, d100d\}$ and $\beta = dddd1$. Then, Γ is a minimal cover for β. By applying the projection reduction, we obtain, $\beta^{\Pi} = \sigma_4$, $\Gamma^{\Pi} = \{ddd1, 1dd0, d00d, 0d10, d100\}$ and Γ^{Π} is a minimal cover for β^{Π}. □

3.6 Mergeability

In this section, we establish the relationship between two concepts: merge and cover, concerning the existence of a merge. Next, we show a stronger result regarding the existence of a merge by taking the notion of *minimal cover* into consideration. This directly leads to a practical application to a special type of simplification called *nonmerge reduction*. This type of simplification would take place more often in a circuit design before minimization than in a minimized version.

To prove that Γ does not cover σ_n, it is enough to find a minterm such that it cannot be covered by each term of Γ. To do that, we design a falsification algorithm named *Algorithm-F* which seeks a counterexample by constructing such a falsifying minterm. Also we show the validity of Algorithm-F. The result obtained from the algorithm tells us the sufficient condition for a merge, using the concept of cover, which will be later extended to a stronger result in Theorem 3.5.

Lemma 3.2 Let Γ be a set of terms.

If (1) for each term $\alpha \in \Gamma$, $\alpha \neq \sigma_n$, and

(2) for each pair of terms $\alpha, \beta \in \Gamma$, α and β are not mergeable,

then Γ does not cover σ_n.

Proof: Construct a minterm $\hat{t}$ which is disjoint with each member of Γ as follows:

Algorithm-F:

Input: $\Delta_0 = \Gamma$, satisfying the antecedent.

Output: a minterm $\hat{t}$ which is disjoint with each member of Γ.

Procedure:

```
  $\hat{t}[1..n] \leftarrow 0$;        { initialize $\hat{t}$}
  $k \leftarrow 0$;                    {$k$ is a position of $\hat{t}$}
  while $\Delta_k$ is not empty do
      $k \leftarrow k + 1$;
      $A \leftarrow \{\alpha \,|\, \alpha \in \Delta_{k-1},\ \alpha[k] = 1\}$;
      if $A$ is not empty, then $\hat{t}[k] \leftarrow 0$;
      else   $A \leftarrow \{\alpha \,|\, \alpha \in \Delta_{k-1},\ \alpha[k] = 0\}$;
             $\hat{t}[k] \leftarrow 1$;
      $\Delta_k \leftarrow \Delta_{k-1} - A$;
  end.while;
  return $\hat{t}$;
end.procedure;
```

Correctness of the Algorithm-F:

The algorithm removes each member of A from Δ_{k-1} at each iteration whenever $A \neq \emptyset$. Thus as long as $A \neq \emptyset$, Δ_{k-1} will be reduced to Δ_k whose cardinality is less than that of Δ_{k-1}. Then, it is possible that Δ_k becomes $\emptyset$ at kth iteration $(1 \leq k < n)$. Hence, the algorithm terminates and returns $\hat{t}$. If $A = \emptyset$ at kth iteration $(1 \leq k < n)$ (i.e. $\alpha[k] = d$ for each member $\alpha \in \Gamma$ at kth iteration.), then we proceed k to the next position (i.e., $k + 1$).

Suppose we have just reached the last iteration, where $k = n$. Then, it is sufficient to show that $\Delta_n = \emptyset$. Suppose Δ_n is not empty. Then, there exists a term $\alpha \in \Delta_n$. Then, we conclude:

(1) There is no $j \leq n$ such that $\alpha[j] = 1$ because otherwise $\alpha \notin \Delta_j$ (α would be excluded at step j).

(2) There is some $j \leq n$ such that $\alpha[j] = 0$ because otherwise $\alpha = \sigma_n$, which is impossible.

So, let j be the last $j \leq n$ such that $\alpha[j] = 0$. Then, there is a term $\beta \in \Delta_{j-1}$ such that $\beta[j] = 1$. For all $k < j$, $\beta[j] \neq 1$, otherwise $\beta \notin \Delta_{j-1}$ because β would be excluded earlier. For all $j < k \leq n$, $\alpha[k] = d$. So, $\alpha[j] = 1 - \beta[j]$ and there is no other complementary positions except j. Then, α and β are mergeable. A contradiction. □

Example 3.8 Let $\Gamma = \{d0dd1d01d,\ 1dd0d1011,\ dd001dd1d,\ dd00ddd11,\ 1dd011ddd,\ 1ddddd011,\ 0dd11dd10,\ 0111d01dd\}$. Table 4 shows the trace of Algorithm-F on Γ for constructing the falsifying minterm $\hat{t}$. □

We now prove the next two Lemmas as preliminaries to Theorem 3.5, which establishes the existence of a merge for *each* term of Γ.

Lemma 3.3 If Γ is a cover for σ_n and $\sigma_n \notin \Gamma$, then there exists γ such that for some $\alpha, \beta \in \Gamma$, $\{\alpha, \beta\} \mapsto \gamma$.

Proof: Directly from Lemma 3.2. □

Lemma 3.4 Let α be a term of Γ.

If (1) α is not mergeable with any $\beta \in \Gamma$, and

(2) there is a minterm $t \in^c \breve{U}(\alpha)$, and

(3) there is a minterm $\hat{t}$ and a position l such that $\alpha[l] \neq d$, $\hat{t}[l] = 1 - t[l]$ and for each position $p(\neq l)$, $\hat{t}[p] = t[p]$,

then $\hat{t} \notin^c \Gamma$.

k	1	2	3	4	5
A	$1dd0d1011$ $1dd011ddd$ $1dddd d011$	$0111d01dd$	$dd001dd1d$ $dd00ddd11$	$0dd11dd10$	$d0dd1d01d$
$\hat{t}[k]$	0	0	1	0	0
Δ_k	$d0dd1d01d$ $dd001dd1d$ $dd00ddd11$ $0dd11dd10$ $0111d01dd$	$d0dd1d01d$ $dd001dd1d$ $dd00ddd11$ $0dd11dd10$	$d0dd1d01d$ $0dd11dd10$	$d0dd1d01d$	$\emptyset$

Table 4: Trace of the Algorithm-F for Example 3.8

Proof: From (1), one of the following relationships between α and β must be true:

- β and α are disjoint with distance ≥ 2.
- β and α are shared.

From (2) and (3), we obtain,

for each $\beta(\neq \alpha) \in \Gamma$, $t \not\in^c \beta$ and $\hat{t} \not\in^c \alpha$.

There are two cases to be considered as stated above:

Case 1. For each term $\beta(\neq \alpha) \in \Gamma$, β and α are disjoint with distance ≥ 2.

Then, $\hat{t} \not\in^c \beta$ because $\hat{t}$ and t are different at exactly one position, while t and β are disjoint with distance ≥ 2.

Case 2. For each term $\beta(\neq \alpha) \in \Gamma$, β and α are shared.

Recall that l is the only position in which $t[l]$ and $\hat{t}[l]$ are complementary. Then, $\hat{t}[l]$ and $\alpha[l]$ are complementary because $\alpha[l] = t[l]$. $\beta[l] = d$ or $\alpha[l]$ because β and α are shared.

Let k be a position where $t[k]$ and $\beta[k]$ are complementary.

If $\beta[l] = \alpha[l]$, then $\hat{t}[l]$ and $\beta[l]$ are complementary because $\hat{t}[l]$ and $\alpha[l]$ are complementary. Then, $\hat{t} \notin^c \beta$. If $\beta[l] = d$, then, $k \neq l$, and $\hat{t}[k] = t[k]$. Then, $\hat{t}[k]$ and $\beta[k]$ are complementary because $t[k]$ and $\beta[k]$ are complementary. Then, $\hat{t} \notin^c \beta$. Hence, $\hat{t} \notin^c \beta$ for each $\beta \in \Gamma$. This completes the proof. □

Theorem 3.5 **(Existence of a Merge)** If Γ is a minimal cover for σ_n and $\sigma_n \notin \Gamma$, then $\forall\alpha[\alpha \in \Gamma \rightarrow \exists\beta[\beta \in \Gamma, \beta \neq \alpha$, α and β are mergeable]].

Proof: Suppose that there exists a term $\alpha \in \Gamma$ such that, for each β $(\neq \alpha) \in \Gamma$, α and β are not mergeable. Since Γ is a minimal cover for σ_n, there exists a minterm $t \in^c \check{U}(\alpha)$.

Let $\hat{t}$ be the minterm resulting from taking the negation of $t[l]$ where l is any arbitrary position of α such that $\alpha[l]$ is either 0 or 1. Then, it immediately follows from Lemma 3.4 that $\hat{t} \notin^c \Gamma$. Then, Γ is not a cover for σ_n. A contradiction. □

Directly from Theorem 3.5, we obtain the following simplification rule called *nonmerge reduction.*

Theorem 3.6 **(Nonmerge Reduction)** Let $\hat{\Gamma} = \Gamma - N$, where N is the set of all nonmergeable terms. Then, $\sigma_n \in^c \Gamma$ iff $\sigma_n \in^c \hat{\Gamma}$.

Proof: $(\Longrightarrow)$ $\sigma_n \in^c \Gamma$ implies that there exists a subset Γ' of Γ such that Γ' is a minimal cover for σ_n (Theorem 3.3). Then, for each $\alpha \in \Gamma'$, there exists a term $\beta \in \Gamma'$ such that α and β are mergeable (Theorem 3.5). But, $\Gamma' \subseteq \hat{\Gamma}$. Then, it follows that $\sigma_n \in^c \hat{\Gamma}$.

$(\Longleftarrow)$ Obvious. □

3.7 Losslessness

In this section, we consider the effects of the merge operation which might lose unique minterms of the base set of terms from which a merge can be derived. We also consider how a lost minterm can be recovered.

A *lossless merge* is a special kind of merge which covers every unique minterm of the base set of terms, defined as follows:

Definition 3.14 γ is said to be a *lossless merge* iff for some $\alpha, \beta \in \Gamma$, (1) $\{\alpha, \beta\} \mapsto \gamma$, and (2) γ contains $\check{U}(\alpha) \cup \check{U}(\beta)$. (i.e. No unique minterm of α or β is lost as the result of the merge operation.) γ is *lossy* if it is not *lossless* . □

Example 3.9 (1) $ddd0$ is the merge of two terms $dd1d$ and $dd00$ in Example 3.4. Then, the merged term is *lossless* because $\check{U}(dd1d) = \{0110, 1110\}$, $\check{U}(dd00) = \{0100, 1100\}$.

(2) Let $\Gamma = \{dd11, ddd0, 00d1, 01dd, 1d0d\}$. Then, $\{dd11, ddd0\} \mapsto dd1d$. The merged term is *lossy* because 0000 is lost because of the merge. □

A *simple merge*, by its definition, always covers the original set of base terms. Thus, a *simple merge* γ of α and β is lossless because $\alpha \cup \beta \equiv \gamma$, but not vice versa. Because of this favorable characteristic, a *simple merge* has higher priority than any other types of merge so that it will be performed before any other merges except unit merges.

The following Lemma shows that replacing α, β by γ preserves the notion of cover when $\{\alpha, \beta\} \mapsto \gamma$ is lossless.

Lemma 3.5 Let Γ be a minimal cover for σ_n and for some $\alpha, \beta \in \Gamma$, $\{\alpha, \beta\} \mapsto \gamma$. Let $\Gamma' = \Gamma - \{\alpha, \beta\} + \{\gamma\}$. Then, γ is lossless iff Γ' is a minimal cover for σ_n.

Proof: The lemma may be proved by a trivial contradiction. □

In case of a *lossy merge*, we first need to decide whether or not lost information can be reclaimed. And then, if it is possible, we want to figure out how lost minterms can be recovered. Theorem 3.7 addresses these issues with the help of supporting

Lemmas 3.6 and 3.7.

Lemma 3.6 Let α, β be two terms in a set of terms Γ. If a minterm $\acute{t} \in^c \alpha$ is lost in $\{\alpha, \beta\} \mapsto \gamma$, then, there is a position k such that $\alpha[k] = d$ and $\beta[k] = \gamma[k] = 1 - \acute{t}[k]$.

Proof: Since $\acute{t} \notin^c \gamma$, there is a position k such that $\gamma[k]$ is the negation of $\acute{t}[k]$. The fact that $\acute{t}$ is a minterm of α implies,

For each position p, $\acute{t}[p] \in^c \alpha[p]$. That is, $\alpha[p] = \acute{t}[p]$ or d.

However, $\alpha[k] \neq \acute{t}[k]$ because $\gamma[k] \neq 1 - \alpha[k]$. So, $\alpha[k] = d$. Then, it follows immediately from the Merge Rule that $\beta[k] = \gamma[k]$. □

In the next lemma, we show that a minterm lost as the result of a lossy merge can always be recovered by another merge involving the term which contains the lost minterm.

Lemma 3.7 If (1) Γ be a cover for σ_n, and

(2) every merge for any two mergeable terms of Γ is lossy, and

(3) α, β are mergeable terms of Γ and $\acute{t} \in^c \alpha$ is a *unique minterm* which is lost in the merge of α and β,

then there is another term $\xi(\neq \beta) \in \Gamma$ such that $\{\alpha, \xi\} \mapsto \gamma$ and $\acute{t} \in^c \gamma$.

Proof: Assume the hypothesis and the negation of the conclusion. Let Δ be the set of all terms of Γ each of which member is mergeable with α. Let Θ be $\{\gamma \mid \{\alpha, \xi\} \mapsto \gamma,\ \xi \in \Delta\}$. Then, by the negation of the conclusion, $\acute{t} \notin^c \gamma$ for each $\gamma \in \Theta$. Let $\hat{t}(\in \sigma_n)$ be the minterm just like $\acute{t}$ except that $\hat{t}[p] = 1 - \acute{t}[p]$, where p is the merge position of α and β. Then, $\hat{t} \notin^c \alpha$, since $\hat{t}[p] = 1 - \alpha[p]$. From Lemma 3.4, it immediately follows that

(a) $\hat{t} \notin^c (\Gamma - \Delta)$.

Now, we show that for each $\xi \in \Delta$, $\hat{t} \notin^c \xi$. For each $\xi(\neq \beta) \in \Delta$, by the definition of Δ, there is a merge γ such that $\{\alpha, \xi\} \mapsto \gamma$. And, by the assumption, $\acute{t} \notin^c \gamma$.

Then, by Lemma 3.6, there is a position k such that $\alpha[k] = d$, $\xi[k] = \gamma[k] = 1 - \acute{t}[k]$, but $\hat{t}[k] = \acute{t}[k]$ because $k \neq p$. So, $\xi[k] = 1 - \hat{t}[k]$. Then,

(b) $\hat{t} \not\in^c \xi$.

From (a) and (b), we conclude that $\hat{t} \not\in^c \Gamma$, but this contradicts the hypothesis. □

Theorem 3.7 is a natural extension to the previous lemma, which extends to the recovery of each lost minterm caused by lossy merges.

Theorem 3.7 (Recovery) Let Γ be a minimal cover for σ_n. Let $\gamma_1, \cdots, \gamma_y$ be all possible distinct merges in one step from Γ. If each merge is lossy, then $\{\gamma_1, \cdots, \gamma_y\}$ covers $\check{U}(\alpha_1) \cup \cdots \check{U}(\alpha_m)$.

Proof: From Theorem 3.5, for each term $\alpha \in \Gamma$, there is a mergeable term with α. Since each merge is lossy, there is a unique minterm lost in each merge. Then, from Lemma 3.7, any unique minterm lost is recovered by $\{\gamma_1, \cdots, \gamma_y\}$. □

The significance of the above theorem is not actual implementation of the recovery because it might lead to an exhaustive search, in general, for right merges which would recover lost minterms, but the validity of the derivational method which will eventually recover every lost minterm. In Chapter 4, we propose an algorithm along with a set of heuristics which obviates the recovery process and yet achieves the goal efficiently.

CHAPTER 4

THE COVER-MERGE ALGORITHM

We introduce the Merge-Loop which is a realization of the derivational approach discussed in Chapter 3. The significance of the algorithm lies in the establishment of the equivalence of the two concepts: cover and derivation. Although the derivational method is valid, it may contain many redundancies and unnecessary operations. Hence, we devise a set of heuristics and simplification rules which would shorten proof procedures in most cases. Heuristics are mainly focused on how potentially favorable merges can be identified and how they can be chosen for next merge operation. Priorities are imposed on each of the heuristics so that most favorable heuristic can be selected, and consequently, a shorter proof can be obtained.

Suppose $\eta \notin^c \Gamma$ for a term η and a set of terms Γ. To test whether or not Γ covers η, the Merge-Loop would try to continually search for $\tilde{\eta}$ (or a generalization of η) until no more merges can newly be generated because the basic structure of the algorithm is designed to prove $\eta \in^c \Gamma$ by showing the existence of a derivation $\Gamma \mapsto \tilde{\eta}$. This indicates that exhaustive searches are unavoidable if the problem is to disprove $\eta \in^c \Gamma$. However, we improve the situation significantly by employing a cost-effective heuristic which would predict the outcome of a cover test and initiate falsification heuristics to try to disprove it if a certain condition for the heuristic is met. We will discuss and analyze these heuristics in this chapter.

Simplification is the most important aspect of equivalence testing as far as running time is concerned. Three different types of simplifications are extensively used for the Merge-Loop: *covered term reduction*, *nonessential term reduction*, and *non-*

and *nonmerge reduction*. Most of the simplifications during the Merge-Loop are performed by removing covered terms due to forward merges. Other types of simplifications are introduced to strengthen simplification process so that simplifications take place in a wider range of conditions and situations.

We propose the final version of the Merge-Loop incorporating heuristics and simplification rules developed so far. Finally, the top level algorithm Cover-Merge for the equivalence problem is presented and discussed.

4.1 The Merge-Loop and Proof

This section contains the Merge-Loop and its proof which establishes the equivalence of the two concepts: cover and derivation. The main purpose of the Merge-Loop is to decide whether or not $\eta \in^c \Gamma$ for a term η and a set of terms Γ. In case of $\eta \notin^c \Gamma$, it may return a counterexample which disproves $\eta \in^c \Gamma$ (Merge_Loop_2 in Section 4.5). The Merge-Loop is invoked by the Algorithm Cover, which is in turn invoked by the top level Cover-Merge Algorithm. The relationship between the Algorithm Cover and the Merge-Loop will be discussed in later sections.

Any method involving exhaustive search cannot be a solution for this algorithm because it would never give us a result within a reasonable amount of time, although the method is theoretically valid. Therefore, we first present a valid (or sound and complete) method for the cover testing problem and then we propose a set of heuristics in order to allow more simplifications and speed up the process.

Since the number of possible merges in a set of terms Γ can be quite big ($O(m^2)$, where $m = |\Gamma|$), it is crucial to generate only those merges that can be useful in the derivation process. It is not a trivial problem, in general, to determine which merge would be good and which would be bad for future use. Such a test should be fast because it should be made for each merge generated. Even if an exact prediction may not be possible, a good estimate can guide the merge process much efficiently. First, we define two types of merges: *forward* and *backward* merges.

Definition 4.1 A merge, $\{\alpha, \beta\} \mapsto \gamma$, is called

(1) a *forward merge* if γ covers α or β (or both).

(2) a *backward merge* if it is not *forward* and $I(\alpha, \beta) \geq 3$.[1] □

Obviously, a forward merge can be viewed as a simplification. So, it should be given a higher priority than backward merges. There are some merges which are neither forward nor backward. Fortunately, this does not happen very often, compared with backward merges. Thus, storing these merges for future use may not cause much overhead to a derivation process and in many cases it allows the production of new favorable merges in subsequent steps. Backward merges are usually considered as not helpful in a derivation. Some backward merges are useful for subsequent merges. However, there are many cases where backward merges are immediately useless, which are recognized as *degenerating.*

Example 4.1 Let $\Gamma = \{q, r, s, u, v, w, x, y, z\}$, where $q(r, s, u, v, w, x, y, z)$ is $d0d01$ ($1d011$, $d11d1$, $01ddd$, $ddd00$, $0d01d$, $1dd10$, $d011d$, $d1d0d$, resp.). Table 5 illustrates the set of all possible forward and backward merges from Γ in one step. Most of the merges generated are found to be backward, which is a typical phenomenon for most PLAs. Some merges are neither forward nor backward. In this example, they are $\{q, y\}, \{s, y\}, \{w, x\}$, and $\{w, y\}$. □

Definition 4.2 A merge γ of two terms from Γ is said to be a *nondegenerating merge* if γ is not covered by any term of Γ. Otherwise, γ is called a *degenerating merge.* □

[1] $I(\alpha, \beta)$ denotes the number of inclusive positions of α and β (Definition 3.4).

Types of Merges	*Merges*
backward	$\{q,r\},\{q,s\},\{q,u\},\{q,w\},\{r,s\},$ $\{r,u\},\{r,y\},\{r,z\},\{s,v\},\{s,w\},$ $\{s,x\},\{u,x\},\{u,y\},\{v,w\},\{v,y\},$ $\{w,z\},\{x,z\}$
forward	$\{q,v\},\{q,z\},\{r,w\},\{r,x\},\{v,x\}$

Table 5: Types of Merges for Example 4.1

Example 4.2 Merges $\{s,v\},\{s,w\}$, and $\{w,z\}$ are degenerating in Example 4.1. □

If a merge is found to be degenerating, then it is immediately useless for the verification. The Merge-Loop ignores each degenerating merge occurred during a derivation, justified by the following observation which shows that (1) if a term α is degenerating and there is a term η such that $\{\alpha,\eta\} \mapsto \gamma$, then γ is always covered by another term or merge, and (2) if there exists no mergeable term η, then, by the nonmerge reduction (Theorem 3.6), α must be eliminated immediately.

Observation 4.1 A covered term or a *degenerating merge* is redundant for an efficient merge process.

Proof: Let α,β be two terms and $\alpha \in^c \beta$. Suppose there exists a term η such that $\{\alpha,\eta\} \mapsto \gamma_1$. There are two cases to be considered:

Case 1. η and β are not mergeable.

Let p be an inclusive position such that $\alpha[p] \neq d, \beta[p] = d$. Let q be a common literal position of α and β. Let r be a position where $\alpha[r] = \beta[r] = d$, if any. Then, for each position w, if $\beta[w] \neq d$, then $\alpha[w] = \beta[w]$ because $\alpha \in^c \beta$. Obviously, MP(α, η) cannot be a common literal position of α and β.[2] Then, MP(α, η) must be an inclusive position of α and β. Let $p_1 = \mathrm{MP}(\alpha, \eta)$ Then,

(a) $\gamma_1[p_1] = \beta[p_1] = d$.

(b) $\forall w[w \neq p_1,\ \alpha[w] \neq d \Rightarrow \gamma_1[w] = \alpha[w]]$.

(c) $\forall r[\alpha[r] = \beta[r] = d \Rightarrow \gamma_1[r] = \eta[r]$.

(d) If there is another inclusive position of α and β, p_2, other than p_1 such that $\alpha[p_2] \neq d, \beta[p_2] = d$, then $\gamma_1[p_2] = \eta[p_2]$.

From (a), (b), (c) and (d), we observe,

$\forall j[[\gamma_1[j] = d \Rightarrow \beta[j] = d]$ and $[\gamma_1[j] \neq d \Rightarrow \beta[j] = d$ or $\gamma_1[j]]]$.

Hence, $\gamma_1 \in^c \beta$.

Case 2. η and β are mergeable.

Let $\{\beta, \eta\} \mapsto \gamma_2$ and x be the merge position. Then, $\beta[x] = 1 - \eta[x] \neq d$, which implies that $\alpha[x] = \beta[x]$ because $\alpha \in^c \beta$. Let y be MP(α, η). Suppose $x \neq y$. Then, $\alpha[x] = 1 - \eta[x] \neq d$. Then, α and η cannot be merged, which is a contradiction. Hence, $x = y$. Then, obviously, $\alpha \in^c \beta \Rightarrow \gamma_1 \in^c \gamma_2$. □

So, there is no point holding degenerating merges or covered terms for the Merge-Loop. We now define the basic structure of the Merge-Loop whose current *merge set* is defined recursively by its previous set.

Definition 4.3 Let Γ^0 be a set of terms. Let $M^{k+1} = \{\gamma \mid \{\alpha, \beta\} \mapsto \gamma$, $\alpha, \beta \in \Gamma^k$, $\gamma \notin \Gamma^k$, γ is a *nondegenerating merge*$\}$ and $\Gamma^{k+1} = \Gamma^k \bigcup M^{k+1}$, for $k \geq 0$. M^j is called the jth *merge set* from Γ^0, which is the set of all new merges from Γ^0 at the jth *merge step.*

We define the *closure* of merges from Γ^0 as follows:

$M^* = M^0 \cup M^1 \cup \cdots \cup M^j$, where $M^{j+1} = \emptyset$. □

[2]MP(α, β) denotes the merge position of α, β. Definitions of inclusive position, common literal position, and merge position are found in Definition 3.4.

Suppose we have a derivation of a term γ from a set of terms. Then, we observe that many nondegenerating merges from the set are redundant with respect to the derivation. Definition 4.4 clarifies this observation.

Definition 4.4 Let Δ be a successful derivation of a term γ from a set of terms Γ. Then, there exists a finite sequence of merges whose last merge is γ. The *nonredundant derivation* of γ, denoted by Δ', is a derivation such that

(1) the last term in Δ' is γ, and

(2) If $\{\alpha, \beta\} \mapsto \eta$ for some α, β in Δ and η in Δ', then both α and β are included in Δ' and they appear before η in the sequence.

(3) Nothing is in Δ' unless it follows from (1) and (2).

If a term α in Δ appears in Δ', then α is said to be *essential* to Δ'. Otherwise, α is *nonessential* to Δ'. $\square$

The essentiality of a merge is relative to a nonredundant derivation. In other words, a merge can be essential to one derivation, but at the same time it could be nonessential to another because there can be many nonredundant derivations for a given goal term.

Definition 4.5 β is a *generalization* of α iff β is the result of replacing zero or more $0's$ or $1's$ in α by $d's$ iff for each position p, $\beta[p] = \alpha[p]$ or $\beta[p] = d$.

A term α is said to be *derivable* from a set of terms Γ iff there is a derivation of a generalization of α from Γ. $\square$

4.1.1 The Merge-Loop

In this section, we propose the Merge_Loop_1 and provide its proof. The main purpose of this section is to establish the relationship:

$\eta \in^c \Gamma \Longleftrightarrow \Gamma \mapsto \tilde{\eta}$, for any term η ($0 \leq |\eta| \leq n$, $\tilde{\eta}$ is a generalization of η).

We will refine the Merge_Loop_1 and produce a better version, Merge_Loop_2, by removing redundancies and employing heuristics to shorten the merge process.

Algorithm Merge_Loop_1;

Input: a set, Γ, of terms and a term η.

Output: True if Γ covers η. False, otherwise.

Procedure:

```
Let $\Gamma^0 = \Gamma$.
$k \leftarrow 0$;
while $\tilde{\eta} \notin \Gamma^k$ ($\tilde{\eta}$ is a generalization of $\eta$) do
    Let $\gamma^k$ be a nondegenerating merge of two terms from $\Gamma^k$.
    if $\gamma^k$ is not found, return False;
    Let $\Gamma^{k+1} = (\Gamma^k - \{\text{term(s) covered by } \gamma^k\}) \bigcup \{\gamma^k\}$.
    $k \leftarrow k + 1$;
end.while;
return True;
end.procedure;
```

It is obvious that the Merge_Loop_1 terminates because

(i) only nondegenerating merges can be added to the system,

(ii) a nondegenerating merge cannot be added more than once because of (i), and

(iii) the set of nondegenerating merges is finite.

Theorem 4.1 (Merge_Loop_1) The Merge_Loop_1 returns True iff $\eta \in^c \Gamma$.

Proof:

Part A. Soundness Proof:

If η is derivable from Γ^0 by the Merge_Loop_1, then Γ^0 covers η.

We prove the following theorem by induction on the length l of a derivation of $\tilde{\eta}$ from

Γ^0, where $\tilde{\eta}$ is a generalization of η:

If there is a derivation of $\tilde{\eta}$ from Γ^0, then Γ^0 covers η.

If $l = 0$, then the theorem immediately follows from $\eta \in \Gamma^0$. Suppose the theorem holds for $l = k - 1$ ($k \geq 1$). If $l = k$ and $\{x, y\} \mapsto \tilde{\eta}$, then, by the hypothesis, Γ^0 covers both x and y. Then, by Observation 3.1 (1), Γ^0 covers $\tilde{\eta}$.

Part B. Completeness Proof:

If Γ^0 *covers* η*, then* η *is derivable from* Γ^0 *by the Merge_Loop_1.*

We prove the following theorem by induction on l (the number of $d's$ in η):

If η contains exactly l $d's$, and Γ^0 covers η, then η is derivable from Γ^0, ($\eta \notin \Gamma^0, 0 \leq l \leq n$).

1. *Basis:* $l = 0$. Then, there is an immediate derivation $\Gamma^0 \mapsto \tilde{\eta}$, where $\tilde{\eta}$ is a *generalization* of η.

2. *Hypothesis:* If η contains exactly $(l-1)d's$ and Γ^0 covers η, then η is derivable from Γ^0.

3. *Induction Step:* Let $r_1, \cdots, r_l$ be the positions where d appears in η. Consider two terms β and α such that they are exactly like η except $\beta[r_j] = 1 - \alpha[r_j] \neq d$ for any $1 \leq j \leq l$. Then, by the hypothesis, β (α) is derivable from Γ^0 if $\eta \in^c \Gamma^0$. Let $\tilde{\beta}$ ($\tilde{\alpha}$) be a generalization of β (α), derived from Γ^0. Then, if $\tilde{\beta}$ or $\tilde{\alpha}$ is a generalization of η, then the theorem immediately follows. If not, $\tilde{\beta}[r_j] = 1 - \tilde{\alpha}[r_j] \neq d$. Then, $\{\tilde{\beta}, \tilde{\alpha}\} \mapsto \tilde{\eta}$ and $\tilde{\eta}$ is a generalization of η. □

The theorem shows that cover testing can be performed by applying the Merge-Loop algorithm using the derivational method. But we want to point out that η can be any term including a complete term. If η is a complete term, then the Merge-Loop conducts a special type of cover testing, namely ***tautology checking.*** In reality, η is indeed a complete term particularly for our cover testing problem because η is the result of performing projection operation on a term β for cover testing. Because of that, we now assume, without loss of generality, that an input term to the Merge-Loop for a cover testing is a complete term. And we often use a '*goal term*' or 'σ_τ' to refer to a complete term for cover testing.

Example 4.3 Let $\Gamma^0 = \{s, u, v, w, x, y, z\}$, where s (u, v, w, x, y, z) is $d11d1$ ($01ddd$, $ddd00$, $0d01d$, $dd0d1$, $d01dd$, $1dd10$, resp.). Trace of Merge_Loop_1 for the derivation of σ_5 is shown below (Table 6). Direct application of the algorithm requires 34 steps to complete the process, while Merge_Loop_2 (Section 4.5) employing a set of heuristics to shorten the length of derivation takes only 11 steps (Table 7). □

k	Γ^k	γ^k	*Covered Terms*
0	$\{s,u,v,w,x,y,z\}$	$\{s,v\} \mapsto \gamma^0$ $(d110d)$	$\emptyset$
1	$\Gamma^0 \cup \{\gamma^0\}$	$\{s,x\} \mapsto \gamma^1$ $(d1dd1)$	s
2	$(\Gamma^1 - \{s\}) \cup \{\gamma^1\}$	$\{u,y\} \mapsto \gamma^2$ $(0d1dd)$	$\emptyset$
3	$\Gamma^2 \cup \{\gamma^2\}$	$\{u,z\} \mapsto \gamma^3$ $(d1d10)$	$\emptyset$
4	$\Gamma^3 \cup \{\gamma^3\}$	$\{v,w\} \mapsto \gamma^4$ $(0d0d0)$	$\emptyset$
5	$\Gamma^4 \cup \{\gamma^4\}$	$\{v,x\} \mapsto \gamma^5$ $(dd00d)$	$\emptyset$
6	$\Gamma^5 \cup \{\gamma^5\}$	$\{v,z\} \mapsto \gamma^6$ $(1ddd0)$	z
7	$(\Gamma^6 - \{z\}) \cup \{\gamma^6\}$	$\{v,\gamma^1\} \mapsto \gamma^7$ $(d1d0d)$	γ^0
8	$(\Gamma^7 - \{\gamma^0\}) \cup \{\gamma^7\}$	$\{v,\gamma^3\} \mapsto \gamma^8$ $(d1dd0)$	γ^3
9	$(\Gamma^8 - \{\gamma^3\}) \cup \{\gamma^8\}$	$\{w,y\} \mapsto \gamma^9$ $(00d1d)$	$\emptyset$
10	$\Gamma^9 \cup \{\gamma^9\}$	$\{w,\gamma^2\} \mapsto \gamma^{10}$ $(0dd1d)$	w,γ^9
11	$(\Gamma^{10} - \{w,\gamma^9\}) \cup \{\gamma^{10}\}$	$\{x,y\} \mapsto \gamma^{11}$ $(d0dd1)$	$\emptyset$
12	$\Gamma^{11} \cup \{\gamma^{11}\}$	$\{x,\gamma^2\} \mapsto \gamma^{12}$ $(0ddd1)$	$\emptyset$
13	$\Gamma^{12} \cup \{\gamma^{12}\}$	$\{x,\gamma^4\} \mapsto \gamma^{13}$ $(0d0dd)$	γ^4
14	$(\Gamma^{13} - \{\gamma^4\}) \cup \{\gamma^{13}\}$	$\{x,\gamma^6\} \mapsto \gamma^{14}$ $(1d0dd)$	$\emptyset$

15	$\Gamma^{14} \cup \{\gamma^{14}\}$	$\{x, \gamma^{8}\} \mapsto \gamma^{15}$ $(d10dd)$	$\emptyset$
16	$\Gamma^{15} \cup \{\gamma^{15}\}$	$\{y, \gamma^{1}\} \mapsto \gamma^{16}$ $(dd1d1)$	$\emptyset$
17	$\Gamma^{16} \cup \{\gamma^{16}\}$	$\{y, \gamma^{5}\} \mapsto \gamma^{17}$ $(d0d0d)$	$\emptyset$
18	$\Gamma^{17} \cup \{\gamma^{17}\}$	$\{y, \gamma^{7}\} \mapsto \gamma^{18}$ $(dd10d)$	$\emptyset$
19	$\Gamma^{18} \cup \{\gamma^{18}\}$	$\{y, \gamma^{8}\} \mapsto \gamma^{19}$ $(dd1d0)$	$\emptyset$
20	$\Gamma^{19} \cup \{\gamma^{19}\}$	$\{y, \gamma^{13}\} \mapsto \gamma^{20}$ $(00ddd)$	$\emptyset$
21	$\Gamma^{20} \cup \{\gamma^{20}\}$	$\{y, \gamma^{14}\} \mapsto \gamma^{21}$ $(10ddd)$	$\emptyset$
22	$\Gamma^{21} \cup \{\gamma^{21}\}$	$\{\gamma^{1}, \gamma^{6}\} \mapsto \gamma^{22}$ $(11ddd)$	$\emptyset$
23	$\Gamma^{22} \cup \{\gamma^{22}\}$	$\{\gamma^{1}, \gamma^{8}\} \mapsto \gamma^{23}$ $(d1ddd)$	$u, \gamma^{1}, \gamma^{7}, \gamma^{8}, \gamma^{15}, \gamma^{22}$
24	$(\Gamma^{23} - \{u, \gamma^{1}, \gamma^{7}, \gamma^{8}, \gamma^{15}, \gamma^{22}\}) \cup \{\gamma^{23}\}$	$\{\gamma^{2}, \gamma^{5}\} \mapsto \gamma^{24}$ $(0dd0d)$	$\emptyset$
25	$\Gamma^{24} \cup \{\gamma^{24}\}$	$\{\gamma^{2}, \gamma^{13}\} \mapsto \gamma^{25}$ $(0dddd)$	$\gamma^{2}, \gamma^{10}, \gamma^{12}, \gamma^{13}, \gamma^{20}, \gamma^{24}$
26	$(\Gamma^{25} - \{\gamma^{2}, \gamma^{10}, \gamma^{12}, \gamma^{13}, \gamma^{20}, \gamma^{24}\}) \cup \{\gamma^{25}\}$	$\{\gamma^{5}, \gamma^{16}\} \mapsto \gamma^{26}$ $(ddd01)$	$\emptyset$
27	$\Gamma^{26} \cup \{\gamma^{26}\}$	$\{\gamma^{5}, \gamma^{18}\} \mapsto \gamma^{27}$ $(ddd0d)$	$v, \gamma^{5}, \gamma^{17}, \gamma^{18}, \gamma^{26}$
28	$(\Gamma^{27} - \{v, \gamma^{5}, \gamma^{17}, \gamma^{18}, \gamma^{26}\}) \cup \{\gamma^{27}\}$	$\{\gamma^{6}, \gamma^{16}\} \mapsto \gamma^{28}$ $(1d1dd)$	$\emptyset$
29	$\Gamma^{28} \cup \{\gamma^{28}\}$	$\{\gamma^{6}, \gamma^{25}\} \mapsto \gamma^{29}$ $(dddd0)$	γ^{6}, γ^{19}
30	$(\Gamma^{29} - \{\gamma^{6}, \gamma^{19}\}) \cup \{\gamma^{29}\}$	$\{\gamma^{11}, \gamma^{23}\} \mapsto \gamma^{30}$ $(dddd1)$	$x, \gamma^{11}, \gamma^{16}$
31	$(\Gamma^{30} - \{x, \gamma^{11}, \gamma^{16}\}) \cup \{\gamma^{30}\}$	$\{\gamma^{14}, \gamma^{25}\} \mapsto \gamma^{31}$ $(dd0dd)$	γ^{14}
32	$(\Gamma^{31} - \{\gamma^{14}\}) \cup \{\gamma^{31}\}$	$\{\gamma^{21}, \gamma^{23}\} \mapsto \gamma^{32}$ $(1dddd)$	$\gamma^{21}, \gamma 28$
33	$(\Gamma^{32} - \{\gamma^{21}, \gamma^{28}\}) \cup \{\gamma^{32}\}$	$\{\gamma^{25}, \gamma^{32}\} \mapsto \gamma^{33}$ $(ddddd)$	Γ^{33}

Table 6: Trace of the Merge_Loop_1 for Example 4.3

The bottleneck of the Merge_Loop_1 is the merge process which blindly generates all *nondegenerating merges* without knowing the efficacy of such merges for subsequent merges toward a goal term. It is interesting to observe that only 13 merges are essential to the derivation of σ_5 in the above example, indicating many redundancies are involved. Table 7 illustrates the sequence of essential merges. Removing redundant merges is essential to speed up the derivation process especially for big PLAs. The focalization of next section will be mainly on how these redundancies can be eliminated.

No.	*Merge*
1	$\{s, w\} \mapsto \gamma^1 \ (= d1dd1)$
2	$\{u, y\} \mapsto \gamma^2 \ (= 0d1dd)$
3	$\{u, z\} \mapsto \gamma^3 \ (= d1d10)$
4	$\{v, w\} \mapsto \gamma^4 \ (= 0d0d0)$
5	$\{v, z\} \mapsto \gamma^6 \ (= 1ddd0)$
6	$\{v, \gamma^3\} \mapsto \gamma^8 \ (= d1dd0)$
7	$\{x, \gamma^4\} \mapsto \gamma^{13} \ (= 0d0dd)$
8	$\{x, \gamma^6\} \mapsto \gamma^{14} \ (= 1d0dd)$
9	$\{y, \gamma^{14}\} \mapsto \gamma^{21} \ (= 10ddd)$
10	$\{\gamma^1, \gamma^8\} \mapsto \gamma^{23} \ (= d1ddd)$
11	$\{\gamma^2, \gamma^{13}\} \mapsto \gamma^{25} \ (= 0dddd)$
12	$\{\gamma^{21}, \gamma^{23}\} \mapsto \gamma^{32} \ (= 1dddd)$
13	$\{\gamma^{25}, \gamma^{32}\} \mapsto \gamma^{33} \ (= ddddd)$

Table 7: Sequence of Essential Merges for the Derivation of Example 4.3

4.2 Heuristics for the Merge-Loop

Although the Merge_Loop_1 guarantees correct answers, it might contain redundant merges and unnecessary operations, which would lead to poor performance. To enhance the algorithm, we propose a set of *prioritized heuristics* which can significantly reduce redundant computations in most cases. Since the order of merges in a derivation greatly affects the performance of the merge process, the prioritization of these heuristics may provide a speed-up of the merge process and more simplifications. In this section, we use 'σ_r', a complete term with r d's, to denote a goal term for our cover testing problem.

Heuristics from H1 to H6 are incorporated into the algorithm to speed up the Merge-Loop approaching σ_r, while H7 is to disprove the existence of a derivation of σ_r by calling fast falsification heuristics to produce a counterexample if the input seems to be a nontautology. Heuristics are ordered by priorities in which H1 has the highest priority and H7 has the least. Each heuristic is tested in this order whether or not its condition is met at each iteration of the Merge-Loop. If so, we put the heuristic into action. When a heuristic has already been in action and yet another decision has to be made regarding the selection of a heuristic, the algorithm will recursively apply the priority rule until a choice is made.

H1. Perform *unit merges* before any other merges are tried. If a unit term is found, unit merges are performed throughout the input. Since a unit merge always reduces the number of literals of the terms involved, it is treated as a simplification called *unit-merge reduction*. Another strong point is that a unit term can easily be identified.

H2. Perform *simple merges*. A simple merge is generally considered as a simplification that can immediately eliminate its base terms. We name this type of simplification as *simple-merge reduction.* It also would lead to big simplifications in many cases.

H3. Experiments on a large scale of data show that terms which have participated in forward merges are more likely to trigger off other forward merges than terms involved in other types of merges. A forward merge must be exploited to generate as many forward merges as possible because it is highly likely that a cascade of new forward merges can be subsequently generated and big simplifications can be achieved. Since a forward merge always simplifies at least one of its base terms, it is considered as a simplification called *forward-merge reduction.* If two or more forward merges are found, apply H4 to choose the best one.

H4. If γ_1 and γ_2 are both forward merges such that $\{\alpha, \beta\} \mapsto \gamma_1$, $\{\alpha, \eta\} \mapsto \gamma_2$, and $I(\alpha, \beta) < I(\alpha, \eta)$, then choose γ_1 before γ_2.

H5. Merge terms α and β before ζ and η if $Lit(\alpha) + Lit(\beta) < Lit(\zeta) + Lit(\eta)$. Terms containing small number of literals are more likely to produce big merges which would allow more simplifications than terms with large number of literals.

H6. Postpone backward merges, if possible. Since backward merges increase the number of literals as the result of the operation, they are less likely to be involved in a successful derivation as essential merges. However, there are some cases in which backward merges are essential to a derivation sequence.

H7. If only backward merges are possible in Γ^k, for any $k \geq 0$, then it is likely that $\sigma_r \notin^c \Gamma^k$. The heuristic is based on the fact that if $\{\alpha, \beta\} \mapsto \gamma$ is a *backward merge*, then (1) γ realizes only a portion of $\alpha \cup \beta$ and (2) $Lit(\gamma)$ is increased as the result of the merge. Note that any successful sequence of merges which can lead to σ_r can be recognized as the process of replacing literals by d's until no more replacement is possible. We can also observe that merges along a successful sequence of derivation are most likely to reduce the number of literals of terms along the sequence. Since backward merges can only increase the number of literals along a merge sequence, we need to avoid them if they are redundant (Experiments show that they are redundant indeed in most cases). We propose two heuristic approaches to implement H7 in Section 4.3, based on falsification procedures.

Unit merges are extremely valuable for simplification, but they rarely occur at the initial stage of a derivation unless input is very sparse, especially for random PLAs. As the process approaches a goal term in a successful derivation, more and more simplifications would take place and input gets very sparse. So, we very often observe lots of unit merges near the end of successful derivations. Also, simple merges and forward merges are rarely found at the initial stage, but they are likely to occur as the process approaches a goal term in a successful derivation. The heuristic H5 is most frequently found at the initial stage and usually generates more favorable merges in subsequent steps.

Heuristics presented in this section are incorporated into the Merge-Loop and tested successfully. Experimental results on standard benchmark problems as well as on random PLAs are shown and discussed in Chapter 6.

Example 4.4 Table 8 demonstrates some of the heuristics on the same input for Example 4.3. Note that the first few heuristics help the process to quickly reach a unit term which would start a series of unit merges. Also note that H1 is heavily used in this example. This is a typical phenomenon which can be found near the end of every successful derivation because merged terms become quite sparse as the derivation approaches near σ_r and unit terms would appear more frequently near the end of a successful derivation of σ_r than any other stages of the derivation. Because of the heuristics used, we are able to shorten the length of the derivation from 34 to

11. The reduction rate increases considerably as the input size grows. Nevertheless, the solution is not optimal (8 is the smallest). However, we are confident that these heuristics work efficiently in many cases. □

No.	*Merge*	*Heuristics Used*
1	$\{u, y\} \mapsto \gamma^0$ $(= 0d1dd)$	H5
2	$\{w, \gamma^0\} \mapsto \gamma^1$ $(= 0dd1d)$	H3, H4
3	$\{z, \gamma^1\} \mapsto \gamma^2$ $(= ddd10)$	H3, H4
4	$\{v, \gamma^2\} \mapsto \gamma^3$ $(= dddd0)$	H2
5	$\{s, \gamma^3\} \mapsto \gamma^4$ $(= d11dd)$	H1
6	$\{x, \gamma^3\} \mapsto \gamma^5$ $(= dd0dd)$	H1
7	$\{s, \gamma^5\} \mapsto \gamma^6$ $(= d1dd1)$	H1
8	$\{x, \gamma^5\} \mapsto \gamma^7$ $(= d0ddd)$	H1
9	$\{\gamma^0, \gamma^5\} \mapsto \gamma^8$ $(= 0dddd)$	H1
10	$\{\gamma^4, \gamma^5\} \mapsto \gamma^9$ $(= d1ddd)$	H1
11	$\{\gamma^7, \gamma^9\} \mapsto \gamma^{10}$ $(= ddddd)$	H1, H2

Table 8: Derivation Sequence using Heuristics for Example 4.4

4.3 Falsification

Since the problem of disproving a tautology is as hard as the problem of proving it, we need to devise *fast* heuristics which can be used repeatedly at each step of the Merge-Loop, although they may not be complete (but sound, nevertheless). Two separate heuristics based on greedy methods are designed for the purpose: Falsify_1 and Falsify_2. Falsify_1 uses a greedy method which tries to find an assignment as a counterexample, seeking to maximize the number of terms falsified by the partial assignment made at each step of choosing a variable and its truth value. This can be done by choosing a variable with the most occurrences and assigning the negation of the truth value which takes place most frequently for the variable.

Procedure Falsify_1;

Input: a set of terms, Γ^k from which only backward merges can be derived.

Output: (1) returns a falsifying minterm $\hat{t}$ if it is found.

(2) returns *'Fail'*, otherwise.

Procedure:

```
P1.  Let $x_1, \cdots, x_n$ be the variables.
     $\hat{t}[1..n] \leftarrow 0$;
     Let $N_i(w)$ be the number of occurrences in $\Gamma^k$ of $x_i$ $(1 \leq i \leq n)$
         having the value $w$ (0 or 1).
     **repeat** *P2, P3, and P4* **until** no more reductions are possible.
P2.  **for** each unit term $\alpha \in \Gamma^k$,
         $\hat{t}[j] \leftarrow 1 - \alpha[j]$, where $\alpha[j] \neq d$;
         Perform reductions on $\Gamma^k$ using $\hat{t}[j]$.
     **end.for**;
     **if** $\Gamma^k$ is empty, **then return** $\hat{t}$;
P3.  Compute $N_1(w), \cdots, N_n(w)$ for $\Gamma^k$.
     Compute the index $j$ and the value $w$ such that
         $N_j(w)$ is the largest among $N_1(w), \cdots, N_n(w)$.
     $\hat{t}[j] \leftarrow 1 - w$;
P4.  Perform reductions on $\Gamma^k$ using $\hat{t}[j]$ as follows:
         **for** each term $\alpha \in \Gamma^k$,
            **if** $\alpha[j] = \hat{t}[j]$, **then** $\alpha[j] \leftarrow d$;
            **else if** $\alpha[j] = 1 - \hat{t}[j]$, **then** $\Gamma^k \leftarrow \Gamma^k - \{\alpha\}$;
            **if** $Lit(\alpha) = 0$, **then return** *'Fail'*;
            **if** $\Gamma^k$ is empty, **then return** $\hat{t}$;
        **end.for**;
**end.procedure**;
```

If Γ is not a tautology, then $\Gamma^k (k \geq 0)$ at kth merge step would most likely show the characteristic of H7. The falsifying heuristic Falsify_1 is called from the Merge-Loop every time Γ^k at kth step shows the characteristic of H7. The procedure Falsify_1 works efficiently for many cases, even though it sometimes fails to generate a falsifying assignment for nontautological expressions. Whenever the first procedure (Falsify_1) fails, we try the second procedure (Falsify_2) again on the same input to construct a falsifying minterm with different and better approaches, but more complicated.

In this manner we may have more chances of getting a falsification. At worst case, the falsification process could merely add computational burdens to the Merge-Loop, failing at each invocation. Fortunately, this rarely happens in practice.

Example 4.5 Let $\Gamma^k = \{u, v, w, x, y, z\}$, where u (v, w, x, y, z) is $d1dd1$ ($01ddd$, $ddd00$, $dd0d1$, $d01dd$, $1ddd0$, resp.). Table 9 shows the trace of Falsify_1 for the construction of $\hat{t}$. Falsify_1 needs to have all five iterations for five variables to obtain $\hat{t} = 00010$ for this example. However, for many cases the procedure only tests some portion of the variables to complete the falsification. We show that for the same example Falsify_2 needs three assignments to falsify the set in Example 4.7. □

Example 4.6 Let $\Gamma^k = \{s, u, v, w, x, y, z\}$, where s (u, v, w, x, y, z) is $1d0d1$ ($01d11$, $d011d$, $d100d$, $0d1dd$, $11d01$, $dd1d0$, resp.). Then, Γ^k is reduced to $\emptyset$ because

Iter.	Γ^k	$\hat{t}$	*Reductions*
1	$\{u, v, w, x, y, z\}$	$\hat{t}[2] = 0$	remove u, v from Γ^k & $y[2] = d$.
2	$\{w, x, y, z\}$	$\hat{t}[3] = 0$	remove y from Γ^k & $x[3] = d$.
3	$\{w, x, z\}$	$\hat{t}[5] = 0$	remove x from Γ^k & $z[5] = d$.
4	$\{w, z\}$	$\hat{t}[1] = 0$	remove z from Γ^k.
5	$\{w\}$	$\hat{t}[4] = 1$	remove w from Γ^k.
6	$\emptyset$		

Table 9: Trace of Falsify_1 for Example 4.5

of the reductions taken place in Falsify_1, using the following sequential assignments:

$\hat{t}[2] = 0$, $\hat{t}[3] = 0$ and $\hat{t}[1] = 0$.

This example shows that three out of five variables are required to be assigned by zeroes. But, Falsify_2 for the same input can disprove the set by having only two assignments in Example 4.8. □

The second heuristic has more computing power than the first one in the sense that it can falsify more nontautological expressions than Falsify_1 does. Falsify_2 utilizes information on the number of literals as in Falsify_1 as well as information on the merges occurred at each merge step. Experimental results show that both procedures work efficiently for most cases, consuming a reasonable amount of computing resources.

Procedure Falsify_2;

Input, Output: same as in Falsify_1.

Procedure:

```
P1.   Let $S = \{(\alpha, \beta) | \alpha, \beta \in \Gamma^k$, $\alpha$ and $\beta$ merge backwardly $\}$
      $R \leftarrow \emptyset$;
```

```
        $\hat{t}[1..n] = d$;
P2.     for each unit term $\alpha$ in a pair of $S$,
            $\hat{t}[j] \leftarrow 1 - \alpha[j]$, where $\alpha[j] \neq d$;
            Perform reductions on $S$ using $\hat{t}[j]$.
        end.for;
        if $\Gamma^k$ is empty, then return $\hat{t}$;
P3.     Let $j$ be the position at which merge positions take place most frequently
            for all pairs in $S$.
        Compute $N_1(w), \cdots, N_n(w)$ as in Falsify_1.
        if $N_j(0) > N_j(1)$, then $\hat{t}[j] \leftarrow 1$;
        else $\hat{t}[j] \leftarrow 0$;
P4.     Perform reductions on the set of pairs $S$ using $\hat{t}[j]$ as follows:
            for each term $\alpha$ of each pair in $S$,
                if $\alpha[j] = \hat{t}[j]$, then $\alpha[j] = d$;
                if $\alpha[j] = 1 - \hat{t}[j]$, then remove $\alpha$ from its pair;
            end.for;
            for each member $s$ of $S$,
                if $s$ contains a single term $\eta$,
                then $S \leftarrow S - \{s\}$;    $R \leftarrow R \cup \{\alpha\}$;
            end.for;
        Remove each member of $R$ from $R$ if it appears in a pair of $S$.
        Perform reductions on the set $R$ using $\hat{t}[j]$ as follows:
            for each term $\alpha \in R$,
                if $\alpha[j] = \hat{t}[j]$, then $\alpha[j] = d$;
                if $\alpha[j] = 1 - \hat{t}[j]$, then remove $\alpha$ from $R$;
        end.for;
    if $S$ is empty, go to $P5$;
    else if $S$ contains a contradictory pair, return 'Fail';
    else go to $P2$.
P5. { Call Algorithm-F of Lemma 3.2 }
    Let $R^{\Pi}|_{\hat{t}} = \{\alpha^{\Pi}|_{\hat{t}} \mid \alpha \in R\}$.
    Call Algorithm-F for constructing a minterm $t^*$ which falsifies $R^{\Pi}$.
    for each position $j$ of $t^*$,
        $\hat{t}[l] \leftarrow t^*[j]$, where $l$ is the $j$th position of $\hat{t}$ whose value is $d$.
    return $\hat{t}$;
end.procedure;
```

The goal of a falsifying procedure is to find a minterm $\hat{t}$ which falsifies Γ^k. A falsifying procedure is said to be *consistent* if the decision of $\hat{t}[j]$ for a position j will remain unchanged throughout the falsifying process. It is obvious that both heuristics are consistent and sound. A pair of terms in S of Falsify_2 is said to be *contradictory* if it contains two complementary unit terms. We show that the existence of a contradictory pair is the necessary and sufficient condition for the failure of constructing the minterm $\hat{t}$.

Observation 4.2 If two complementary unit terms are generated by the procedure Falsify_1, then the two terms must be paired in S.

Proof: If they are not paired, then they are not mergeable. Then, there must be another disjoint position j, where both literals on j must have been reduced to d's by the procedure, which is impossible. □

Upon completion of *P4*, if R is empty, then we conclude that $\hat{t}$ falsifies S. Otherwise, R contains those terms that have not yet been falsified by the procedure. We then perform the next part of falsification.

Right before *P5* is to be executed, the following facts are understood:

Observation 4.3 For each term $\alpha \in R$,

(1) $\alpha \neq \sigma_n$,

(2) $\forall p[\hat{t}[p] \neq d \Rightarrow \alpha[p] = d]$, and

(3) there exists a position p such that $\hat{t}[p] = d$ and $\alpha[p] \neq d$. □

Observation 4.4 The remaining terms of the set R are not mergeable with each other.

Proof: Suppose there are two remaining terms α', β' and they are mergeable. Let α (β) be the original term for α' (β').

Case 1. α and β are mergeable.

Then, the pair (α, β) would be removed from S by a reduction at *P4*. So, at least one of α', β' would be removed from R.

Case 2. α and β are not mergeable.

If α and β have no complementary position, then α' and β' are not mergeable. If α and β have two or more complementary positions, there are two cases to be considered:

(i) If there is a complementary position p such that $\hat{t}[p] \neq d$, then one of α' and β' would be removed from R.

(ii) If, for each complementary position p, $\hat{t}[p] = d$, then α' and β' are not mergeable. □

Then, at *P5*, We observe:

1. For each $\alpha^{\Pi}|_{\hat{t}} \in R^{\Pi}|_{\hat{t}}$, $\alpha^{\Pi}|_{\hat{t}}$ is not a *complete minterm.*

2. Each pair of terms of $R^{\Pi}|_{\hat{t}}$ is not mergeable.

Then, by Algorithm-F of Lemma 3.2, there must be a minterm t^* which falsifies

$R^{11}|_{\hat{t}}$. Now, by assigning new values of t^* to the corresponding positions of $\hat{t}$, $\hat{t}$ falsifies the original set S.

To demonstrate the true capacity of Falsify_2, we need to show Γ^k which is not falsified by Falsify_1 but falsified by Falsify_2. We have many such cases in experimental results for big PLAs, but it is quite hard to get a small example showing the quality because almost all small PLAs which are nontautological can be falsified by Falsify_1 alone. Because of that, same examples shown in Examples 4.5 and 4.6 are used as examples for Falsify_2.

Example 4.7 We use the same input Γ^k of Example 4.5. Using information on merges in addition to the information used in Falsify_1, Falsify_2 can reduce the number of iterations for the construction of $\hat{t}$. Table 10 shows the trace of Falsify_2 on Γ^k. The procedure falsifies the set by making three assignments of truth values to $\hat{t}$, while Falsify_1 needs to have five in Example 4.5. □

Iter.	S	R	$\hat{t}$	*Reductions*
1	$\{(u,w),(u,y),(u,z),(v,y),$ $(v,z),(w,x),(x,y),(x,z)\}$	$\emptyset$	$\hat{t}[5]=0$	remove $\{(u,w),(u,y),(u,z),$ $(w,x),(x,y),(x,z)\}$ from S $R=\{w\}$
2	$\{(v,y),(v,z)\}$	$\{w\}$	$\hat{t}[1]=0$	remove $\{(v,z)\}$ from S
3	$\{(v,y)\}$	$\{w\}$	$\hat{t}[2]=0$	remove $\{(v,y)\}$ from S
4	$\emptyset$	$\{w\}$		

Table 10: Trace of Falsify_2 for Example 4.7

Example 4.8 On the same input Γ^k of Example 4.6, Falsify_2 can falsify the set by having only two assignments: $\hat{t}[3]=0$ and $\hat{t}[5]=0$. One more assignment is necessary to falsify the set $R=\{w\}$ in order to return $\hat{t}$. This can be done by running the Algorithm-F, which completes the falsification. □

A falsifying procedure (or algorithm) is said to be *complete* if a set of terms, Γ, does not cover σ_n, then the procedure guarantees to construct a falsifying minterm $\hat{t}$. We believe that both heuristics are not complete because the problem of designing a *complete* falsification algorithm is as hard as the original problem of checking the tautologyhood, while the time complexity of Falsify_2 is only $O(nm^2)$, where n is the number of variables and m is the number of terms of Γ. So, 'Fail' returned from the procedures does not necessarily imply $\sigma_n \in^c \Gamma^k$. However, both heuristics are *sound* because if such a minterm $\hat{t}$ can be constructed, then obviously Γ does not cover the minterm.

4.4 Simplifications during the Merge-Loop

Without simplification, the Merge-Loop would soon be overwhelmed by enormous search space for even a small size input. Therefore, we need to incorporate various types of simplification techniques into the Merge-Loop so that

- it would not produce useless merges, and
- it would produce good merges which cover other terms, and
- it would shorten the length of the verification by choosing most favorable merge for next merge step.

Simplifications can be done within the Merge-Loop in different ways as follows:

(1) Covered Term reduction:

Simplifications during the Merge-Loop are mostly done by removing covered terms due to forward merges. Unit-merge reduction and simple-merge reduction tend to reduce greatly and cause the process to quickly approach a goal. As the derivation approaches near a goal term, the average cardinality of merges is likely to become small, which means more simplifications can be expected.

(2) Nonessential Term Reduction:

Another possibility is that some terms may never be participated as essential elements in a derivation of a goal term. Removing them will reduce the number of merges at current merge step as well as subsequent steps. The reduction is justified by the following theorem:

Theorem 4.2 (Nonessential Term Reduction) Let Δ be a derivation of σ_n from Γ. If there is a position j and a term α such that $\alpha[j] \neq d$ and $\alpha[j] \neq 1 - \beta[j]$ for each term $\beta(\neq \alpha) \in \Gamma$, then α is *nonessential* to Δ.

Proof: Let γ be a merge of α and η. Since $\eta[j] \neq 1 - \alpha[j]$, the position j cannot be the merge position. So, $\gamma[j] = \alpha[j] \neq d$. And any merge using γ cannot produce d at position j for the same reason. The theorem follows directly from a simple induction on $|\Delta|$. $\square$

The reduction can take place in the entire range of the Merge-Loop. Although the reduction rate is not as big as the first one, it certainly contributes to the simplification of the merge process to some extent.

(3) Nonmerge Reduction:

By Theorem 3.6, nonmerge terms are removed from the input. The situation can be found mostly in a sparse circuit where many redundancies are involved.

Nonessential term reduction and nonmerge reduction are found to be effective in many experimental cases. We want to compare the two reduction schemes in detail. First of all, a nonessential term may or may not be mergeable with other terms, while a nonmerge term cannot be merged with any term at all as the name suggests. Second, a positional testing is enough for detecting nonessential terms, but a test for each pair of terms is required for recognizing nonmerge terms. Clearly, we can observe that testing nonmerge reduction is more costly than the other case. Therefore, it is natural to put the nonessential term test before the nonmerge test.

4.5 The Merge_Loop_2

We now present the final version of the Merge-Loop, the Merge_Loop_2 as an improvement over the Merge_Loop_1, which is an implementation of heuristics, falsification procedures and simplification rules proposed in previous sections.

Algorithm Merge_Loop_2;

Input, Output: same as in the Merge_Loop_1

Procedure:

```
P1.   Let $\Gamma^0 = \Gamma$.
      $k \leftarrow 0$;
P2.   while $\tilde{\eta} \notin \Gamma^k$ ($\tilde{\eta}$ is a generalization of $\eta$) do
          Choose the first heuristic from H1 to H6.
          Perform a merge process, using the heuristic chosen.
          Perform reductions according to Section 4.4.
          if only backward merges are possible in $\Gamma^k$ for generating a new merge,
          then Perform Falsify_1 on $\Gamma^k$.
               if Falsify_1 returns a falsifying minterm $\hat{t}$, then return $\hat{t}$;
               else Perform Falsify_2 on $\Gamma^k$;
                    if Falsify_2 returns a falsifying minterm $\hat{t}$, then return $\hat{t}$;
          Let $\gamma^k$ be a nondegenerating merge of two terms from $\Gamma^k$.
          if $\gamma^k$ is not found, return False.
          Let $\Gamma^{k+1} = (\Gamma^k - \{\text{term(s) covered by } \gamma^k\}) \bigcup \{\gamma^k\}$.
          $k \leftarrow k + 1$;
```

```
    end.while;
    return True;
end.procedure;
```

We show that the above procedure is valid.

Theorem 4.3 (Merge_Loop_2) The Merge_Loop_2 returns True iff $\eta \in^c \Gamma$.

Proof: Immediately from the following:

(1) The procedure terminates because (i) only nondegenerating merges can be added to the system as the result of using heuristics, and (ii) the Merge_Loop_1 terminates.

(2) The Merge_Loop_1 is sound and complete (Theorem 4.1).

(3) Reduction rules employed in the Merge_Loop_2 preserves the tautologyhood.

□

Example 4.9 shows when and how Falsify_1 is performed and Example 4.10 demonstrates how heuristics for accelerating the Merge-Loop are selected and how various reductions can be made in order to quickly approach the goal.

Example 4.9 Let $\Gamma^0 = \{q, r, s, t, u, v, w, x, y, z\}$, where q (r, s, t, u, v, w, x, y, z) is $01001d0$ ($ddddd11$, $001d1dd$, $1dd0d00$, $d111001$, $10ddd10$, $d0d101d$, $11dddd0$, $dd1dd0d$, $0001dd1$, resp.). The sequence of merges derived from Γ^0 is $\gamma^0 \cdots \gamma^7$, which represents $1dddd10$, $1dddd1d$, $1d1dddd$, $d01d1dd$, $1dd0dd0$, $d1001d0$, $dd1ddd1$, $00d1dd1$, respectively in the same order. Table 11 shows the trace of Merge_Loop_2. When $k = 8$, no nonbackward merge is attainable for generating a new merge. The procedure now calls Falsify_1 to find a counterexample for the set Γ^8. Falsify_1 returns a minterm $\hat{t}$ after making five assignments: $\hat{t}[2] = 1$, $\hat{t}[6] = 1$, $\hat{t}[7] = 0$, $\hat{t}[1] = 0$, $\hat{t}[3] = 1$. □

Example 4.10 Let $\Gamma^0 = \{o, p, q, r, s, t, u, v, w, x, y, z\}$, where o (p, q, r, s,

k	Γ^k	γ^k	*Heuristics Used*	*Reductions*
0	$\{q, r, s, t, u, v, w, x, y, z\}$	$\{v, x\} \mapsto \gamma^0$	H4	v:CV
1	$\{q, r, s, t, u, w, x, y, z, \gamma^0\}$	$\{\gamma^0, r\} \mapsto \gamma^1$	H3, H4, H5	γ^0:CV
2	$\{q, r, s, t, u, w, x, y, z, \gamma^1\}$	$\{\gamma^1, y\} \mapsto \gamma^2$	H3, H5, H6	$\emptyset$
3	$\{q, r, s, t, u, w, x, y, z, \gamma^1, \gamma^2\}$	$\{\gamma^2, s\} \mapsto \gamma^3$	H4	s:CV
4	$\{q, r, t, u, w, x, y, z, \gamma^1, \gamma^2, \gamma^3\}$	$\{\gamma^1, t\} \mapsto \gamma^4$	H4	t:CV, γ^2:NM
5	$\{q, r, u, w, x, y, z, \gamma^1, \gamma^3, \gamma^4\}$	$\{\gamma^4, q\} \mapsto \gamma^5$	H3, H4	q:CV
6	$\{r, u, w, x, y, z, \gamma^1, \gamma^3, \gamma^4, \gamma^5\}$	$\{r, y\} \mapsto \gamma^6$	H5	$\emptyset$
7	$\{r, u, w, x, y, z, \gamma^1, \gamma^3, \gamma^4, \gamma^5, \gamma^6\}$	$\{\gamma^6, z\} \mapsto \gamma^7$	H4	z:CV
8	$\{r, u, w, x, y, z, \gamma^1, \gamma^3, \gamma^4, \gamma^5, \gamma^6, \gamma^7\}$			

CV: covered term reduction, NM: nonmerge reduction

Table 11: Trace of the Merge_Loop_2 for Example 4.9

t, u, v, w, x, y, z) is $d1d10$ ($010d1$, $0dd1d$, $10dd1$, $d010d$, $111d0$, $dd01d$, $d0dd0$, $0dd01$, $d100d$, $d11d1$, $0d100$, resp.). The sequence of merges derived from Γ^0 is $\gamma^0 \cdots \gamma^{14}$, which represents $ddd10$, $0d1d0$, $d11d0$, $d11dd$, $dd10d$, $d1d0d$, $10ddd$, $0ddd1$, $0d1dd$, $d1d1d$, $d1ddd$, $dddd0$, $0dddd$, $d0ddd$, $ddddd$, respectively in the same order. Table 12 shows the trace of Merge_Loop_2 on Γ^0, illustrating how σ_5 can be reached. □

In summary, the major improvements made in the Merge_Loop_2 are

- The addition of reduction rules in order to reduce the search space.
- The implementation of heuristics for speed-up of the merge process.
- The implementation of the falsification procedures for seemingly nontautolog-

ical expressions.

4.6 Algorithm Cover-Merge

In this section, we present the top level algorithm Cover-Merge, which calls the Algorithm Cover to test whether or not a PLA ϕ covers a term β of the other PLA. We first show how the Algorithm Cover works and then provide a proof that establishes the relationship between cover testing and tautology testing. Next, we describe the Algorithm Cover-Merge which basically implements the divide-and-conquer method, described in Theorem 3.1.

```
Algorithm Cover;
Input: a PLA $\phi$ and a term $\beta$.
Output: returns 'Yes' if $\phi$ covers $\beta$. 'No', otherwise.
Procedure:
P1.     Let $\Gamma_0$ be the set of all terms of $\phi$.
        if any term of $\Gamma_0$ covers $\beta$, then return 'Yes';
        Let $\Gamma_1$ be $\{\alpha \,|\, \alpha \in \Gamma_0,\ \alpha$ is not disjoint with $\beta\}$.
        Let $\Gamma = \Gamma_1^{\Pi}|_{\beta}$.
P2.     Call the Merge-Loop to test the tautologyhood of $\Gamma$.
        if $\Gamma$ is found to be a tautology,
        then return 'Yes';
        else return 'No';
end.procedure;
```

Theorem 4.4 (The Algorithm Cover) The Algorithm Cover returns *'Yes'* iff $\beta \in^{c} \phi$.

Proof: The proof of *P2* is shown in Theorem 4.3. It is sufficient to prove that

$\beta \in^c \Gamma_0$ iff $\sigma_b \in^c \Gamma$, where b is the number of d's in β.

Part A: Suppose $\beta \in^c \Gamma_0$. Then, $\beta \in^c \Gamma_1$, directly from Theorem 3.3 and Observation 3.3. Then, $\sigma_b \in^c \Gamma$, justified by the projection reduction (Theorem 3.4).

Part B: Conversely, suppose $\sigma_b \in^c \Gamma$. Let $\hat{\beta} = \beta^{-\Pi}\|\beta^{\Pi}$ and $\hat{\Gamma_1} = \{\alpha_1^{-\Pi}\|\alpha_1^{\Pi}, \cdots, \alpha_m^{-\Pi}\|\alpha_m^{\Pi}\}$ according to the definition of $\beta^{-\Pi}, \beta^{\Pi}, \alpha^{-\Pi}$ and α^{Π} from projection operation (Definition 3.4), where $\Gamma_1 = \{\alpha_1, \cdots, \alpha_m\}$ and $\beta^{\Pi} = \sigma_b$ ($x\|y$ denotes the concatenation of two strings x and y). Recall that each term $\alpha \in \Gamma_1$ is not disjoint with β. Then,

(1) for each position i, $\alpha[i]$ and $\beta[i]$ are not complementary, and

(2) for each position i, $\beta^{-\Pi}[i] \neq d$, by the definition of $\beta^{-\Pi}$.

From (1) and (2), there are only two cases to be considered:

- $\beta^{-\Pi}[i] = \alpha^{-\Pi}[i] = 0$ or 1.
- $\beta^{-\Pi}[i] = 0$ or 1, and $\alpha^{-\Pi}[i] = d$.

Then, it follows that $\beta^{-\Pi} \in^c \alpha^{-\Pi}$. That is, $\beta^{-\Pi} \in^c \alpha_1^{-\Pi}$, $\cdots$, $\beta^{-\Pi} \in^c \alpha_m^{-\Pi}$. Then, clearly, $\{\beta^{-\Pi}\|\alpha_1^{\Pi}, \cdots, \beta^{-\Pi}\|\alpha_m^{\Pi}\}$ covers $\beta^{-\Pi}\|\beta^{\Pi}$, which implies that,

$\{\alpha_1^{-\Pi}\|\alpha_1^{\Pi}, \cdots, \alpha_m^{-\Pi}\|\alpha_m^{\Pi}\}$ covers $\beta^{-\Pi}\|\beta^{\Pi}$. So, $\hat{\beta} \in^c \hat{\Gamma_1}$. Hence, $\beta \in^c \Gamma_1$. Therefore, $\beta \in^c \Gamma_0$ because Γ_0 is a superset of Γ_1. □

The Algorithm Cover-Merge is the top level equivalence checker realizing the divide-and-conquer strategy of Theorem 3.1.

Algorithm Cover-Merge;

Input: two PLAs ϕ_1 and ϕ_2.

Output: returns *'Yes'* if $\phi \equiv \phi_2$. *'No'*, otherwise.

Procedure:

```
for each $\alpha \in \phi_1$,
    Call Algorithm Cover to test whether $\alpha \in^c \phi_2$.
```

```
        **if** $\alpha \notin^c \phi_2$, **then return** *'No'*;
    **end.for**;
    **for** each $\beta \in \phi_2$,
        Call Algorithm Cover to test whether $\beta \in^c \phi_1$.
        **if** $\beta \notin^c \phi_1$, **then return** *'No'*;
    **end.for**;
    **return** *'Yes'*;
**end.procedure**;
```

Theorem 4.5 (The Cover-Merge Algorithm) The Algorithm Cover-Merge returns *'Yes'* iff $\phi_1 \equiv \phi_2$.

Proof: Directly from Theorems 4.4 and 3.1. □

k	Γ^k	γ^k	*Heuristics*	*Reductions*
0	$\{o, p, q, r, s, t, u, v, w, x, y, z\}$	$\{o, v\} \mapsto \gamma^0$	H4, H5	o:CV
1	$\{p, q, r, s, t, u, v, w, x, y, z, \gamma^0\}$	$\{\gamma^0, z\} \mapsto \gamma^1$	H3	z:CV
2	$\{p, q, r, s, t, u, v, w, x, y, \gamma^0, \gamma^1\}$	$\{\gamma^1, t\} \mapsto \gamma^2$	H3	t:CV
3	$\{p, q, r, s, u, v, w, x, y, \gamma^0, \gamma^1, \gamma^2\}$	$\{\gamma^2, y\} \mapsto \gamma^3$	H3, H2	y, γ^2:CV
4	$\{p, q, r, s, u, v, w, x, \gamma^0, \gamma^1, \gamma^3\}$	$\{\gamma^3, s\} \mapsto \gamma^4$	H3, H5	s:CV
5	$\{p, q, r, u, v, w, x, \gamma^0, \gamma^1, \gamma^3, \gamma^4\}$	$\{\gamma^4, x\} \mapsto \gamma^5$	H3, H4	x:CV
6	$\{p, q, r, u, v, w, \gamma^0, \gamma^1, \gamma^3, \gamma^4, \gamma^5\}$	$\{r, v\} \mapsto \gamma^6$	H4, H5	r:CV
7	$\{p, q, u, v, w, \gamma^0, \gamma^1, \gamma^3, \gamma^4, \gamma^5, \gamma^6\}$	$\{q, w\} \mapsto \gamma^7$	H4, H5	p, w:CV
8	$\{q, u, v, \gamma^0, \gamma^1, \gamma^3, \gamma^4, \gamma^5, \gamma^6, \gamma^7\}$	$\{\gamma^7, \gamma^1\} \mapsto \gamma^8$	H3	γ^1:CV
9	$\{q, u, v, \gamma^0, \gamma^3, \gamma^4, \gamma^5, \gamma^6, \gamma^7, \gamma^8\}$	$\{\gamma^3, u\} \mapsto \gamma^9$	H5	$\emptyset$
10	$\{q, u, v, \gamma^0, \gamma^3, \gamma^4, \gamma^5, \gamma^6, \gamma^7, \gamma^8, \gamma^9\}$	$\{\gamma^9, \gamma^5\} \mapsto \gamma^{10}$	H3, H2	$\gamma^3, \gamma^5, \gamma^9$:CV
11	$\{q, u, v, \gamma^0, \gamma^4, \gamma^6, \gamma^7, \gamma^8, \gamma^{10}\}$	$\{\gamma^{10}, v\} \mapsto \gamma^{11}$	H1	v, γ^0:CV
12	$\{q, u, \gamma^4, \gamma^6, \gamma^7, \gamma^8, \gamma^{10}, \gamma^{11}\}$	$\{\gamma^{11}, \gamma^7\} \mapsto \gamma^{12}$	H1, H3	q, γ^7, γ^8:CV, u:NM, γ^4, γ^{11}:NE
13	$\{\gamma^6, \gamma^{10}, \gamma^{12}\}$	$\{\gamma^{12}, \gamma^6\} \mapsto \gamma^{13}$	H1	γ^6:CV, γ^{12}:NE
14	$\{\gamma^{10}, \gamma^{13}\}$	$\{\gamma^{13}, \gamma^{10}\} \mapsto \gamma^{14}$	H1	$\gamma^{10}, \gamma 13$:CV
15	$\emptyset$			

CV: covered term reduction, NM: nonmerge reduction, NE: nonessential reduction

Table 12: Trace of the Merge_Loop_2 for Example 4.10

CHAPTER 5

RANDOM MODELS

Many approaches to solve satisfiability-related problems have been made to achieve good performances on specific problem domains. Using domain-specific knowledges and built-in information, one can devise an efficient algorithm under particular conditions. (An example for a circuit testing problem: some types of circuits might have the symmetric property which can be found very effective for designing a testing algorithm.) Although algorithms like that could run well on special types of data sets which contain domain-specific knowledge, it may not be the case that the same algorithms could run also well on other types of data sets. So, parameterized random sets of problems have gained much attention from research people on this area in order to give an unbiased way of testing algorithms [24, 25, 50, 109].

Thus, randomized problem instances do not include any built-in knowledge on specific domain of problems. But this does not mean that devising an algorithm which embodies domain-specific knowledge for a particular type of problems is not important. The paradigm of this is that algorithms that do well on random problems would do also well on specific problems too [23].

There are various types of *random satisfiability models* that have been proposed by researchers. Even though there is no standard random model, we do have the popular model which has been used most prevalently by researchers. There is no name for the model, but most people call it *the popular model* [24, 25, 109, 55]. Let us first define the control parameters:

- t : the number of clauses to be generated.

- v : the number of variables.
- p : the probability that each literal appears in a clause.

The popular model assumes that each literal has the same probability p to be selected for a clause and each clause is independently generated. With this scheme, a clause can contain both a variable x and its negation, which becomes a tautological clause. In that case, the truth value of such clause is true regardless of truth values of other variables that are in the clause. Also, it is possible that a clause can contain no literals at all (i.e. a *null clause*).

In the following sections, we first describe how random PLAs can be generated using the popular model. To obtain two equivalent random PLAs, we introduce a set of *transformation rules* which derives a logically equivalent set of terms from a random PLA. Next, a probabilistic analysis for random PLAs is stated and experimental results for various parametric values show the probabilistic properties of random PLAs. We further study the properties of random PLAs consisting only of active terms.

5.1 Constructing Random PLAs

We use the popular model to define our random PLAs so that each term is randomly generated according to the paradigm of the popular model. Thus, a *random PLA* can be generated with parameters:

- v : the number of variables.
- t : the number of terms.
- p : the probability that each literal appears in a term.

A *term* is generated randomly by independently selecting the $2v$ literals with probability p. A *random PLA* is generated randomly by independently forming t terms.

Because of the way we build random PLAs, we can expect the following phenomena:

- Some terms may be empty (*null terms*).
- Some terms may consist of both x_i and $\bar{x}_i$ (*contradictory*, treated as *false*).
- Some terms may be duplicates of others.
- Some variables may not occur at all in a random PLA.

The probability that each of these occurs depends on the values of the parameters, which will be shown in next section. However, we need to remove the first two occurrences from the resulting PLAs because they cannot be considered as meaningful terms for our purpose.

A pair of random PLAs can be generated in two different modes for the equivalence problem:

(1) *Mode 1* : Two random PLAs are generated independently.

(2) *Mode 2* : A PLA, ϕ_1, is generated randomly and then another PLA, ϕ_2, is derived from ϕ_1 using *transformation rules* which preserve the logical equivalence. This method guarantees to produce two equivalent random PLAs.

The first one is straightforward but the second mode depends on the *transformation rules* which need to be explained in detail. The overall objective of the second mode is to obtain a PLA ϕ_2 which is quite different from ϕ_1 in terms of its syntactic structures, but nevertheless preserves the logical equivalence.

First, we propose the *transformation rules* and define their functionalities. Let ϕ_1 be a random PLA. Define the following transformation rules:

1. Combine: The *combine* operation is to replace two complementary terms $\alpha, \beta \in \phi_1$ by γ, where γ is the merge of α and β.

2. Divide: The *divide* operation is the reverse of *combine*. If a variable x does not occur in a term $\alpha \in \phi_1$, then replace α by $\alpha \cdot x + \alpha \cdot \bar{x}$.

3. Remove-Intersection: This function removes the intersection of two terms $\alpha, \beta \in \phi_1$. A subexpression $\alpha+\beta$ of ϕ_1 is replaced by $\alpha+(\beta\ominus(\alpha\cap\beta))$ or $\alpha\ominus(\alpha\cap\beta)+\beta$. For example, $\bar{x_1}\bar{x_2}x_5\bar{x_6} + \bar{x_2}\bar{x_4}x_5$ is replaced by $\bar{x_1}\bar{x_2}x_5\bar{x_6} + x_1\bar{x_2}\bar{x_4}x_5 + \bar{x_1}\bar{x_2}\bar{x_4}x_5x_6$ or $\bar{x_1}\bar{x_2}x_4x_5\bar{x_6} + \bar{x_2}\bar{x_4}x_5$.

The *divide* operation is always possible on any term except a minterm and it should take a term of ϕ_1 as its argument. The operation first selects a term randomly and then a variable to be divided randomly. However, this transformation will be used only when other rules are not applicable because otherwise ϕ_2 would show a structural resemblance to ϕ_1 and equivalence testing can be trivially done by performing a sequence of simple merges. To mitigate that problem, we test each of the *divided terms* for further transformations except *dividing.*

The probability of applying *combine* rule is relatively low in comparison with that of *remove-intersection.* Special care must be taken not to combine two terms which have been previously divided and vice versa. To do that, a special data structure is maintained to keep track of the changes made by the *divide* or *combine rule.* The chance of getting *remove-intersection rule* depends on the number of nondisjoint pairs of ϕ_1 which might be dynamically changed by applying other rules.

The *combine* and *remove-intersection* operations ought to take at least a term of ϕ_1 or a divided term as its argument. The transformation process ends after each term of ϕ_1 has been replaced. Example 5.1 shows how the transformation rules are used to derive a new PLA from an existing one.

Example 5.1 Let $\phi_1 = u + v + w + x + y + z$, where u (v, w, x, y, z) is $01dd1$ ($d0d1d$, $dd110$, $d01d1$, $1d1dd$, $1d010$, respectively). Table 13 shows that after 8 transformations we obtain a new set of terms ϕ_2 from ϕ_1:

$\phi_2 = (r_1 + r_2 + r_3 + r_4 + r_6 + r_7 + r_8 + r_{13} + r_{14} + r_{15})$,

where r_1 (r_2, r_3, r_4, r_6, r_7, r_8, r_{13},r_{14},r_{15}) is 1d10d (1d111, 00d1d, 1011d, 001d1, 10111, 0d110, 11d10, 01d01, 01d11). □

The resulting PLA tends to be bigger than the original because of the transformation rules. The actual number of terms of ϕ_2 depends on the types of transformation rules used, the order of the rules applied and their arguments too.

5.2 Probabilistic Properties of Random PLAs

In this section, we study various probabilistic properties of the random model.

Let $x = x_1 \cdots x_i \cdots x_v$ be a term. Let $P(x_i = 1)$ and $P(x_i = 0)$ be the probability that $x_i = 1$ and $x_i = 0$, respectively.

P1. $P(x_i = 1) = P(x_i = 0) = p$.

The probability that both x_i and $\bar{x}_i$ occur is:

P2. $P(x_i = 1, x_i = 0) = p^2$.

The probability that neither x_i nor $\bar{x}_i$ occur is:

P3. $\neg P(x_i = 1, x_i = 0) = 1 - p^2$.

The probability that a term is not contradictory is:

P4. $\neg P(cont) = (1 - p^2)^v$.

The probability that a term is contradictory is:

P5. $P(cont) = 1 - (1 - p^2)^v$.

The probability that x_i does not occur is:

P6. $P(x_i = d) = \neg P(x_i = 1) \& \neg P(x_i = 0) = (1 - p)^2$.

The probability that a term is null is:

P7. $P(null) = (1 - p)^{2v}$.

The probability that a term is useless is:

P8. $P(useless) = P(cont)$ or $P(null) = 1 - (1-p^2)^v + (1-p)^{2v}$.

The probability that a term is active is:

P9. $P(active) = 1 - P(useless) = (1-p^2)^v - (1-p)^{2v}$.

Let L be the number of literals of a term and W be the number of active terms of a random PLA.

The expected number of literals of a term is:

P10. $E(L) = 2pv$.

The expected number of active terms of a random PLA is:

P11. $E(W) = t[(1-p^2)^v - (1-p)^{2v}]$.

Define *density* as $D = \frac{L}{v}$ for a term. The *expected density* of a random PLA is:

P12. $E(D) = \frac{E(L)}{v}$.

Table 14 shows some of the probabilistic results for $v = 10, 30, 50$ and $p = 0.001$ to 0.5. We observe the following results:

(1) when $p \geq 0.4$, it is hardly likely to obtain a term without a contradiction except for small v.

(2) when $p > 0.3$ and v is large, it will be hard to obtain an active term.

(3) when $p \leq 0.001$, almost all terms will be null.

(4) when $p \leq 0.01$, the expected number of literals in a term is too small (less than two).

So, for a practical purpose, a feasible range of p would be $[0.1, 0.2]$. Also, we point out that $E(L)$ gets close to v as p approaches 0.5. This happens because contradictory terms are not excluded from random PLAs. Experimental results (Table 15) show a close resemblance to Table 14.

5.2.1 Probabilistic Analysis of Active Terms

Because of the way we construct random PLAs using the *popular model*, useless terms such as contradictory and null terms may be included. To get a more meaningful random model, it is necessary to remove any nonsensical terms from a random PLA so that only active terms should remain.

Let x and y be two active terms. Then, x_i and y_i can be one of three values — $0, 1$ and d with the following probabilities:

$P(x_i = 0) = P(x_i = 1) = \frac{p}{1+p^2}$.

$P(x_i = d) = \frac{(1-p)^2}{1+p^2}$.

The probability that x_i and y_i are disjoint is:

P13. $P(x_i,y_i\text{:}disjoint) = P(x_i = 0, y_i = 1) + P(x_i = 1, y_i = 0) = 2(\frac{p}{1+p^2})^2$.

The expected number of literals in an active term x is:

P14. $E(L) = v - vP(x_i = d) = v - v\frac{(1-p^2)^2}{1+p^2}$.

The probability that two terms x, y are shared is:

P15. $P(shared) = [1 - P(x_i,y_i\text{:}disjoint)]^v = [1 - 2(\frac{p}{1+p^2})^2]^v$.

The probability that two terms x, y are disjoint is:

P16. $P(disjoint) = 1 - P(shared) = 1 - [1 - 2(\frac{p}{1+p^2})^2]^v$.

The probability that two terms x, y are mergeable is:

P17. $P(merge) = vP(x_i,y_i\text{:}disjoint)\ \neg P(x_i,y_i\text{:}disjoint)^{v-1} = 2v(\frac{p}{1+p^2})^2(1 - 2(\frac{p}{1+p^2})^2)^{v-1}$.

Example 5.2 Let x, y be any two active terms. Then, $P(shared) = 0.267$ (when, $v = 30, p = 0.15$) and 0.108 (when $v = 50, p = 0.15$). □

Table 16 shows the probabilistic results for W(number of active terms), L(number of literals in a term), $P(disjoint)$ and $P(merge)$. Table 17 shows the results from random PLAs. We observe that the probabilistic analysis and the actual results exhibit a close resemblance.

From the the statistics gathered, we observe the following facts:

1. Let $\delta_p = (1-p^2)^v - (1-p)^{2v}$ (*P9*). Then, obviously, $\delta_p > 0$ for $0 < p < 1$. The popular model always produce active terms for $0 < p < 1$ and $v > 0$. However, It is possible that δ_p can be so small that t should be very big to obtain an active term, which is considered to be impractical in most cases.

2. At $p = 0.1$, W is maximum for $v \leq 50$ and decreases sharply as p deviates from 0.1. For $50 < v < 100$, W is maximized at $p = 0.01$ because $\delta_{0.01} > \delta_{0.1}$.

3. For the same value of p (≥ 0.1), W decreases as v increases. For $p \leq 0.01$ and $v < 100$, W increases as v increases because $L < 2$.

4. As p approaches 0.5, W quickly diminishes to zero for $v > 10$.

5. The larger the value of v is, the faster W approaches zero. For example, if $v > 50$, p is expected to be no greater than 0.2 to obtain any active terms.

6. For small v (for instance, $v \leq 10$), it is always possible for $p \leq 0.5$ that active terms can be obtained.

7. The value of L increases as the probability p grows for $p < 0.5$ until $W = 0$. Table 18 and 19 shows the behavior of W and L, respectively, for a wider range of v ($10 \leq v \leq 800$), based on the empirical results collected from the random construction of PLAs.

8. Maximum density for different values of v and p decreases as v grows. For example, $D = 0.68$ $(0.58, 0.34)$ when $v = 10$ $(20, 30$, respectively). In other words, the random PLAs tend to be sparser as v gets bigger, which is shown in Table 20 (Density).

9. The range of p for $W > 0$ shrinks quickly for $v < 50$ and then gets narrower slowly as v increases, which is shown in Table 20 (Upper Bound).

10. $P(disjoint)$ increases as p increases until $W = 0$. However, $P(merge)$ decreases sharply as p increases after p shows peak values. This is because the number of complementary positions of two active terms would increase as p increases.

11. If active terms exist ($W > 2$), almost always $P(merge) > 0$.

12. Maximum value of $P(merge)$ decreases as v increases.

We have seen that random PLAs using the popular model have intrinsic properties determined by the setting of the control parameters. Even though random models may not exhibit any particular structural characteristics such as symmetry and hierarchy, it certainly convey some probabilistic properties acquired from control parameters for constructing random PLAs. Given the values of the parameters, one could expect the type of random PLAs to be created. (For example, the expected number of active terms, the density, etc.)

Therefore, we should not restrict the parameters to a certain range of values because, if we do, random PLAs created under a restricted range might represent a particular type of PLAs having similar probabilistic properties. By appropriately setting the control parameters, we might be able to see different types of random PLAs.

Iter.	ϕ_2	*Transformations*	*Rules Used*
1	$u+v+w+x+y+z$	replace $w+y$ by $w+r_1+r_2$ $r_1 = 1d10d, \quad r_2 = 1d111$	R
2	$u+v+w+x+z+r_1+r_2$	replace $v+z$ by $z+r_3+r_4+r_5$ $r_3 = 00d1d, \quad r_4 = 1011d,$ $r_5 = 100d1$	R
3	$u+w+x+z+r_1+r_2+r_3$ $+r_4+r_5$	replace $x+r_1$ by $r_1+r_6+r_7$ $r_6 = 001d1, \quad r_7 = 10111$	R
4	$u+w+z+r_1+r_2+r_3+r_4$ $+r_5+r_6+r_7$	replace $w+r_4$ by $r_4+r_8+r_9$ $r_8 = 0d110, \quad r_9 = 11d10$	R
5	$u+z+r_1+r_2+r_3+r_4+r_5$ $+r_6+r_7+r_8+r_9$	replace z by $r_{10}+r_{11}$ $r_{10} = 10010, \quad r_{11} = 11010$	D
6	$u+r_1+r_2+r_3+r_4+r_5+r_6$ $+r_7+r_8+r_9+r_{10}+r_{11}$	replace r_5+r_{10} by r_{12} $r_{12} = 1001d$	C
7	$u+r_1+r_2+r_3+r_4+r_6+r_7$ $+r_8+r_9+r_{11}+r_{12}$	replace r_9+r_{11} by r_{13} $r_{13} = 11d10$	C
8	$u+r_1+r_2+r_3+r_4+r_6+r_7$ $+r_8+r_{12}+r_{13}$	replace u by $r_{14}+r_{15}$ $r_{14} = 01d01, \quad r_{15} = 01d11$	D
9	$r_1+r_2+r_3+r_4+r_6+r_7+r_8$ $+r_{12}+r_{13}+r_{14}+r_{15}$		

R: remove-intersection, C: combine, D: divide

Table 13: Generating an Equivalent PLA for Example 5.1

Probabilistic Results for $v = 10, 30, 50$ and $t = 1000$												
p	*P(cont) (%)*			*P(null) (%)*			*P(act) (%)*			*E(L)*		
	10	30	50	10	30	50	10	30	50	10	30	50
0.001	0.0	0.0	0.0	98.0	94.1	90.4	1.9	5.8	9.5	0.0	0.0	0.1
0.01	0.1	0.3	0.4	81.7	54.7	36.6	18.1	44.9	62.8	0.2	0.6	1.0
0.1	9.5	26.0	39.4	12.1	0.1	0.0	78.2	73.7	60.4	2.0	6.0	10.0
0.15	20.3	49.4	67.9	3.8	0.0	0.0	75.7	50.5	32.0	3.0	9.0	15.0
0.2	33.5	70.6	87.0	1.1	0.0	0.0	65.3	29.3	12.9	4.0	12.0	20.0
0.3	61.0	94.0	99.1	0.0	0.0	0.0	38.8	5.9	0.8	6.0	18.0	30.0
0.4	82.5	99.4	99.9	0.0	0.0	0.0	17.4	0.5	0.0	8.0	24.0	40.0
0.5	94.3	99.9	100.0	0.0	0.0	0.0	5.6	0.0	0.0	10.0	30.0	50.0

P(cont) : probability that a term is contradictory.

P(null) : probability that a term is null.

P(act) : probability that a term is active.

E(L) : expected number of literals in a term.

Table 14: Probabilistic Analysis of Random PLAs

Empirical Results for $v = 10, 30, 50$ and $t = 1000$												
p	*Contradictory (%)*			*Null (%)*			*Active (%)*			L		
	10	30	50	10	30	50	10	30	50	10	30	50
0.001	0.0	0.0	0.0	97.8	94.8	92.0	2.2	5.2	8.0	1.0	1.0	1.1
0.01	0.0	0.1	0.5	84.0	60.2	39.7	16.0	39.7	60.2	1.1	1.2	1.5
0.1	9.1	27.3	38.9	13.2	0.3	0.0	79.2	72.4	61.1	2.2	5.3	9.1
0.15	19.4	48.8	67.4	6.2	0.1	0.0	76.0	51.1	32.6	2.8	7.9	13.2
0.2	32.2	69.6	86.8	1.9	0.0	0.0	66.8	30.1	13.2	3.4	10.2	16.6
0.3	59.9	94.1	99.1	0.7	0.0	0.0	40.1	5.9	0.9	4.7	13.5	24.2
0.4	81.0	99.8	99.8	0.1	0.0	0.0	19.0	0.2	0.2	5.9	18.0	31.0
0.5	95.4	99.9	100.0	0.0	0.0	0.0	4.6	0.1	0.0	6.5	25.0	42.7

Contradictory (%) : percentage of contradictory terms.

Null (%) : percentage of null terms.

Active (%) : percentage of active terms.

L : number of literals in a term.

Table 15: Experimental Results for Random PLAs

Probabilistic Results for $v = 10, 30, 50$ and $t = 1000$												
p	$E(W)$			$E(L)$			$P(disjoint)$ (%)			$P(merge)$		
	10	30	50	10	30	50	10	30	50	10	30	50
0.001	20	58	95	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0
0.01	181	450	629	0.2	0.6	1.0	0.2	0.6	1.0	0.2	0.6	1.0
0.1	783	738	605	2.0	5.9	9.9	18.0	44.8	62.8	16.4	33.1	37.2
0.15	758	505	321	2.9	8.8	14.7	35.6	73.3	88.9	29.0	36.1	24.9
0.2	653	294	130	3.8	11.5	19.2	53.6	90.0	97.9	37.0	23.9	8.6
0.3	389	59	9	5.5	16.5	27.5	80.7	99.3	100.0	34.5	3.9	0.2
0.4	175	5	5	6.9	20.7	20.7	93.4	100.0	100.0	20.6	0.3	0.3
0.5	56	0	0	8.0	0.0	0.0	97.9	0.0	0.0	9.9	0.0	0.0

$E(W)$: expected number of active terms.

$E(L)$: expected number of literals in a term.

$P(disjoint)$: probability that a pair of active terms is disjoint.

$P(merge)$: probability that two active terms are mergeable.

Table 16: Probabilistic Analysis of Active Terms

Empirical Results for $v = 10, 30, 50$ and $t = 1000$												
p	W			L			$P(disjoint)$ (%)			$P(merge)$		
	10	30	50	10	30	50	10	30	50	10	30	50
0.001	15	47	92	1.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
0.01	181	406	582	1.0	1.3	1.6	5.3	2.6	2.4	5.3	2.6	2.4
0.1	795	735	588	2.1	5.5	8.9	21.0	39.9	55.1	19.5	31.1	36.2
0.15	758	497	297	2.6	7.8	13.0	30.3	64.4	82.0	26.1	37.1	31.0
0.2	648	299	120	3.4	9.9	17.1	44.5	81.8	95.3	34.1	32.2	15.6
0.3	396	58	8	4.7	13.7	22.8	68.7	96.2	100.0	38.9	12.4	3.6
0.4	178	8	0	5.8	17.4	0.0	83.5	100.0	0.0	32.4	0.0	0.0
0.5	71	0	0	6.8	0.0	0.0	92.9	0.0	0.0	21.1	0.0	0.0

W : number of active terms.

L : number of literals in a term.

$P(disjoint)$: probability that a pair of active terms is disjoint.

$P(merge)$: probability that two active terms are mergeable.

Table 17: Experimental Results for Active Terms

The values of W for $v = [10, 800]$ and $t = 1000$										
p	W									
	10	50	100	200	300	400	500	600	700	800
0.001	15	92	185	351	466	545	650	709	737	798
0.01	181	582	822	955	977	977	966	952	950	950
0.1	795	588	343	131	45	16	2	1	0	0
0.15	758	297	91	12	3	0	0	0	0	0
0.2	648	120	13	0	0	0	0	0	0	0
0.3	396	8	0	0	0	0	0	0	0	0
0.4	178	0	0	0	0	0	0	0	0	0
0.5	71	0	0	0	0	0	0	0	0	0

W : number of active terms.

Table 18: Empirical Results for W

The values of L for $v = [10, 800]$ and $t = 1000$										
p	L									
	10	50	100	200	300	400	500	600	700	800
0.001	1.0	1.0	1.1	1.2	1.3	1.4	1.6	1.8	1.8	2.0
0.01	1.0	1.6	2.2	3.7	5.4	7.1	8.8	10.7	12.5	14.5
0.1	2.1	8.9	18.0	36.8	54.8	70.6	93.0	0.0	0.0	0.0
0.15	2.6	13.0	25.8	52.8	81.0	0.0	0.0	0.0	0.0	0.0
0.2	3.3	17.1	33.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.3	4.7	23.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.4	5.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.5	6.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

L : average number of literals of a term.

Table 19: Empirical Results for L

Results	v									
	10	50	100	200	300	400	500	600	700	800
(1) *Density (%)*	76.0	47.8	33.5	26.4	27.0	17.6	18.6	1.7	1.7	1.8
(2) *Upper Bound for p*	0.6	0.3	0.2	0.15	0.15	0.1	0.1	0.01	0.01	0.01

Density : largest $\frac{L}{v}$ for different values of p.

Upper Bound for p : Largest possible value of p satisfying $W > 0$

Table 20: Empirical Results for (1) Density and (2) Upper Bound for p

CHAPTER 6

EXPERIMENTAL RESULTS

In this chapter, we discuss the experimental results for the equivalence problem. A program was written in C on a Sequent Symmetry, implementing the algorithms and heuristics presented in the previous chapters. We present the results on the standard benchmark problems as well as on various types of PLAs constructed randomly as described in Chapter 5. Algorithms and heuristics implemented in the program will be discussed and analyzed as to how particular techniques are effectively used in the program. Also, the results are compared with the most widely used Boolean function manipulator called *Binary Decision Diagrams* or BDDs (see Section 2.4).

It is not easy to accurately compare the performance of verification methods. However, CPU time required for verification is generally accepted as a good indicator of performance because it would show the asymptotic behavior of running time as the size of input grows. We measure CPU times for both programs (ours and BDD) in 10 milliseconds on the same Sequent machine. The results show that the Cover-Merge Algorithm combining with heuristics works very efficiently for most cases in comparison with BDD, especially for big PLAs where BDD shows no hope of completing the verification because of memory shortage caused by its memory greedy representational scheme.

We prepare four separate programs (CV-A, CV-B, CV-C and CV-D) each of which implements a different set of techniques such that comparison of their performances can be easily made. All four programs implement the same base algorithms: the Algorithm Cover-Merge, the Algorithm Cover and the Merge-Loop. However, they use different techniques in different places and in different ways. Most heuristics are used for simplification of input terms at each iteration of the Merge-Loop, and some are for cutting down the length of proofs. These heuristics are incorporated in various stages of the programs as follows:

1. Before the Algorithm Cover-Merge.
2. Before the Merge-Loop.
3. Inside the Merge-Loop.

Table 21 shows when and where each of the techniques is used for the programs.

The program CV-A is the simplest version which implements the base algorithms with a few essential simplification rules. The program CV-B is essentially same as CV-A except two more reduction rules are added. The program CV-C uses a few more heuristics in addition to those used in CV-B. Since the heuristics used in CV-C are not prioritized, it is possible that a heuristic can be fired even though more promising heuristics are available. The program CV-D is the final version of the Algorithm Cover-Merge, which implements techniques mentioned in the previous chapters. In addition to a few more heuristics added to CV-C, the program CV-D improves its performance significantly by employing the prioritized heuristics and the falsification heuristics.

Benchmark Results

We ran the program CV-D on the benchmarks in the PLASCO[1] group of IFIP. The results show that the program has successfully verified the benchmarks:

1. The outputs *cst2* and *cas1* have non-equivalent expressions in the werner.be.
2. The outputs h, j, k, m and n are non-equivalent in d3.be.

All the benchmarks have been run in less than a few seconds of CPU of the

[1]PLASCO consists of a group of standard benchmark problems for equivalence testing

same Sequent computer. Table 22 shows the comparison of running time of BDD and CV-D. Both programs demonstrate practical performances. However, the benchmark problems are not big enough to demonstrate the true capacity of algorithms.

Results on Random PLAs

Table 23 and 24 show the comparison of the programs run on different sizes of random PLAs in two different modes stated in Chapter 5.

We observe the following results based on the above experiments:

1. Both CV-C and CV-D outperform BDD.
2. CV-D shows the best performance and it does not indicate any steep rise in execution time in the range $v \leq 50$ and it is fairly stabilized within the range.
3. CV-A and CV-B show almost same performance on the first mode and CV-A outperforms CV-B on the second mode of experiments, which suggests that the two rules added to CV-A only creates computational overheads to CV-B. However, those rules are often found to be effective in simplification for many cases when they work in conjunction with other rules and heuristics.
4. The program CV-C outperforms the previous programs for all cases except the case where $v = 20$ on the second mode (see Table 24). This is because simplification rules and heuristics are extensively used at each step of the Merge-Loop and big simplifications occur because of the techniques used.

CV-D improves its performance considerably by employing two important techniques: the prioritization of heuristics and the falsification procedures. The former repeatedly seeks in depth-first the most favorable merge utilizing previous history so that big simplifications take place as many as possible. Speed-ups of the merge process have been achieved because of that. The latter works efficiently in detecting the nontautologyhood of a PLA and in disproving it in most cases.

Experiments on different values of the control parameters might show different performances. CV-D will eventually show an exponential running time with a wider range of v, t, and different values of p because of the NP-completeness of the equivalence problem.

Techniques Used	*CV-A*	*CV-B*	*CV-C*	*CV-D*
1. Before the Cover-Merge				
covered term reduction	√	√	√	√
simple-merge reduction	√	√	√	√
unit-merge reduction		√	√	√
2. Before the Merge-Loop				
projection reduction	√	√	√	√
3. Inside the Merge-Loop				
covered term reduction	√	√	√	√
simple-merge reduction	√	√		
nonessential term reduction			√	√
nonmerge reduction		√	√	√
prioritized heuristics				√
heuristic H1			√	√
heuristic H2			√	√
heuristic H3			√	√
heuristic H4				√
heuristic H5				√
heuristic H6			√	√
heuristic H7				√
falsification heuristics				√

Heuristics H1 through H7 can be found in Section 4.2.

Table 21: Implementation of Algorithms and Heuristics

Benchmarks	CPU Time	
	BDD	*CV-D*
counter.be	7	7
d3.be	93	31
in1.be	1238	206
hostint1.be	6	4
mul.be	7	5
pitch.be	335	52
table.be	64	23
werner.be	1	1

Measured in 10 milliseconds

Table 22: Equivalence Testing of Benchmark Problems: The outputs *cst2* and *cas1* have non-equivalent expressions in the werner.be. In d3.be, *h, j, k, m* and *n* are non-equivalent.

v	CPU Time Measurements in 10 milliseconds				
	BDD	*CV-A*	*CV-B*	*CV-C*	*CV-D*
10	104	8	65	1	3
15	301	30	80	2	3
20	1982	801	92	2	3
25	36152	∞	∞	2867	2693
30	∞	∞	∞	∞	3592
35	∞	∞	∞	∞	5024
40	∞	∞	∞	∞	4820
45	∞	∞	∞	∞	5839
50	∞	∞	∞	∞	7481

Table 23: Equivalence Testing of Two Random PLAs (Mode 1) : $T = 2000$, $p = 0.15$. (Mode 1: two independently generated random PLAs, (see Section 5.1 for construction), T: total number of input terms, p: the probability that a literal appears in a term) '∞' indicates that jobs are terminated manually after about six hours of execution on the Sequent.

Note: BDD spends 1308.61 seconds for $v = 30$ and $T = 1200$, while CV-D takes 28.31 seconds.

v	CPU Time Measurements in 10 milliseconds				
	BDD	*CV-A*	*CV-B*	*CV-C*	*CV-D*
10	191	143	74	21	7
15	495	34	92	4	2
20	2522	13524	146	1263	396
25	44392	46478	∞	10408	9709
30	∞	∞	∞	15551	10828
35	∞	∞	∞	20241	7658
40	∞	∞	∞	19053	9939
45	∞	∞	∞	19714	8595
50	∞	∞	∞	16554	8973

Table 24: Equivalence Testing of Two Random PLAs (Mode 2) : $T = 2000$, $p = 0.15$. (Mode 2: two equivalent random PLAs (see Section 5.1 for construction), T: total number of input terms, p: the probability that a literal appears in a term) '∞' indicates that jobs are terminated manually after about six hours of execution on the Sequent.

Note: BDD spends 1217.87 seconds for $v = 30$ and $T = 1600$, while CV-D takes 72.77 seconds.

CHAPTER 7

CONCLUSION

In this dissertation, we have presented a new algorithm based on a divide-and-conquer strategy using the concept of cover and a derivational method to solve the equivalence problem. We have proved the correctness of the algorithm and developed heuristics which have been implemented and tested successfully on a large scale of data. We have shown that it is an improvement over the state-of-the-art technology in this area in terms of performance.

Any algorithm for solving this type of problem must incorporate a wide variety of simplification techniques, and perform simplifications as much as possible at every stage of the algorithm. Most big PLAs require an enormous amount of space for representation regardless of representational methodologies, which indicates simplification is essential to the verification process. The algorithm Cover-Merge employs various types of simplification techniques as its essential part. A set of prioritized heuristics is employed to choose the most favorable heuristic which would most likely ensure more simplifications or shorter paths to a goal than any other choices. The falsification heuristics have been empirically found to be quite effective in detecting a nontautology and finding out a counterexample of it. Big speed-ups has been achieved by extensively using these heuristics.

Good representational methods can express properties and characteristics of input circuits in a succinct way and enable efficient logic function manipulations in many applications. However, experimental results show that the issue of simplification is far more important than that of representation for solving this type of problem.

Although a simplified version may lose some of structural information inherent in original circuits, simplification techniques must be incorporated into a verification process as an essential part so that combinatorial explosion would less likely occur. More research on simplification techniques for an economical representation expressing as much structural information as possible would be anticipated in this area.

The algorithm presented in this thesis is widely different from other methods popular on the subject. It is not clear how the algorithm can be combined with other methods such as BDDs. However, this algorithm can be used as an alternative approach where algorithms based on representational schemes such as BDDs experience difficulties in manipulating big PLAs. For future research, we would like to investigate the effectiveness of using Logic Programming to formalize simplification rules and heuristics for Boolean logic verification.

BIBLIOGRAPHY

[1] Akers, S.B., Binary decision diagrams, *IEEE Trans. on computers* **C-27** (6) (1978) 509-516.

[2] Angluin. D., and Valiant, L.G., Fast probabilistic algorithms for Hamiltonian circuits and matchings. *J. Comput. Syst.* **19** (1979) 155-193.

[3] Arvind, V., and Biswas, S., An $O(n^2)$ algorithms for the satisfiability problem of a subset of propositional sentences in CNF that includes all Horn sentences, *Inform. Process. Lett.* **24** (1987) 67-69.

[4] Ashar, P., Devadas, S., and Ghosh, A., Boolean satisfiability and equivalence checking using general binary decision diagrams, *The International Conference on Computer Design*, (Cambridge, Mass., IEEE, New York, 1991) 259-264.

[5] Aspvall, B., Plass, M.F., and Tarjan, R.E., A linear-time algorithm for testing the truth of certain quantified Boolean formulas, *Information Processing Letters* **8** (3) (1979) 121-123.

[6] Azuma, K., Weighted sums of certain dependent variables, *Tohoku Math. J.* **3** (1967) 357-367.

[7] Bayol, C., and Paillet, J., Using TACHE for proving circuits, *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design* **2** (1989) 13-16.

[8] Bitner, J., and Reingold, E., Backtrack Programming Techniques, *Comm. ACM* **18** (1975) 651-655.

[9] Bledsoe, W.W., Non-resolution theorem proving, *Artificial Intelligence* **9** (1977) 1-35.

[10] Bledsoe, W.W., Splitting and reduction heuristics in automatic theorem prov--ing, *Artificial Intelligence* **2** (1971) 55-57.

[11] Boole, G., An investigation of the laws of thought, (Reprint, Dover Publications, 1854).

[12] Brand, D., Exhaustive simulation need not require an exponential number of tests, *IEEE Trans. on Computer-Aided Design* **12** (11) (1993).

[13] Brand, D., and Sasao, T., Minimization of AND-EXOR expressions using rewrite rules, *IEEE Trans. on Computers* **42** (5) (1993).

[14] Brayton, R.K., Hachtel, G.D., McMullen, C.T., and Sangiovanni-Vincentelli, A., Logic minimization algorithms for VLSI synthesis, (Kluwer Academic Publishers, Boston, 1984).

[15] Bryant, R.E., Symbolic boolean manipulation with ordered binary-decision diagrams, *ACM Computing Surveys* **24** (3) (1992) 293-318.

[16] Bryant, R.E., A methodology for hardware verification based on logic simulation, *J. ACM* (Apr, 1991).

[17] Bryant, R.E., Verifying a static RAM design by logic simulation, *Advanced Research in VLSI: Proc. Fifth MIT Conf.* (Jonathan Allen and F.Thomson Leighton, Eds, The MIT Press, Mar 1988).

[18] Bryant, R.E., Graph-based algorithms for boolean function manipulation, *IEEE Trans. on computers* **C-35** (8) (1986) 677-691.

[19] Bryant, R.E., Symbolic manipulation of boolean functions using a graphical representation, *IEEE 22nd Design Automation Conference,* (1985).

[20] Brown, C., Finkelstein, L., and Purdom, P., Backtrack searching in the presence of symmetry, *AAECC-6 Conference Proceedings, Lecture Notes in Computer Science 357* (Springer-Verlag, New York, 1989) 99-110.

[21] Brown, C., and Purdom, P., An empirical comparison of backtracking algorithms, *IEEE Tr. Pattern Anal. Machine Intelligence* **4** (1982) 309-316.

[22] Brown, C., and Purdom, P., An average time analysis of backtracking, *SIAM J. Comput.* **10** (1981) 583-593.

[23] Bugrara, K., and Purdom, P., Average time analysis of clause order backtracking, Indiana University Tech. Rpt. 311 (1990).

[24] Bugrara, K., Pan, Y., and Purdom, P., Exponential average time for the pure literal rule, *SIAM J. Comput.* **18** (1989) 409-418.

[25] Bugrara, K., and Cynthia, B., On the average case analysis of some satisfiability model problems, *Inform. Sciences* **40** (1986) 21-38.

[26] Burch, J.R., Using BDDs to verify multipliers, *Proceedings of the 28th ACM / IEEE Design Automation Conference*, (San Francisco, ACM, New York, 1991) 408-412.

[27] Buttner, W., and Simonis, H., Embedding Boolean expressions into logic programming, *Journal of Symbolic Computation* **4** (1987) 191-205.

[28] Carter, L., Stockmeyer, L., and Wegman, M., The complexity of backtrack searches, *Proceedings of the 17th Annual ACM Symposium on Theory of Computing.* (ACM, New York, 1985) 449-457.

[29] Chakravarty, S., A characterization of binary decision diagrams, *IEEE Trans. on computers* **42** (2) (1993) 129-137.

[30] Chang, C.L.. and Lee, R.C.T., Symbolic Logic and Mechanical Theorem Proving, (Academic Press, New York, 1973).

[31] Chen, Z., A fast and efficient parallel algorithm for finding a satisfying truth assignment to a 2-CNF formula, *Information Processing Letters* **43** (1992) 191-193.

[32] Chvátal, V., and Szemeredi, E., Many hard examples for resolution, *JACM* **35** (1988) 759-768.

[33] Cook, S.A., and M.Luby, M., A simple parallel algorithm for finding a satisfying truth assignment to a 2-CNF formula, *Information Processing Letters* **27** (3) (1988) 141-146.

[34] Cook, S., Dwork, C., and Reischuk, R., Upper and lower time bounds for parallel random access machines without simultaneous writes, *SIAM J. Comput.* **15** (1986) 87-97.

[35] Cook, S., The complexity of theorem-proving procedures, *Proc. 3rd ACM Symp. on Theory of Computing* (1971) 151-158.

[36] Coudert, O., Berthet, C., and Madre, J.C., Verification of sequential machines using Boolean functional vectors, *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design* **2** (1989).

[37] Dalal, M., A hierarchy of tractable satisfiability problems, *Information Processing Letters* **44** (1992) 173-180.

[38] Dalal, M., Tractable deduction in knowledge representation systems, *Proc. Third Internat. Conf. on Principles of Knowledge Representation and Reasoning KR '92* , (Boston, MA, 1992).

[39] Davis, M., Logeman, G., and Loveland, D., A machine program for theorem proving, *Comm. ACM.* **5** (1962) 394-397.

[40] Davis, M., and Putnam, H., A computing procedure for quantification theory, *J. ACM* **7** (1960) 201-215.

[41] Devadas, S., Ma, H.T., and Newton, A.R., On the verification of sequential machines of differing levels of abstraction, *IEEE Trans. on Computer-Aided Design* **7** (1988) 713-722.

[42] Deverchere, P., Madre, J.C., Guignet, J.B., and Currat, M., Functional abstraction and formal proof of digital circuits, *Proc. European Design Automation Conf.* (Mar. 1992) 458-462.

[43] Dincbas, M., Hentenryck, P.V., Himonis, H., Aggoun, A., Graf, T., and Berthier, F., The constraint logic programming language CHIP, *Proceedings in the International Conference on Fifth Generation Computer Systems FGCS-88* (Tokyo, Japan, 1988).

[44] Dowling, W., and Gallier, J., Linear-time algorithms for testing the satisfiability of propositional Horn formulae, *J. Logic Programming* **3** (1984) 267-284.

[45] Even, S., Itai, A., and Shamir, A., On the complexity of timetable and multicommodity flow problems, *SIAM J. Comput.* **5** (4) (1976) 191-700.

[46] Finkel, R.A., and Manber, U., DIB - A distributed implementation of backtracking, *ACM Trans. Prog. Lang. Syst.* **9** (2) (1987) 235-256.

[47] Fisher, A.L., and Bryant, R.E., Performance of COSMOS on the IFIP workshop benchmarks, *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design* **2** (1989) 595-599.

[48] Fleisher, H, and Maissel, L.I., An introduction to array logic, *IBM Journal of Research and Development* **19** (1975) 98-109.

[49] Fortune, S., Hopcroft, J., and Schmidt, E.M., The complexity of equivalence and containment for free single variable program schemes, *Automata, Languages and Programming, Lecture Notes in Computer Science, vol. 62, G. Goos, H. Hartmanis, Ausiello, and Boehm, Eds.* (Springer-Verlag, Berlin, 1978) 227-240.

[50] Franco, J., Elimination of infrequent variables improves average case performance of satisfiability algorithms, Indiana Univ. Tech. Rpt. **294** (1989).

[51] Franco, J., On the occurrence of null clauses in random instances of satisfiability, *Discrete Applied Mathematics,* **41** (1993) 203-209.

[52] Franco, J., and Keutzer, K., Probabilistic analysis of algorithms for stuck-at test generation in PLAs, Indiana University Tech. Rpt. 278 (1989).

[53] Franco, J., On the probabilistic performance of algorithms for the satisfiability problem, *Inform. Proc. Lett.* **23** (1986) 103-106.

[54] Friedman, S.J., and Supowit, K.J., Finding the optimal variable ordering for binary decision diagrams, *Proc. 24th ACM-IEEE Design Automation Conference,* Miami Beach (1987).

[55] Fujita, M., Fujisawa, H., and Matsunaga, Y., Variable ordering algorithms for ordered binary decision diagrams and their evaluation, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* **12** (1) (1993) 6-12.

[56] Fujita, M., et al. Evaluation and improvements of Boolean comparison methods based on binary decision diagrams, *Proceedings ICCAD* (1988) 2-5.

[57] Fujiwara, H., Logic testing and design for testability, (MIT Press, Cambridge MA, 1895).

[58] Galil, Z., On the complexity of regular resolution and Davis-Putnam procedure, *Theoret. Comput. Sci.* **4** (1977) 23-46.

[59] Gallo, G., and Scutella, M.G., Polynomially solvable satisfiability problems, *Inform. Process. Lett.* **29** (5) (1988) 221-227.

[60] Garey, M.R., and Johnson, D.S., Computers and Intractability: A Guide to the Theory of NP-completeness, (Freemann, San Francisco. 1979).

[61] Gelder, A.V., A Satisfiability tester for non-clausal propositional calculus, *Proceedings of the 7th International Conference on Automated Deduction (Lecture Notes in Computer Science)* **170** (New York, Springer-Verlag, 1988).

[62] Goldberg, A., Purdom, P., and Brown, C., Average time analysis of simplified Davis-Putnam procedures, *Info. Proc. Lett. 15,* 1982, 72-75. (Printer errors corrected in **16** 1983) 213.

[63] Goldberg, A., Average case complexity of the satisfiability problem, *Proc. Fourth Workshop on Automated Deduction* (1979) 1-6.

[64] Hachtel, G.D., and Jacoby, R.M., Verification algorithms for VLSI synthesis, *IEEE Trans. on Computer-Aided Design* **7** (1988) 616-640.

[65] Haralick, R.M., and Wu, S.-H., An approximate linear time propagate and divide theorem prover for propositional logic, *International J. Pattern Recognition, Artificial Intelligence* **1** (1987) 141-155.

[66] Hayes, J.P., Computer architecture and organization, (McGraw-Hill Computer Science Series, McGraw-Hill, 1988).

[67] Henschen, L. and Wos, L., Unit refutations and Horn sets, *J. Assoc. Comput. Mach.* **21** (1974) 590-605.

[68] Hooker, J.N., Resolution vs. cutting plane solution of inference problems: Some computational experience, *Operations Research Letters* **7** (1988) 1-7.

[69] Hunt, H.B. III and Stearns, R.E., The complexity of very simple boolean formulas with applications, *SIAM J. Comput.* **19** (1) (1990) 44-70.

[70] Hunter, G., Metalogic : An introduction to the Metatheory of Standard First Order Logic, University of California Press, Berkeley and Los Angeles, California (1971).

[71] Hwang, G-H., Shen, W-Z., Restructuring and logic minimization for testable PLA, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* **12** (4) (1993).

[72] Itai, A., and Makowsky, J., On the complexity of Herbrand's theorem, Tech. Rept. No. 243 Dept. of Computer Science, Israel Institute of Technology (1982).

[73] Iwama, K., CNF satisfiability test by counting and polynomial average time, *SIAM J. Comput.* **18** (1989) 385-391.

[74] Jain, P., and Gopalakrishnan, G., Efficient symbolic simulation-based verification using the parametric form of Boolean expressions, *IEEE Trans. on Computer-Aided Design* **13** (8) (1994) 1005-1015.

[75] Jeong, S.W., Plessier, B., Hachtel, G.D., and Somenzi, F., Variable ordering and selection for FSM traversal, *The International Conference on Computer-Aided Design,* (Santa Clara, Calif., IEEE, New York, 1991) 476-479.

[76] Jones, N.D., and Laaser, W.T., Complete problems for deterministic polynomial time, *Theoret. Comput. Sci.* **3** (1976) 105-117.

[77] Joyner, W.H. Jr., Resolutions strategies and decision procedures, *J. Assoc. Comput. Mach.* **23** (1976) 398-417.

[78] Kalish, D., Montague, R., and Mar, G., Logic Techniques of Formal Reasoning, Harcourt Brace Jovanovich, Inc. N.Y. (1964).

[79] Kam, T., and Subrahmanyam, P.A., Comparing layouts with HDL models: A formal verification technique, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* **14** (4) (1995) 503-509.

[80] Kamath, A.P., Karmarkar, N.K., Ramakrishman, K.G., and Resende, M.G.C., Computational experience with an interior point algorithm on the satisfiability problem, (AT&T Bell Lab., Murray Hill, 1989).

[81] Karnaugh, M., A map method for synthesis of combinational logic circuit, *Trans. AIEE, Comm. and Electronics* **72** (I) (1953) 593-599.

[82] Karp, R.M., and Zhang, Y., A randomized parallel branch-and-bound procedure, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing.* (Chicago, Ill., May 2-4, ACM, New York, 1988) 290-300.

[83] Karp, R.M., and Wigderson, A., On a search problem related to branch-and-bound procedures, *Proceedings of the 27th Symposium on Foundations of Computer Science.* (IEEE, New York, 1986) 19-28.

[84] Karp, R.M., Reducibility among combinatorial problems, *Complexity of Computer Computations,* (Plenum, New York, 1972) 85-103.

[85] Karplus, K., Using if-then-else DAGs for multi-level logic minimization, *Advance Research in VLSI*, (C. Seitz, Ed. MIT Press, Cambridge, Mass., 1989) 101-118.

[86] Knuth, D., Estimating the efficiency of backtracking programs, *Math. Comput.* **29** (1975) 121-236.

[87] Koutsoupias, E., and Papadimitrious, C.H., On the greedy algorithm for satisfiability, *Information Processing Letters* 43 (1992) 53-55.

[88] Kutyłowski, M., Time complexity of boolean functions on CREW PRAMS, *SIAM J. Comput.* **20** (5) (1991) 824-833.

[89] Lam, C.W.H., and Thiel, L., Backtrack search with isomorph rejection and consistency check, *Symbolic Comput.* **7** (1989) 473-486.

[90] Lammens, P., Claesen, L., and Man, H.D., Tautology checking benchmarks: results with TC, *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design* **2** (1989) 13-16.

[91] Lammens, P., TC: A tautology-checker, the kernel of a functional verification system for combinational logic, (Internal Report IMEC, 1987).

[92] Lee, C.Y., Representation of switching circuits by binary-decision programs, *Bell. Syst. Tech. J. 38* (1959) 985-999.

[93] Mackworth, A.K., and Freuder, E.C., The complexity of constraint satisfaction revisited, *Artificial Intelligence* **59** (1993) 57-62.

[94] Madre, J.C., Benchmarks for tautology checking - experimental results, *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design* **2** (1989) 13-16.

[95] Madre, J.C., and Billon, J.P., Proving circuit correctness using formal comparison between expected and extracted behavior, *25th Design Automation Conference, ACM/IEEE* (1988) 205-210.

[96] Malik, S., Wang, A., Brayton, R.K., and Sangiovanni-Vincentelli, A., Logic verification using binary decision diagrams in a logic synthesis environment, *Proceedings ICCAD* (1988) 6-9.

[97] Man, H.D., et al., DIALOG: an expert debugging system for MOS VLSI design, *IEEE Trans. on CAD of int. circuits and systems* **CAD-4** (3) (1985) 303-311.

[98] McCluskey, E.J. Jr., Introduction to the theory of switching circuit, (McGraw-Hill, 1965).

[99] McCluskey, E.J.Jr., Minimization of Boolean functions, *Bell System Tech. J.* **35** (6) (1956) 1417-1444.

[100] McFarland, M.C., Formal verification of sequential hardware: A tutorial, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* **12** (5) (1993) 633-654.

[101] Mead, C., and Conway, L., Introduction to VLSI Systems, (Addison-Wesley, Reading, MA, 1980).

[102] Minato, S., Ishiura, N., and Yajima, S., Fast tautology checking using shared binary decision diagram - Benchmark results, *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design* **2** (1989) 13-16.

[103] Monien, B., and Speckenmeyer, E., Solving satisfiability in less than 2^n step, *Discrete Applied Mathematics* **10** (1985) 287-295.

[104] Moret, B.M.E., Decision trees and diagrams, *Computing Surveys* **14** (4) (1982) 593-623.

[105] Nicol, D., Performance of backtracking, *SIAM J. Comput.* **17** (1980) 114-127.

[106] Oppacher, F., and Suen, E., HARP: A tableau-based theorem prover, *Journal of Automated Reasoning* **4** (1988) 69-100.

[107] Parberry, I., and Yan, P.Y., Improved upper and lower time bounds for parallel random access machines without simultaneous writes, *SIAM J. Comput.* **20** (1991) 88-99.

[108] Payne, R.W., Reticulation and other methods of reducing the size of printed diagnostic keys, *J. Gen. Microbiol.* **98** (1977) 595-597.

[109] Purdom, P., Random satisfiability problems, *International Workshop on Discrete Algorithms and Complexity, The Institute of Electronics, Information and Communication Engineers,* Tokyo, Japan (1989) 253-259.

[110] Purdom, P., and Brown, C., Polynomial average time satisfiability problems, *Information Sciences* **41** (1987) 23-42.

[111] Purdom, P., and Brown, C., The pure literal rule and polynomial average time, *SIAM J. Comput.* **14** (1985) 943-953.

[112] Purdom, P., and Brown, C., An analysis of backtracking with search rearrangement, *SIAM J. Comput.* **12** (1983) 717-733.

[113] Purdom, P., Search rearrangement backtracking and polynomial average time, *Artificial Intelligence* **21** (1983) 117-133.

[114] Purdom, P., Brown, C., and Robertson, E., Backtracking with multi-level dynamic search rearrangement, *Acta Informatica* **15** (1981) 99-113.

[115] Purdom, P., Tree Size by Partial Backtracking, *SIAM J. Comput.* **7** (1978) 481-491.

[116] Purdom, P., Search Rearrangement backtracking and polynomial average time, *Artificial Intelligence* **21** (1983) 117-133.

[117] Quine, W.V., The problem of simplifying truth functions, *Am. Math. Monthly* **59** (8) (1952) 521-531.

[118] Quine, W.V.O., Methods of logic, (Holt, Rinehart, Winston, New York, 1950).

[119] Ranade, A., Optimal speedup for backtrack on a butterfly network, *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures.* (ACM, New York, 1991) 44-48.

[120] Ranade, A., A simpler analysis of the Karp-Zhang parallel branch-and-bound method, Tech. rep. No. 586. Computer Science Division, Univ. California at Berkeley, Berkeley, Calif., (1990).

[121] Ravi, S.S., and Lloyd, E.L., Graph theoretic analysis of PLA folding heuristics, *Journal of Computer and System Sciences* **46** (1993) 326-348.

[122] Reischuk, R., Fast evaluation of boolean functions, (Wiley-Teubner Series in Computer Science, John Wiley, New York, 1987).

[123] Roth, J.P., Computer Logic, Testing and Verification, (Computer Science Press, Dotomac, Md, 1980).

[124] Roth, J.P., Hardware Verification, *IEEE Trans. on Computers* **C-26** (1977) 1292-1294.

[125] Roth, J.P., VERIFY: An algorithm to verify a computer design, *IBM Tech. Disclosure Bull.* **15** (1973) 2646-2648.

[126] Roth, J.P., Diagnosis of automata failures: A calculus and a method, *IBM J. Res. Develop.* (1966).

[127] Sasao, T., EXMIN2: A simplification algorithm for exclusive-or-sum-of products expressions for multiple-valued-input two-valued-output functions, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* **12** (5) (1993).

[128] Sentovich, K.S., Moon, C., Savoj, H., Brayton, R., and Sangiovanni-Vincentelli, A., Sequential circuit design using synthesis and optimization, *Proc. Int. Conf. Computer Design* (Oct. 1992) 328-333.

[129] Shannon, C.E., A symbolic analysis of relay and switching circuits, *Trans. AIEE* **57** (1938) 713-723.

[130] Shen, W-Z., Hwang, G-H, and Jan, Y-J., Design of pseudoexhaustive testable PLA with low overhead, *IEEE Trans. on Computers* **42** (7) (1993).

[131] Simonis, H., and Provost, T.L., Circuit verification in CHIP: Benchmark results, *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design* **2** (1989) 13-16.

[132] Simonis, H., Formal verification of multipliers, *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design* **2** (1989) 13-16.

[133] Simonis, H., Test generation using the constraint logic programming language CHIP, *Proceedings of the 6th International Conference on Logic Programming* (Lisboa, Portugal, 1989).

[134] Simonis, H., Nguyen, H.N., and Dincbas, M., Verification of digital circuits using CHIP. *Proceedings of the IFIP WG 10.2 International Working Conference on the Fusion of Hardware Design and Verification, G.J. Milne, Ed* (Glasgow, Scotland, IFIP, North Holland, 1988).

[135] Simonis, H., and Dincbas, M., Using an extended Prolog for digital circuit design, *IEEE International Workshop on AI Applications to CAD Systems for Electronics* (Munich, W.Germany, 1987) 165-188.

[136] Smith, G.L., Bahnsen, R.J., and Halliwell, H., Boolean comparison of hardware and flowcharts, *IBM J. Res. Dev.* **26** (1982) 106-116.

[137] Speckenmeyer, E., Monien, B., and Vornberger, O., Superlinear speedup for parallel backtracking, *Proc. of supercomputing 87, Lecture Notes in Computer Science* **297** (1987) 985-993.

[138] Stavridou, V., Formal methods and VLSI engineering practice, *The computer Journal* **37** (2) (1994) 96-113.

[139] Stearns, R.E., and Hunt, H.B. III, On the complexity of the satisfiability problem and the structure of NP, State University of New York at Albany Technical Report TR 86-21 (1986).

[140] Spencer, J., Ten Lectures on the Probabilistic Method, *SIAM,* (Philadelphia, Pa., 1987).

[141] Supowit, K.J., and Friedman, S.J., A new method for verifying sequential circuits, *Proceedings of the 23rd Design Automation Conference* (1986).

[142] Tanaka, Y., A dual algorithm for the satisfiability problem, *Information Processing Letters* **37** (1991) 85-89.

[143] Tarnlund, S.A., Horn clause computability, *BIT* **17** (1977) 215-226.

[144] Thiel, L., Lam, C., and Swiercz, S., Using a Cray-1 to perform backtrack search, *Proc. of the second international conference on supercomputing* **3** (1987) 92-99.

[145] Trier, U., Additive weights of a special class of nonuniformly distributed backtrack trees, *Information Processing Letters* **42** (1992) 67-76.

[146] Verkest, D., and Claesen, L., Special benchmark session on Tautology checking, *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design* **2** (1989) 13-16.

[147] Vlach, F., Simplification in a satisfiability checker for VLSI applications, *Journal of Automated Reasoning* **10** (1) (1993) 115-136.

[148] Vlach, F., A tautology checker for VLSI applications that uses rule-based simplification and Directed backtracking, Technical Report N-90-0010, Department of Computer Science, University of North Texas (1990).

[149] Vlach, F., A Note on the INSTEP Tautology checker and the IFIP and ISCAS benchmarks, *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design* **2** (1989) 13-16.

[150] Walker, R.A., The status of high-level synthesis, *IEEE design & Test of Computers* (Winter, 1994) 42-54.

[151] Wang, H., Toward mechanical mathematics, *IBM J. Research Dev.* **4** (1960) 2-22.

[152] Wegener, I., On the complexity of branching programs and decision trees for clique functions, *J. ACM* **35** (2) (1988) 461-471.

[153] Wei, R.-S., and Sangiovanni-Vincentelli, A., PROTEUS: A logic verification system for combinational circuits, *Proceedings 1986 IEEE International Conference* (1986) 350-358.

[154] Windley, P.J., Formal modeling and verification of microprocessors, *IEEE Trans. on Computers* **44** (1) (1995) 54-72.

[155] Wolfram, D.A., Forward checking and intelligent backtracking, *Information Processing Letters* **32** (1989) 85-87.

[156] Xu, Y., Abd-El-Barr, M., and McCrosky, C., Graph-based output phase assignment for PLA minimization, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* **14** (5) (1995).

[157] Yamasaki, S., and Doshita, S., The satisfiability problem for a class consisting of Horn sentences and some non-Horn sentences in propositional logic, *Inform. and Control* **59** (1-3) (1983) 1-12.