國立臺灣大學電機資訊學院電子工程學研究所

碩士論文

Graduate Institute of Electronics Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

利用可滿足性求解法與克雷格內插法之大尺度亞氏函式拆解

Large Scale Ashenhurst Decomposition via SAT Solving and Craig Interpolation

林炫伯

Hsuan-Po Lin

指導教授：江介宏 博士

Advisor: Jie-Hong Roland Jiang, Ph.D.

中華民國 98 年 6 月

June, 2009

Acknowledgements

I am grateful to many people who made this thesis possible. I would like to thank my advisor, Dr. Jie-Hong Roland Jiang, with his enthusiasm, his academic experiences, and his inspiration. Throughout the past two years, he provides encouragements, useful advise, and lots of good ideas. I would have lost without him.

The members of ALCom Lab, I-Hsin Chen, Wei-Lun Hung, Sz-Cheng Huang, Chia-Chao Kan, Ruei-Rung Lee, Chih-Fan Lai, Meng-Yan Li, Jane-Chi Lin, and Fu-Rong Wu, have been my great memory in academic journey. We had a lot of inspiring discussions and joyful cooperations. I would also like to thank all of my friends who helped me get through difficult time and provided invaluable supports.

I cannot end this acknowledgement without thanking my parents and brother for their constant supports and encouragements no matter what happened. To them I dedicate this thesis.

Hsuan-Po Lin

National Taiwan University

June 2009

利用可滿足性求解法與克雷格內插法之大尺度亞氏函式拆解

研究生：林炫伯　　　指導教授：江介宏博士

國立台灣大學電子工程學研究所

摘要

函式拆解著眼於將一個布林函式拆解成一系列較小的子函式。在本篇論文裡面，我們著重在亞氏函式拆解，這是一種因為他的簡易性而有許多實際應用的常見函式拆解法。我們將函式拆解問題包裝成可滿足性求解問題，更進一步的採用克雷格內插法以及函式相依性計算來找出相對應的子函式。在我們採用可滿足性求解法為核心的研究中，輸入變數分組的過程可以被自動的處理，並且嵌入我們的函式拆解演算法中。我們也可以自然的將我們的演算法延伸，應用在允許共用輸入變數與多輸出變數的函式拆解問題上，這些問題在以往採用二元決策圖為核心資料結構的演算法中都很難被解決。實驗結果顯示，我們提出的演算法可以有效的處理輸入變數達到三百個之多的函式。

關鍵字：布林函式、函式拆解、可滿足性求解法、克雷格內插法、函式相依性

Large Scale Ashenhurst Decomposition via SAT Solving and Craig Interpolation[1]

Student: Hsuan-Po Lin   Advisor: Dr. Jie-Hong Roland Jiang

Graduate Institute of Electronics Engineering

National Taiwan University

Abstract

Functional decomposition aims at decomposing a Boolean function into a set of smaller sub-functions. In this thesis, we focus on Ashenhurst decomposition, which has practical applications due to its simplicity. We formulate the decomposition problem as SAT solving, and further apply Craig interpolation and functional dependency computation to derive composite functions. In our pure SAT-based solution, variable partitioning can be automated and integrated into the decomposition procedure. Also we can easily extend our method to non-disjoint and multiple-output decompositions which are hard to handle using BDD-based algorithms. Experimental results show the scalability of our proposed method, which can effectively decompose functions with up to 300 input variables.

Keywords: Boolean function, functional decomposition, SAT solving, Craig interpolation, functional dependency.

[1] The preliminary version of this thesis appears in [18].

Contents

Acknowledgements i

Chinese Abstract ii

Abstract iii

List of Figures viii

List of Tables ix

1 Introduction 1

1.1 Thesis Overview 1

1.2 Related Work 7

1.3 Our Contributions 8

1.4 Thesis Organization 10

2 Preliminaries 11

2.1 Functional Decomposition 12

2.1.1 Decomposition Chart 13

2.2 Functional Dependency 15

2.3 Satisfiability and Interpolation 16

2.3.1 Conjunctive Normal Form 16

2.3.2 Circuit to CNF Conversion 16

2.3.3 Propositional Satisfiability 18

2.3.4 Refutation Proof 19

2.3.5 Craig interpolation 22

3 Main Algorithms 27

3.1 Single-Output Ashenhurst Decomposition 27

3.1.1 Decomposition with Known Variable Partition 28

3.1.2 Decomposition with Unknown Variable Partition 37

3.2 Multiple-Output Ashenhurst Decomposition 43

3.2.1 Shared Variable Partition 45

3.3 Beyond Ashenhurst Decomposition 46

3.4 Decomposition under Don't-Cares 47

3.5 Implementation Issues 48

3.5.1 Minimal UNSAT Core Refinement 48

3.5.2 Balanced Partition 48

3.5.3 Elimination of Equality Constraint Clauses 49

3.6 A Complete Example 49

4 Experimental Results 54

4.1 Single- and Two-Output Ashenhurst Decomposition 54

4.2 Quality of Variable Partition 59

4.3 Fast Variable Partitioning 61

4.4 n-Output Ashenhurst Decomposition 67

4.5 Quality of Composite Functions 68

5 Conclusions and Future Work 71

Bibliography 74

List of Figures

1.1 Ashenhurst decomposition 6

2.1 The diagonal decomposition chart due to introduce of common variable 14

2.2 Circuit to CNF representation 17

2.3 Refutation proof and the unsatisfiable core 22

2.4 Craig interpolation 23

2.5 Interpolant construction of an unsatisfiable formula $\varphi = (\neg c)(\neg b + a)(c + \neg a)(b)$ 26

3.1 Circuit representation of Formulas (3.2) and (3.3) 30

3.2 (a) Relation characterized by $\psi_A(X_G^1, X_G^2, c)$ for some $c \in [\![X_C]\!]$ (b) Relation after cofactoring $\psi_A(X_G^1 = a, X_G^2, c)$ with respect to some $a \in [\![X_G^1]\!]$ 30

3.3 Example circuit used in the complete example 50

3.4 The decomposition chart of the example circuit 51

3.5 (a) Relation of the derived interpolant function, (b) Relation after cofactor by $(a^1, b^1) = (0, 0)$ 52

3.6 (a) The base functions, (b) h function derived by functional dependency 53

4.1 Best variable partition found in 60 seconds – without minimal UNSAT core refinement 58

4.2 Best variable partition found in 60 seconds – with minimal UNSAT core refinement 58

4.3 Variable partition qualities under four different efforts 60

4.4 First-found valid partition – without minimal UNSAT core refinement 61

4.5 First-found valid partition – with minimal UNSAT core refinement . . 62

4.6 Comparison of disjointness between different partition efforts 65

4.7 Comparison of disjointness between different partition efforts 66

4.8 Disjointness versus total variables of the best found partition in 60 seconds with minimal UNSAT core operation 66

4.9 Runtime of multi-output Ashenhurst decomposition 68

List of Tables

4.1 Single-output Ashenhurst decomposition 55

4.2 Two-output Ashenhurst decomposition 56

4.3 Variable distribution in different partition efforts 64

4.4 Circuit information of s9234 69

Chapter 1

Introduction

1.1 Thesis Overview

Functional decomposition [1, 7, 14, 22] plays an important role in the analysis and design digital systems. It refers to the process of breaking a complex function into parts, which are less complex than the original function and are relatively independent to each other. In addition the behavior of the original function can be reconstructed if we compose these parts together. In logic synthesis, a complex function is decomposed into a set of sub-functions, such that each sub-function is easier to analyze, to understand, and to further synthesize. Functional decomposition has long been recognized as a pivotal role in LUT-based FPGA synthesis. It also has various applications to the minimization of circuit communication complexity.

Functional decomposition can be classified as follows:

- The function to be decomposed can be a *completely* or an *incompletely* specified function.
- The function to be decomposed can be a *single-* or *multiple-output* function.
- A decomposition is *disjoint* if the sub-functions do not share common input variables; otherwise, it is *non-disjoint*.
- A decomposition is called *Ashenhurst decomposition* or *simple decomposition* if the topology of the decomposition is $f(X) = h(g(X_G), X_H)$, where $X = X_G \cup X_H$ and g is a single-output function; a decomposition is called *bi-decomposition* if the topology is $f(X) = h(g_1(X_A), g_2(X_B))$, where $X = X_A \cup X_B$ and h is a two-input gate.

In this thesis, our method focuses on completely specified functions. In addition the proposed method deals with multiple-output and non-disjoint decompositions. Furthermore, we pay our attention to Ashenhurst decomposition.

The functional decomposition problem was first formulated by Ashenhurst in 1959 [1]. He visualized the decomposition feasibility with a decomposition chart, which is a two-dimensional Karnaugh map with rows corresponding to variables in the free set and columns corresponding to variables in the bound set. The decomposition chart is used to determine whether a given function f can be simply disjointly decomposed with respect to a set of given bound set variables. Ashenhurst reduced the chart by merging all identical columns, and he showed that there exists a simple disjoint decomposition if and only if there are at most two distinct columns in the reduced decomposition chart. The disadvantage of Ashenhurst's method is that we have to construct every chart with respect to every variable partition we consider. If we consider all the variable partitions, a function with n variables would have $O(2^n)$ charts since every variable can be in either the bound set or free set.

Roth and Karp proposed a cover-based method [22], trying to reduce the memory requirement of Ashenhurst's method. They used *covers* to represent the functions. By cover manipulations, the minterms of bound set variables will be mapped into equivalence classes. These equivalence classes are in one-to-one correspondence with the distinct columns in Ashenhurst's method. Nevertheless, the process time of the algorithm is still a problem except we restrict the size of the bound set variables to be up-bounded by some constant k. A typical value of k is 5 which is the common input size of a LUT.

Recent progress on function manipulation using BDDs makes the BDD data structure becomes a popular tool to handle the functional decomposition problem. Lai, Pedram, and Vrudhula [15] proposed a fast BDD-based method to implement the Roth-Karp algorithm. They used BDDs to represent Boolean functions, and showed that every variable ordering in the BDD implies a variable partition of the decomposition of the function. They ordered all the bound set variables above the cut, and ordering all the free set variables in and below the cut. Based on Lai's method, Stanion and Sechen proposed a method [25] to enumerate all the possible cuts in the BDD in order to find a good decomposition. The enumeration can be achieved by constructing a characteristic function for the set of cuts, then using a branch-and-bound procedure in order to find a cut which produces the best decomposition.

Identifying the common sub-functions in decomposing a function set is an important issue in multiple-output functional decomposition. The first approach to multiple-output functional decomposition was proposed by Karp [14]. However Karp's multiple-output approach works only for two-output functions. In order to overcome the limitation of Karp's method, researchers proposed several ways to handle the multiple-output decomposition problem. Wurth, Eckl, and Antreich presented an algorithm [29] based on the concept of a preferable decomposition function, which is a decomposition function valid for a single output and with the potential to be shared by other output functions to handle multiple-output decomposition. They showed that the construction of all preferable decomposition functions can be achieved by partitioning the minterms of bound set variables into some global classes, and this information can be used to identify the common preferable decomposition functions of a multiple-output function.

Sawada, Suyama, and Nagoya proposed a BDD-based algorithm [24] to deal with the issue of identifying common sub-functions. They proposed a effective Boolean resubstitution technique based on support minimization to effectively identify the common LUTs from a large amount of candidates. However when the size of the support variables is large, the examination becomes time-consuming and sometimes fails due to memory explosion.

Most prior work on functional decomposition used BDD as the underlying data structure. BDD can be used to handle the functional decomposition problem with proper variable ordering, which the size of the BDD is under an acceptable threshold. However in some cases, the BDD-based algorithm has several limitations:

- Firstly, there are memory explosion problems for BDD-based algorithms. BDD can be of large size in representing a Boolean function. In the worst case, BDD size is exponential to the number of variables. When special variable ordering rules need to be imposed on BDDs for functional decomposition, the memory size needed in BDD computation may not be improved by changing a suitable variable ordering. Therefore it is typical that a function under decomposition using BDD as the underlying data structure can have just a few variables.

- Secondly, variable partitioning needs to be specified *a priori*, and cannot be automated as an integrated part of the decomposition process. Decomposability under different variable partitions need to be analyzed case by case. In order to enumerate different variable partitions effectively and keep the size of BDD reasonably small, the set of bound set variables cannot be large. Otherwise, the computation time will easily over the pre-specified threshold.

- Thirdly, for BDD-based approaches, non-disjoint decomposition cannot be handled easily. In practical, decomposability needs to be analyzed by cases exponential to the number of joint (or common) variables.
- Finally, even though multiple-output decomposition [30] can be converted to single-output decomposition [12], BDD sizes may grow largely in this conversion.

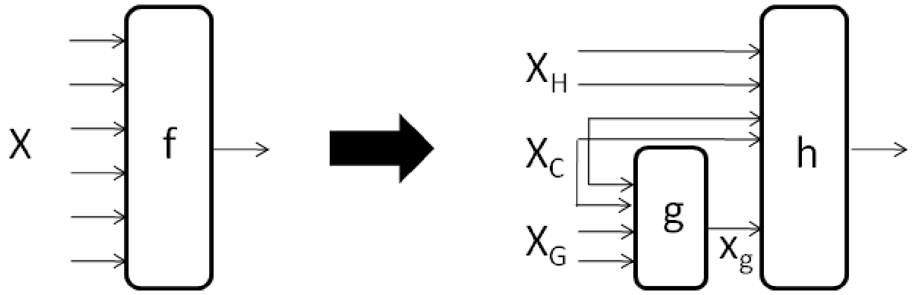


Figure 1.1: Ashenhurst decomposition

The above limitations motivate the need for new data structures and computation methods for functional decomposition. This thesis shows that, when Ashenhurst decomposition [1] is considered, these limitations can be overcome through satisfiability (SAT) based formulation. Ashenhurst decomposition is a special case of functional decomposition, where, as illustrated in Figure 1.1, a function $f(X)$ is decomposed into two sub-functions $h(X_H, X_C, x_g)$ and $g(X_G, X_C)$ with $f(X) = h(X_H, X_C, g(X_G, X_C))$. For general functional decomposition, the function g can be a functional vector $(g_1, \ldots, g_k)$ instead. It is the simplicity that makes Ashenhurst decomposition particularly attractive in practical applications.

The techniques we used in this thesis, in addition to SAT solving, include Craig interpolation [6] and functional dependency [16]. Specifically, the decomposability of function f is formulated as SAT solving, the derivation of function g is by Craig interpolation, and the derivation of function h is by functional dependency.

1.2 Related Work

Aside from the related prior work using BDD as underlying data structure, we compare some related work on functional decomposition and Boolean matching using SAT-based techniques. In bi-decomposition [17], a function f is decomposed into $f(X) = h(g_1(X_A, X_C), g_2(X_B, X_C))$ under variable partition $X = \{X_A|X_B|X_C\}$, where function h is known as *a priori* and is fixed to be special function types, which can be two-input OR, AND, XOR gates, etc.. The functions g_1 and g_2 are the unknowns to be computed. Compared with bi-decomposition, Ashenhurst decomposition $f(X) = h(X_H, X_C, g(X_G, X_C))$ focuses on both unknown functions h and g, where no any fixed type of gates are specified. The Ashenhurst decomposition problem needs be formulated and solved differently while the basic technique used in our thesis is similar to that in [17].

FPGA Boolean matching, see, e.g., [3], is a subject closely related to functional decomposition. In [19], Boolean matching was achieved with SAT solving, where quantified Boolean formulas were converted into CNF formulas. In order to eliminate the universal quantifiers, the CNF formulas will be duplicated exponential times related to the input variable sizes. The intrinsic exponential explosion in formula sizes limits the scalability of the approach. Typically, this method cannot handle functions with input variables greater than 10. To overcome this problem, a two-stage SAT-based Boolean matching algorithm [26] and an implicant-based method [4] were proposed to improve the limitation of methods mentioned in [19]. Different from the above algorithms mainly focus on completely specified functions, Wang and Chan proposed a SAT-based Boolean matching method [28] to handle functions with don't-cares. On the other hand, our method may provide a partial solution to the Boolean matching problem, at least for some special PLB configurations similar to the topology of Ashenhurst decomposition.

1.3 Our Contributions

Compared with BDD-based methods, the proposed SAT-based algorithm is advantageous in the following aspects.

- Firstly, it does not suffer from the memory explosion problem and is scalable to large functions. Experimental results show that Boolean functions with more than 300 input variables can be decomposed effectively.

- Secondly, it needs not be specified *a priori* when variable partitioning, and can be automated and derived on the fly during decomposition. Hence the size of the bound set variables X_G needs not be small. Bound set variables X_G can be as large as free set variables X_H to obtain a more balanced variable partition.

- Thirdly, it works for non-disjoint decomposition naturally. The automated variable partition process in our method can indeed generate a non-disjoint variable partition. In practical, we wish the number of the common (non-disjoint) variables X_C to be small, thus we further propose an UNSAT core refinement process to heuristically reduce the number of common variables as much as possible.

- Finally, it can be easily extended to multiple-output Ashenhurst decomposition if we assume that all of the output functions f_i share the same variable partition $X = \{X_H|X_G|X_C\}$.

However when generalizing our method to functional decomposition beyond Ashenhurst's special case, both SAT-based formula size and computation time of SAT solving grow. This is the limitation of our proposed method.

As interconnects become a dominating concern in modern nanometer IC designs, scalable decomposition methods play a pivotal role in circuit communication minimization. While functional decomposition breaks the original function into smaller and relatively independent sub-functions, the communication complexity between these sub-functions are greatly reduced proportional to the number of interconnecting wires between these parts.

With the advantages of the proposed method, hierarchical logic decomposition could be made feasible in practice. In addition, our results may provide a new way on scalable Boolean matching for heterogeneous FPGAs as well as topologically constrained logic synthesis.

1.4 Thesis Organization

This thesis is organized as follows. Chapter 2 introduces essential preliminaries. Our main algorithms are presented in Chapter 3, and evaluated with experimental results in Chapter 4. Finally, Chapter 5 concludes the thesis and outlines future work.

Chapter 2

Preliminaries

In this chapter, we provide the necessary and sufficient background needed in this thesis. As conventional notation, in this thesis, sets are denoted in upper-case letters, e.g., S; set elements are in lower-case letters, e.g., $e \in S$. The cardinality of S is denoted as $|S|$. A partition of a set S into $S_i \subseteq S$ for $i = 1, \ldots, k$ (with $S_i \cap S_j = \emptyset, i \neq j$, and $\bigcup_i S_i = S$) is denoted as $\{S_1|S_2|\ldots|S_k\}$. For a set X of Boolean variables, its set of valuations (or truth assignments) is denoted as $[\![X]\!]$, e.g., $[\![X]\!] = \{(0,0),(0,1),(1,0),(1,1)\}$ for $X = \{x_1, x_2\}$.

2.1 Functional Decomposition

Definition 2.1 Given a completely specified Boolean function f, variable x is a *support variable* of f if $f_x \neq f_{\neg x}$, where f_x and $f_{\neg x}$ are the positive and negative cofactors of f on x, respectively.

Definition 2.2 A set $\{f_1(X), \ldots, f_m(X)\}$ of completely specified Boolean functions is *(jointly) decomposable* with respect to some variable partition $X = \{X_H | X_G | X_C\}$ if every function f_i, $i = 1, \ldots, m$, can be written as

$$f_i(X) = h_i(X_H, X_C, g_1(X_G, X_C), \ldots, g_k(X_G, X_C))$$

for some functions h_i, g_1, $\ldots$, g_k with $k < |X_G|$. The decomposition is called *disjoint* if $X_C = \emptyset$, and *non-disjoint* otherwise.

For $m = 1$, it is known as *single-output decomposition*, and for $m > 1$, *multiple-output decomposition*. Note that $h_1, \ldots, h_m$ share the same functions $g_1, \ldots, g_k$ if we are dealing with multiple-output decomposition problem. The so-called *Ashenhurst decomposition* [1] is for $k = 1$, which g function has only one output bit.

Note that, for $|X_G| = 1$, there is no successful decomposition because of the violation of the criterion $k < |X_G|$. On the other hand, the decomposition trivially holds if $X_C \cup X_G$ or $X_C \cup X_H$ equals X. The corresponding variable partition is called *trivial*. This paper is concerned about decomposition under non-trivial variable partition. Moreover, we focus on Ashenhurst decomposition.

The decomposability of a set $\{f_1, \ldots, f_m\}$ of functions under the variable partition $X = \{X_H|X_G|X_C\}$ can be analyzed through the so-called *decomposition chart*, consisting of a set of matrices, one for each member of $[\![X_C]\!]$. The rows and columns of a matrix are indexed by $\{1, \ldots, m\} \times [\![X_H]\!]$ and $[\![X_G]\!]$, respectively. For $i \in \{1, \ldots, m\}$, $a \in [\![X_H]\!]$, $b \in [\![X_G]\!]$, and $c \in [\![X_C]\!]$, the entry with row index (i, a) and column index b of the matrix of c is of value $f_i(X_H = a, X_G = b, X_C = c)$.

Proposition 2.1 [1,7,14] A set $\{f_1, \ldots, f_m\}$ of Boolean functions is decomposable as

$$f_i(X) = h_i(X_H, X_C, g_1(X_G, X_C), \ldots, g_k(X_G, X_C))$$

for $i = 1, \ldots, m$ under variable partition $X = \{X_H|X_G|X_C\}$ if and only if, for every $c \in [\![X_C]\!]$, the corresponding matrix of c has at most 2^k column patterns (i.e., at most 2^k different kinds of column vectors).

2.1.1 Decomposition Chart

Given a variable partition $X = \{X_1|X_2\}$, we want to check if a decomposition $f = h(g(X_1), X_2)$ exists. To do this, we build a table called *decomposition chart*. Decomposition chart is a table rearrange from the K-map of a given function f. The table has $2^{|X_1|}$ columns correspond to input vectors of X_1 and $2^{|X_2|}$ rows correspond to input vectors of X_2. Each entry in the decomposition chart represents a function value whose input value is a combination of the index of corresponding rows and columns. Moreover, if we allow some variables can be common in both the row and column, the resulting decomposition chart will be formed by some sub-chart in a diagonal way.

In our Ashenhurst decomposition, the corresponding decomposition chart of the given function f uses input variables of g as its column index variable, and input variables of h as its row index variables. The whole decomposition chart of f is composed by some sub-charts in diagonal way. Each sub-chart corresponds to one valuation of X_C. Hence if there are k common variables, the number of sub-chart will be 2^k. For example if we have one variable in X_C, one variable in X_G, and one variable in X_H. The corresponding decomposition chart is shown in Figure 2.1.

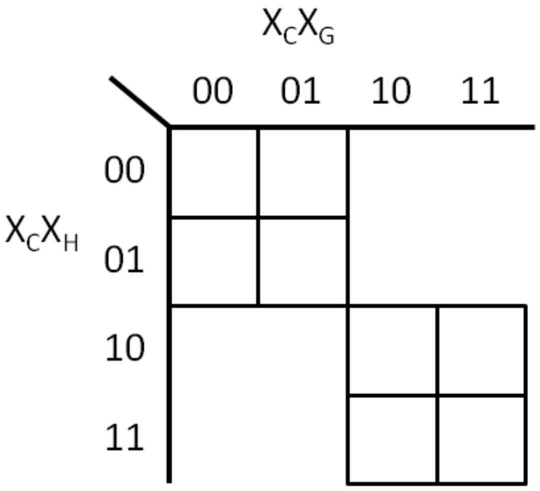


Figure 2.1: The diagonal decomposition chart due to introduce of common variable

2.2 Functional Dependency

Definition 2.3 Given a Boolean function $f : \mathbb{B}^m \to \mathbb{B}$ and a vector of Boolean functions $G = (g_1(X), \ldots, g_n(X))$ with $g_i : \mathbb{B}^m \to \mathbb{B}$ for $i = 1, \ldots, n$, over the same set of variable vector $X = (x_1, \ldots, x_m)$, we say that f *functionally depends* on G if there exists a Boolean function $h : \mathbb{B}^n \to \mathbb{B}$, called the *dependency function*, such that $f(X) = h(g_1(X), \ldots, g_n(X))$. We call functions f, G, and h the *target function*, *base functions*, and *dependency function*, respectively.

Note that functions f and G are over the same domain in the definition; h needs not depend on all of the functions in G.

The necessary and sufficient condition of the existence of the dependency function h is given as follows.

Proposition 2.2 [11] Given a target function f and base functions G, let $h^0 = \{a \in \mathbb{B}^n : a = G(b) \text{ and } f(b) = 0, b \in \mathbb{B}^m\}$ and $h^1 = \{a \in \mathbb{B}^n : a = G(b) \text{ and } f(b) = 1, b \in \mathbb{B}^m\}$. Then h is a feasible dependency function if and only if $\{h^0 \cap h^1\}$ is empty. In this case, h^0, h^1, and $\mathbb{B}^n \backslash \{h^0 \cup h^1\}$ are the off-set, on-set, and don't-care set of h, respectively.

By Proposition 2.2, one can not only determine the existence of a dependency function, but also deduce a feasible one. A SAT-based formulation of functional dependency was given in [16]. It forms an important ingredient in part of our computation.

2.3 Satisfiability and Interpolation

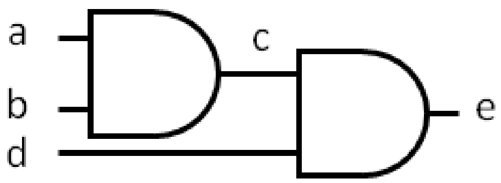
2.3.1 Conjunctive Normal Form

A formula is said to be in Conjunctive Normal Form (CNF) if it is a conjunction of clauses. Each clause is a disjunction of one or more literals, where a literal is the occurrence of a variable x or its complement $\neg x$. Without loss of generality, we shall assume that there are no any equivalent or complementary literals in one clause. For example, the formula $(x + \neg y)(\neg x + y)$ is a CNF representation of the equality function for variable x and y. The formula has 2 clauses, and each clause has 2 literals.

The CNF is similar to Product of Sum (POS) representation in the circuit theory. It is one of the major contributing factor for the recent success of the Boolean Satisfiability (SAT) problem. The CNF representation of a SAT problem provides a data structure for efficient implementation of various techniques used in most popular SAT solvers.

2.3.2 Circuit to CNF Conversion

The CNF formula of a combinational circuit can be constructed in linear time by introducing some intermediate variables. The CNF formula of a combinational circuit is the conjunction of the CNF formulas for each gate output, where the CNF formula for each gate denotes the valid input and output assignments of the gate. The detail information for converting a circuit to CNF representation can be found in [27].



$$\begin{aligned}\varphi &= (a+\neg c)(b+\neg c)(\neg a+\neg b+c)\\ &\quad (c+\neg e)(d+\neg e)(\neg c+\neg d+e)\end{aligned}$$

(a) Consistent assignment

$$\begin{aligned}\varphi' &= (a+\neg c)(b+\neg c)(\neg a+\neg b+c)\\ &\quad (c+\neg e)(d+\neg e)(\neg c+\neg d+e)(\neg e)\end{aligned}$$

(b) With property $e = 0$

Figure 2.2: Circuit to CNF representation

Figure 2.2 shows an example of a simple circuit with the corresponding CNF formula indicating the truth assignment and the CNF formula with some given properties. As can be seen, the CNF formula can be divided into two parts, each part is a set of clauses for one particular gate. Two gates share the same intermediate variable c. So the CNF representation of the circuit is the conjunction of the sets of clauses for each particular gate. Hence, given a combinational circuit it is straightforward to construct the CNF formula for the circuit as well as the CNF formula for proving a given property of a the circuit.

2.3.3 Propositional Satisfiability

In the beginning of this subsection, we firstly define some terminologies of propositional satisfiability problem in the scope of modern SAT solvers. Let $V = \{v_1, \ldots, v_k\}$ be a finite set of *Boolean variables*, where each $v_i \in B = \{0, 1\}$. A *SAT instance* is a set of Boolean formulas in CNF. A *assignment* over V gives each variable $v_i \in V$ a Boolean value either True(1) or False(0). A SAT instance is said to be *satisfiable* is there exists an assignment over V such that the CNF formula is evaluated as True. More specifically, each clause of the CNF formula is evaluated as True. Otherwise, it is *unsatisfiable.* A *SAT problem* is a decision problem asked the given SAT instance is satisfiable or not. A *SAT solver* is designed to answer the SAT problem.

SAT was the first known NP-complete problem, as proved by Stephen Cook in 1971 [5]. The problem remains NP-complete even if the SAT instance is written in conjunctive normal form with 3 variables per clause, yielding the 3SAT problem. Most of the popular modern SAT solvers use David-Putnam-Logemann-Loveland (DPLL) [8,9] procedure as the basic algorithm to solve the SAT problem. DPLL is tested to solve large propositional satisfiable problem efficiently.

The basic idea of DPLL procedure in solving a SAT problem is to branch on variables V until a conflict arises or a satisfying assignment is derived. Once a conflict arises, DPLL chooses another branch to keep testing the satisfiability. If the variable value on a certain branch results in a satisfying assignment over the SAT problem, the DPLL procedure stops the rest branching step immediately, then returns the answer satisfiable as well as the satisfying assignment. Otherwise, DPLL procedure needs to test all the branches to report the unsatisfiability. In this thesis, we use MiniSat [10] developed by Niklas Eén and Niklas Sörensson as the underlying SAT solver in the experiments.

2.3.4 Refutation Proof

Some of the modern SAT solvers generate a refutation proof to demonstrate the unsatisfiability of a SAT instance. Refutation proof is a series of resolution steps show the unsatisfiable SAT instance will eventually imply an empty clause. Each step in these series of steps called resolution. The detailed definition of resolution and other terminology are shown in this subsection.

Definition 2.4 Let $(v \vee c_1)$ and $(\neg v \vee c_2)$ are two clauses. Where v is a Boolean variable, v and $\neg v$ are literals, c_1 and c_2 are disjunction of other literals different from v and $\neg v$. The *resolution* of these two clauses on variable v is a new clause $(c_1 \vee c_2)$. The variable v here is called a *pivot variable*, and $(c_1 \vee c_2)$ is *resolvent.* The resolvent exists when only one pivot variable exists. In other word, the resolvent can not be a tautological clause.

For example, clauses $(a \vee \neg b)$ and $(\neg a \vee b)$ on variable a has no resolvent since clause $(\neg b \vee b)$ is tautological. Also, resolution on variable v over two given clauses is similar to existential quantification on variable v. That is, $\exists v.(v \vee c_1) \wedge (\neg v \vee c_2) = (c_1 \vee c_2)$.

Proposition 2.3 The conjunction of $(v \vee c_1)$ and $(\neg v \vee c_2)$ implies its resolvent $(c_1 \vee c_2)$.

$$(v \vee c_1) \wedge (\neg v \vee c_2) \Rightarrow (c_1 \vee c_2)$$

Theorem 2.1 For an unsatisfiable SAT instance, there exists a resolution refutation steps leads to an empty clause.

Proof. Since every unsatisfiable SAT instance must imply a contradiction that is an empty clause. Hence, theorem 2.1 can be proved directly by Proposition 2.3. There must exists some resolution steps leads to an empty clause. □

Often, only a subset of clauses of an unsatisfiable SAT instance participate in the resolution steps, which lead to an empty clause. This subset of clauses of a unsatisfiable SAT instance is called *unsatisfiable core*, the detailed definition of unsatisfiable core will be described later.

Definition 2.5 A *refutation proof* of an unsatisfiable formula φ is a directed acyclic graph (V_φ, E_φ). Where V_φ is a set of clauses, and the SAT instance of formula φ is a clause set C, such that

- for each vertex $k \in V_\varphi$, either
  - $k \in C$, and k is a root, or
  - k has exactly two predecessors, k_1 and k_2, and k is the resolvent of k_1 and k_2, and
- there exists an empty clause be the unique leaf of (V_φ, E_φ).

Theorem 2.2 If there is a refutation proof of unsatisfiability for clause set C, C must be unsatisfiable.

Definition 2.6 For an unsatisfiable SAT instance of formula φ, the *unsatisfiable core* of the formula is the subset of clauses whose conjunction is still unsatisfiable. The unsatisfiable core is called *minimal* if all the subsets of it are satisfiable. The unsatisfiable core is called *minimum* if it contains the smallest number of the original clauses to be still unsatisfiable.

The intuition behind the unsatisfiable core is that, these subset of the original clauses are sufficient causing whole CNF formula to be constant $False$. The assignment of variables not in the unsatisfiable core will not effect the unsatisfiability of the formula. Note that the unsatisfiable core of a unsatisfiable formula is not unique, there are one or many different unsatisfiable cores of a particular unsatisfiable formula. In the point of view of a resolution refutation proof, the simplest unsatisfiable core is the subset of root clauses that has a path to the unique empty leaf clause.

Figure 2.3 illustrates the resolution refutation proof, and one of the simplest unsatisfiable core of the unsatisfiable formula $\varphi = (\neg c)(\neg b{+}a)(c{+}\neg a)(b)(d{+}e)(\neg d)$.

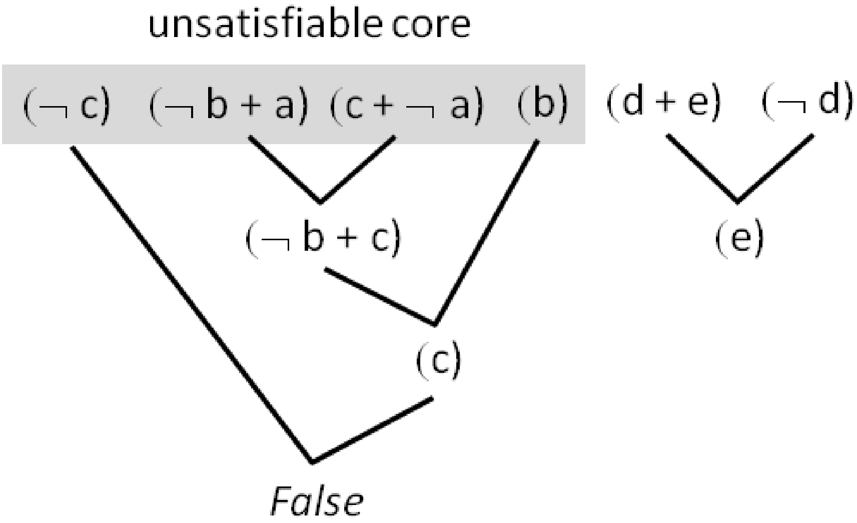


Figure 2.3: Refutation proof and the unsatisfiable core

2.3.5 Craig interpolation

Theorem 2.3 (Craig Interpolation Theorem) [6]

Given two Boolean formulas φ_A and φ_B, with $\varphi_A \wedge \varphi_B$ unsatisfiable, then there exists a Boolean formula ψ_A satisfy the 3 properties that

- ψ_A referring only to the common variables of φ_A and φ_B
- $\varphi_A \Rightarrow \psi_A$
- $\psi_A \wedge \varphi_B$ is unsatisfiable.

The Boolean formula ψ_A is referred to as the *interpolant* of φ_A with respect to φ_B. Some modern SAT solvers, e.g., MiniSat [10], are capable of constructing an interpolant from an unsatisfiable SAT instance. Figure 2.4 illustrates the solution space of φ_A, φ_B and the interpolant ψ_A. Note that the smallest interpolant is φ_A itself and the largest interpolant is $\neg\varphi_B$.

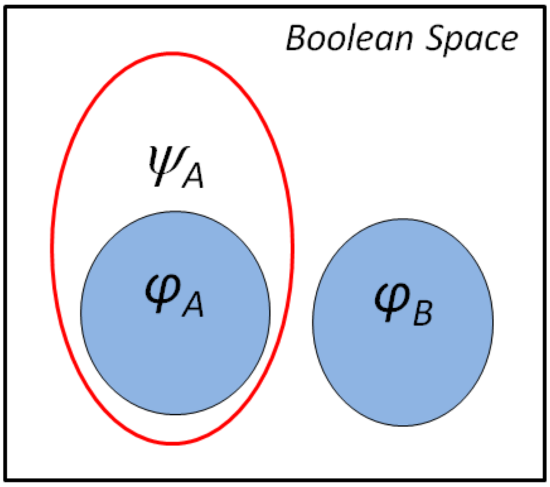


Figure 2.4: Craig interpolation

There are many researches [20, 21] show that we can build an interpolant from the resolution refutation proof of an unsatisfiable SAT instance in linear time to the proof size itself. In this thesis, we use the method described in [20] as our underlying way to get an interpolant.

Suppose we are given a pair of clause sets (A, B) derived from the formula φ and a refutation proof of unsatisfiability (V_φ, E_φ) of $A \cup B$. For the clause sets (A, B), a variable is said to be *global* if it appears in both A and B, and *local* to A if it appears only in A. Other variables are considered *irrelevant* in the interpolant construction procedure. Similarly, a literal is global or local depending on the variable it contains. Moreover, given any clause c, we use $g(c)$ to denote the disjunction of the global literals in c.

The linear time algorithm for interpolant construction mentioned in [20] is in the following rules.

- Let $f(c)$ be a boolean formula for all vertices $c \in V_\varphi$
- if c is a root, then
  - if $c \in A$ then $f(c) = g(c)$
  - if $c \in B$ then $f(c)$ is the constant $True$
- if c is the intermediate vertex, let c_1 and c_2 be the predecessors of c, and v be their pivot variable.
  - if v is local to A, then $f(c) = f(c_1) \vee f(c_1)$
  - else, $f(c) = f(c_1) \wedge f(c_1)$

Since every unsatisfiable SAT instance implies the constant $False$, so the last step of the resolution refutation proof resolve a constant $False$. Hence, $f(False)$ is a boolean function constructed from the above rules. Also, it is the interpolant of the unsatisfiable clause sets $A \cup B$ and the corresponding refutation proof (V_φ, E_φ).

Figure 2.5 illustrates the resolution refutation proof and detailed construction step of an unsatisfiable formula $\varphi = (\neg c)(\neg b + a)(c + \neg a)(b)$.

Note that the interpolant construction can be done in $O(N + L)$ through the topology of the refutation proof, where N is the number of vertices in V_φ and L is the total literal number in the resolution refutation proof (V_φ, E_φ). We need the complexity of L since each time a resolvent is going to be generated, we have to scan all of the literals in the predecessor clauses to find the pivot variable.

$$\begin{aligned} A &= (\neg b + a)(c + \neg a) \\ B &= (b)(\neg c) \\ local &= a \\ global &= b, c \end{aligned}$$

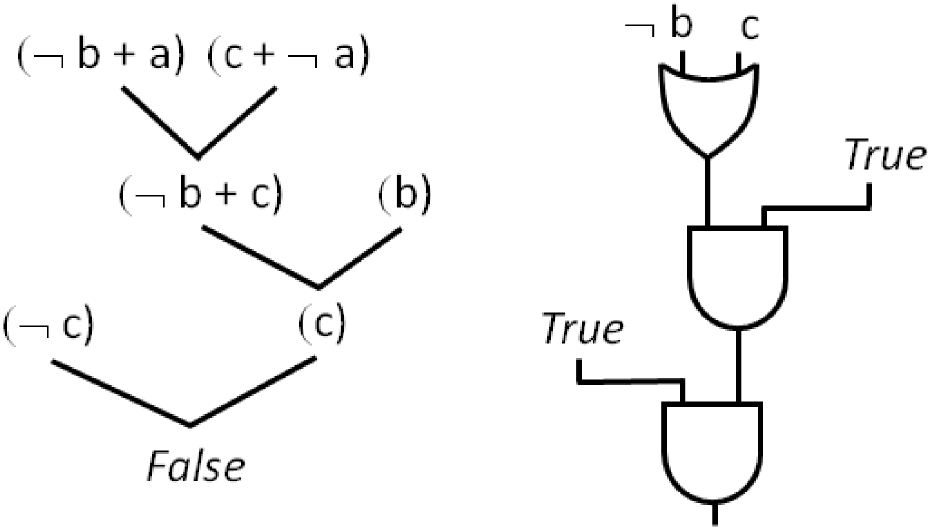


Figure 2.5: Interpolant construction of an unsatisfiable formula $\varphi = (\neg c)(\neg b + a)(c + \neg a)(b)$

Chapter 3

Main Algorithms

We show that Ashenhurst decomposition of a set of Boolean functions $\{f_1, \ldots, f_m\}$, or in some cases we called it m-output Ashenhurst decomposition, can be achieved by SAT solving, Craig interpolation, and functional dependency. Whenever a non-trivial decomposition exists, we derive functions h_i and g automatically for $f_i(X) = h_i(X_H, X_C, g(X_G, X_C))$ under the corresponding variable partition $X = \{X_H|X_G|X_C\}$.

3.1 Single-Output Ashenhurst Decomposition

We first consider single-output Ashenhurst decomposition for a Boolean function $f(X) = h(X_H, X_C, g(X_G, X_C))$.

3.1.1 Decomposition with Known Variable Partition

Proposition 2.1 in the context of Ashenhurst decomposition of a single function can be formulated as satisfiability solving as follows.

Proposition 3.1 A completely specified Boolean function $f(X)$ can be expressed as $h(X_H, X_C, g(X_G, X_C))$ for some functions g and h if and only if the Boolean formula

$$\begin{aligned} &(f(X_H^1, X_G^1, X_C) \not\equiv f(X_H^1, X_G^2, X_C)) \wedge \\ &(f(X_H^2, X_G^2, X_C) \not\equiv f(X_H^2, X_G^3, X_C)) \wedge \\ &(f(X_H^3, X_G^3, X_C) \not\equiv f(X_H^3, X_G^1, X_C)) \end{aligned} \tag{3.1}$$

is unsatisfiable, where a superscript i in Y^i denotes the i^{th} copy of the instantiation of variables Y.

Observe that Formula (3.1) is satisfiable if and only if there exists more than two distinct column patterns in some matrix of the decomposition chart. Hence the unsatisfiability means there are at most two different kind of column patterns in every matrix of the decomposition chart. Since the g function has only one output bit, so the unsatisfiability of Formula (3.1) is exactly the existence condition of Ashenhurst decomposition.

The intuition why we only allowed at most two different kind of column patterns in a matrix can be realized using the decomposition chart. Function g can be treated as a mapping from a input assignment $a \in [\![X_G, X_C]\!]$ to the Boolean value 0 or 1. Since every column index of a matrix in the decomposition chart is exactly a input assignment $a \in [\![X_G, X_C]\!]$, if there are more than two different kind of column patterns, it cannot be mapped using binary value 0 and 1.

Note that, unlike BDD-based counterparts, the above SAT-based formulation of Ashenhurst decomposition naturally extends to non-disjoint decomposition. It is because the unsatisfiability checking of Formula (3.1) essentially tries to assert that under every valuation of variables X_C the corresponding matrix of the decomposition chart has at most two column patterns. In contrast, BDD-based methods have to check the decomposability under every valuation of X_C separately.

Now we have known that the decomposability of function f can be checked through SAT solving of Formula (3.1), the derivations of functions g and h can be realized through Craig interpolation and functional dependency, respectively, as shown below.

To derive function g, we partition Formula (3.1) into two sub-formulas

$$\begin{aligned} \varphi_A &= f(X_H^1, X_G^1, X_C) \not\equiv f(X_H^1, X_G^2, X_C), \text{ and} \quad (3.2) \\ \varphi_B &= (f(X_H^2, X_G^2, X_C) \not\equiv f(X_H^2, X_G^3, X_C)) \wedge \\ &\quad (f(X_H^3, X_G^3, X_C) \not\equiv f(X_H^3, X_G^1, X_C)). \quad (3.3) \end{aligned}$$

Figure 3.1 shows the corresponding circuit representation of Formulas (3.2) and (3.3). The circuit representation can be converted into a CNF formula in linear time [27], and thus can be checked for satisfiability.

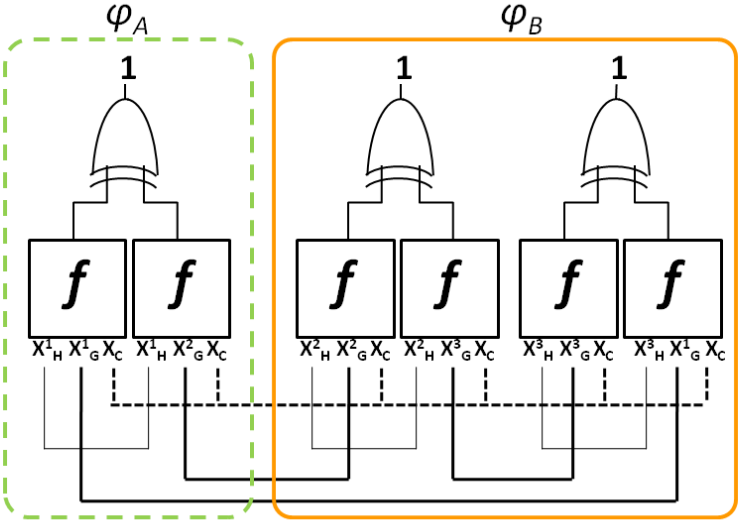


Figure 3.1: Circuit representation of Formulas (3.2) and (3.3)

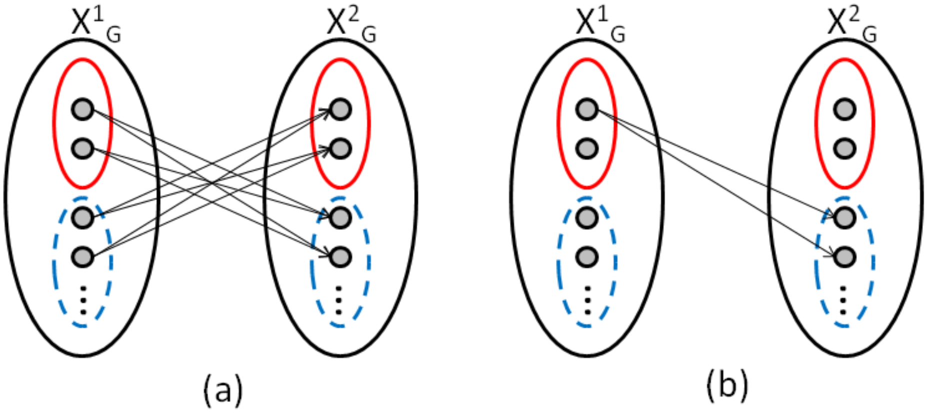


Figure 3.2: (a) Relation characterized by $\psi_A(X_G^1, X_G^2, c)$ for some $c \in [\![X_C]\!]$ (b) Relation after cofactoring $\psi_A(X_G^1 = a, X_G^2, c)$ with respect to some $a \in [\![X_G^1]\!]$

Lemma 3.1 For function $f(X)$ decomposable under Ashenhurst decomposition with variable partition $X = \{X_H|X_G|X_C\}$, the interpolant ψ_A with respect to φ_A of Formula (3.2) and φ_B of Formula (3.3) corresponds to a characteristic function such that,

1. for φ_A satisfiable under some $c \in [\![X_C]\!]$, there exist $b_1 \in [\![X_G^1]\!]$ and $b_2 \in [\![X_G^2]\!]$ such that $\psi_A(b_1, b_2, c) = 1$ if and only if the column patterns indexed by b_1 and b_2 in the matrix of c of the decomposition chart of f are different;

2. for φ_A unsatisfiable under some $c \in [\![X_C]\!]$, there is only one column pattern in the matrix of c of the decomposition chart of f;

3. for φ_A unsatisfiable under all $c \in [\![X_C]\!]$, or in other word, unsatisfiable φ_A, variables X_G are not the support variables of f and thus $\{X_H|X_G|X_C\}$ is a trivial variable partition for f.

Proof. For f decomposable, $\varphi_A \wedge \varphi_B$ is unsatisfiable by Proposition 3.1. Moreover from Theorem 2.3, we know the interpolant ψ_A of the unsatisfiability proof is a function that refers only to the common variables, $X_G^1 \cup X_G^2 \cup X_C$, of φ_A and φ_B.

We analyze what the characteristic function ψ_A means for φ_A satisfiable under some $c \in [\![X_C]\!]$. Let $a_i \in [\![X_H^i]\!]$, $b_i \in [\![X_G^i]\!]$, for $i = 1, 2, 3$, in the following discussion. Let $R_c \subseteq [\![X_G^1]\!] \times [\![X_G^2]\!]$ be the relation that $\psi_A(X_G^1, X_G^2, X_C = c)$ characterizes. On the one hand, since $\varphi_A \Rightarrow \psi_A$, we know that ψ_A is an over-approximation of the solution space of φ_A projected to the common variables $X_G^1 \cup X_G^2 \cup X_C$. So R_c must contain the set

$$\{(b_1, b_2) \mid f(a_1, b_1, c) \neq f(a_1, b_2, c) \text{ for some } a_1\}.$$

That is, a pair (b_1, b_2) must be in R_c if the columns indexed by b_1 and b_2 of the matrix of c in the decomposition chart of f are of different patterns.

On the other hand, since $\psi_A \wedge \varphi_B$ is unsatisfiable, the solution space of ψ_A is disjoint from that of φ_B projected to the common variables. Suppose that valuations a_2, a_3, b_1, b_2, b_3 satisfy φ_B under c, that is, $f(a_2, b_2, c) \neq f(a_2, b_3, c)$ and $f(a_3, b_3, c) \neq f(a_3, b_1, c)$. Then the columns indexed by b_2 and b_3 of the matrix of c in the decomposition chart of f are of different column patterns, and the column indexed by b_3 and b_1 has the same property. Since we know that f is decomposable under Ashenhurst decomposition, that means there are at most two different kind of column patterns for every matrix under c. So the columns indexed by b_1 and b_2 of the matrix of c must be the same column pattern. For $\psi_A \wedge \varphi_B$ unsatisfiable, it represents that (b_1, b_2) cannot be in R_c if the columns indexed by b_1 and b_2 are of the same pattern. We can now conclude that the interpolant $\psi_A(X_G^1, X_G^2, c)$ characterize all different kind of column patterns indexed by (b_1, b_2), where $b_1 \in [\![X_G^1]\!]$ and $b_2 \in [\![X_G^2]\!]$ for some matrix of c.

Figure 3.2(a) illustrates the relation that $\psi_A(X_G^1, X_G^2, c)$ characterizes for some $c \in [\![X_C]\!]$. The left and right sets of gray dots denote the elements of $[\![X_G^1]\!]$ and $[\![X_G^2]\!]$, respectively. For function f to be decomposable, there are at most two equivalence classes for the elements of $[\![X_G^i]\!]$ for $i = 1, 2$. In Figure 3.2(a), the two clusters of elements in $[\![X_G^i]\!]$ signify two equivalence classes of column patterns indexed by $[\![X_G^i]\!]$. An edge (b_1, b_2) between $b_1 \in [\![X_G^1]\!]$ and $b_2 \in [\![X_G^2]\!]$ represents that b_1 is not in the same equivalence class as b_2, they have different kind of column patterns. In this case, $\psi_A(b_1, b_2, c) = 1$. Since all of the different column pattern pairs (b_1, b_2) in the matrix of c will be characterized by the interpolant $\psi_A(X_G^1, X_G^2, c)$. So every element in one equivalence class of $[\![X_G^1]\!]$ is connected to every element in the other equivalence class of $[\![X_G^2]\!]$ as Figure 3.2(a) shows.

According to the above analysis, we conclude that, for φ_A satisfiable under $c \in [\![X_C]\!]$, relation R_c characterizes exactly the set of index pairs (b_1, b_2) whose corresponding column patterns are different.

For φ_A unsatisfiable under some $c \in [\![X_C]\!]$, it represents that there are no any different kind of column patterns in the matrix of c, so that there is only one column pattern in the matrix of c of the decomposition chart of f. Hence, under $X_C = c$, the valuation of f is independent of the truth assignments of X_G. No matter what different truth assignment of X_G is, the value of f under the same truth assignment of X_H is of the same. (In this case, ψ_A under $X_C = c$ can be an arbitrary function due to the solution space of φ_A is empty.) If φ_A is unsatisfiable under every $c \in [\![X_C]\!]$, then the valuation of function f does not depend on X_G, that is, X_G are not the support variables of f.

Since the analysis holds for every $c \in [\![X_C]\!]$, the lemma follows. □

Next, we show how to derive function g from the interpolant ψ_A.

Lemma 3.2 For an arbitrary $a \in [\![X_G^1]\!]$, the cofactored interpolant $\psi_A(X_G^1 = a, X_G^2, X_C)$ is a legal implementation of function $g(X_G^2, X_C)$.

Proof. By Lemma 3.1, the interpolant $\psi_A(X_G^1, X_G^2, X_C)$ characterizes all different column pattern pairs in every matrix who has exactly two different kind of column patterns. Let $\gamma_a(X_G^2, X_C) = \psi_A(X_G^1 = a, X_G^2, X_C)$ for some arbitrary $a \in [\![X_G^1]\!]$. Then, under every such valuation $c \in [\![X_C]\!]$, $\gamma_a(X_G^2, c)$ characterizes the set of indices whose corresponding column patterns are different from the column pattern indexed by a. In other word, $\gamma_a(X_G^2, c)$ characteristic either of the two equivalence classes of column patterns in the matrix of c.

Taking Figure 3.2(b) as an example, assume that a is the topmost element in $[\![X_G^1]\!]$. After cofactoring, all the edges in Figure 3.2(a) will disappear except for the edges connecting a with the elements in the other equivalence class of $[\![X_G^2]\!]$.

On the other hand, for $c \in [\![X_C]\!]$ whose corresponding matrix contains only one kind of column pattern. Since we know that $X_C = c$ is a don't-care condition for function g, and thus $\gamma_a(X_G^2, c)$ can be arbitrary.

From the above analysis, we can conclude that $\gamma_a(X_G^2, X_C)$ characterizes either the onset or the offset of function $g(X_G^2, X_C)$. So $\gamma_a(X_G^2, X_C)$ can be treated as the function $g(X_G^2, X_C)$ or negation of the function $g(X_G^2, X_C)$. After renaming X_G^2 to X_G, we get the desired $g(X_G, X_C)$. □

So far we have successfully derived function g by interpolation and cofactor operation. Next we need to compute function h. The problem can be formulated as computing functional dependency as follows. Let $f(X)$ be our target function; let function $g(X_G, X_C)$ we just derived and identity functions $\imath_x(x) = x$, one for each variable $x \in X_H \cup X_C$, be our base functions. So the computed dependency function is exactly our desired h. Since functional dependency can be formulated using SAT solving and interpolation [16], it well fits in our SAT-based computation framework.

Remark: For disjoint decomposition, i.e., $X_C = \emptyset$. Rather than using functional dependency, we can derive the function h in a simple way.

Given two functions $f(X)$ and $g(X_G)$ with variable partition $X = \{X_H|X_G\}$, our goal is to find a function $h(X_H, x_g)$ such that $f(X) = h(X_H, g(X_G))$, where x_g is the output variable of function $g(X_G)$. Let $a, b \in [\![X_G]\!]$ with $g(a) = 0$ and $g(b) = 1$. Then by Shannon expansion

$$h(X_H, x_g) = (\neg x_g \wedge h_{\neg x_g}(X_H)) \vee (x_g \wedge h_{x_g}(X_H)),$$

where $h_{\neg x_g}(X_H) = f(X_H, X_G = a)$ and $h_{x_g}(X_H) = f(X_H, X_G = b)$. The derivation of the offset and onset minterms of g is easy because we can pick an arbitrary minterm c in $[\![X_G]\!]$ and see if $g(c)$ equals 0 or 1. We then perform SAT solving on either $g(X_G)$ or $\neg g(X_G)$ depending on the value $g(c)$ to derive another necessary minterm.

The above derivation of function h, however, does not scale well for decomposition with large $|X_C|$ because we may need to compute $h(X_H, X_C = c, x_g)$, one for every valuation $c \in [\![X_C]\!]$. There are $2^{|X_C|}$ cases to analyze. So when common variables exist, functional dependency may be a better way to compute function h.

The correctness of the so-derived Ashenhurst decomposition follows from Lemma 3.2 and Proposition 2.1, as the following theorem states.

Theorem 3.1 Given a function f decomposable under Ashenhurst decomposition with variable partition $X = \{X_H | X_G | X_C\}$, then $f(X) = h(X_H, X_C, g(X_G, X_C))$ for functions g and h obtained by the above derivation.

Algorithm 1 Derive g and h with a given variable partition

```
Input: $f$ and a given variable partition $X = \{X_H | X_G | X_C\}$
Output: $g$ and $h$
1: $F \Leftarrow CircuitInstantiation(f, X_H, X_G, X_C)$
2: $(\varphi_A, \varphi_B) \Leftarrow CircuitPartition(F)$
3: $P(X_G^1, X_G^2, X_C) \Leftarrow Interpolation(\varphi_A, \varphi_B)$
4: $g(X_G, X_C) \Leftarrow Cofactor(P, a \in [\![X_G^1]\!])$
5: $h \Leftarrow FunctionalDependency(f, g)$
6: return $(g, h)$
```

Algorithm 1 summarizes the algorithms we used to derive the function g and h. Line 1-2 duplicate the original function f into 6 copies, and then partition it into two part, just as Figure 3.1 shows. Line 3-4 utilize interpolation technique to derived a relation contain the information of pairs of distinct column patterns. Then using cofactor to get the g function we want. In line 5, we formulate the problem as computing functional dependency to get the h function.

3.1.2 Decomposition with Unknown Variable Partition

The construction in the previous subsection assumes that a variable partition $X = \{X_H|X_G|X_C\}$ is given. In this subsection, we will show how to automate the variable partition process within the decomposition process of function f. A similar approach was used in [17] for bi-decomposition of Boolean functions.

We introduce two control variables α_{x_i} and β_{x_i} for each variable $x_i \in X$. In addition we instantiate the original input variables X into six copies X^1, X^2, X^3, X^4, X^5, and X^6. Let

$$\begin{aligned}
\varphi_A &= (f(X^1) \not\equiv f(X^2)) \wedge \bigwedge_i ((x_i^1 \equiv x_i^2) \vee \beta_{x_i}) \text{ and} && (3.4)\\
\varphi_B &= (f(X^3) \not\equiv f(X^4)) \wedge (f(X^5) \not\equiv f(X^6)) \wedge \\
&\quad \bigwedge_i (((x_i^2 \equiv x_i^3) \wedge (x_i^4 \equiv x_i^5) \wedge (x_i^6 \equiv x_i^1)) \vee \alpha_{x_i}) \wedge \\
&\quad \bigwedge_i (((x_i^3 \equiv x_i^4) \wedge (x_i^5 \equiv x_i^6)) \vee \beta_{x_i}), && (3.5)
\end{aligned}$$

where $x_i^j \in X^j$ for $j = 1, \ldots, 6$ are the instantiated versions of $x_i \in X$. Why α_{x_i} and β_{x_i} are called control variables is because each of the control variable can enable of disable the corresponding clause. For example, a clause $(a + b)$ associates with a control variable α results in a new clause $(a + b + \alpha)$. When $\alpha = 1$, the clause is immediate evaluated as constant $True$, then it become a redundant clause in the CNF formulas. In this case, we may say this clause is *disabled.* In other hand, if $\alpha = 0$ the clause is just as the original one $(a + b)$. In this case, the clause is said to be *enabled* by the control variable α. Under this point of view, $(\alpha_{x_i}, \beta_{x_i}) = (0, 0)$, $(0, 1)$, $(1, 0)$, and $(1, 1)$ indicate that $x_i \in X_C$, $x_i \in X_G$, $x_i \in X_H$, and x_i can be in either of X_G and X_H, respectively.

In SAT solving the conjunction of Formulas (3.4) and (3.5), we make *unit assumptions* [10] on the control variables. Unit assumption can be made on a list of literals so that the solution space of SAT solving is restricted to these pre-specified space. Generally speaking, when SAT solving, for a variable appears in the assumption list, not both valuation 0 and 1 are tried. It is forced to be the assumption value as specified in the assumption list. Once the SAT instance is unsatisfiable under the assumption, SAT solver will return a final conflict clause contain variables only in the assumption list. This final conflict clause indicates that these part of assumption values are sufficient to make the CNF formulas to be unsatisfied.

Similar to [17] but with a subtle difference, we introduce the following *seed variable partition* to avoid trivial variable partition, which is $X_C \cup X_G$ or $X_C \cup X_H$ equals X, and to avoid $|X_G| = 1$. For the unit assumption, initially we specify three distinct variables x_j, x_k, and x_l with x_j in X_H partition and x_k, x_l in X_G partition, and specify all other variables in X_C partition. That is, we have $(\alpha_{x_j}, \beta_{x_j}) = (1, 0)$, $(\alpha_{x_k}, \beta_{x_k}) = (0, 1)$, $(\alpha_{x_l}, \beta_{x_l}) = (0, 1)$, and $(\alpha_{x_i}, \beta_{x_i}) = (0, 0)$ for $i \neq j, k, l$.

Lemma 3.3 For an unsatisfiable conjunction of Formulas (3.4) and (3.5) under a seed variable partition, the final conflict clause consists of only the control variables, which indicates a valid non-trivial variable partition.

Proof. The values of control variables are specified in the unit assumption as if they are in the first decision level. In solving an unsatisfiable instance, both 0 and 1 valuations of any other variable must have been tried and failed, and only the control variables are not valuated in both cases. Because unit assumption causes the unsatisfiability, the final learned clause indicates the conflict decisions made in the first decision level.

To see why the final learned clause corresponds to a valid variable partition, notice that every literal in the conflict clause is of positive phase because the conflict arises from a subset of the control variables set to 0. The returned conflict clause reveals that setting to 0 the control variables present in the conflict clause is sufficient making the whole CNF formula unsatisfiable. Hence setting the control variables who do not appear in the conflict clause to 1 cannot affect the unsatisfiability. Hence the final conflict clause indicates a variable partition X_H, X_G, X_C on X. For example, the conflict clause $(\alpha_{x_1} + \beta_{x_1} + \alpha_{x_2} + \beta_{x_3})$ indicates that the subset $\alpha_{x_1} = 0$, $\beta_{x_1} = 0$, $\alpha_{x_2} = 0$, and $\beta_{x_3} = 0$ of the original unit assumption sufficiently results in the unsatisfiability. Setting other control variables absent from the conflict clause cannot effect the unsatisfiability, and also, setting these control variables to be 1 potentially move the corresponding input variable from X_C to X_G or X_H, or even more, from X_G or X_H to a more freedom state which can be either in X_G or X_H. Hence, it in turn suggests that $x_1 \in X_C$, $x_2 \in X_G$, and $x_3 \in X_H$.

In addition, the corresponding variable partition is non-trivial since $|X_H| \geq 1$ and $|X_G| \geq 2$ due to the seed variable partition. □

If the conjunction of Formulas (3.4) and (3.5) is unsatisfiable under a seed variable partition, then the corresponding decomposition, where the variable partition is indicated by the final conflict clause, is successful. Otherwise, it indicates that the function f is not decomposable under this seed variable partition, we should try another seed variable partition. For a given function $f(X)$ with $|X| = n$, the existence of non-trivial Ashenhurst decomposition can be checked with at most $3 \cdot C_3^n$ different seed partitions. Note that the constant 3 here represents once the three different variables are chosen, any variable of the three can be in X_H partition.

Rather than just looking for a valid variable partition, we may further target one that is more balanced (i.e., $|X_H|$ and $|X_G|$ are of similar sizes) and closer to disjoint (i.e., $|X_C|$ is small) by enumerating different seed variable partitions. As SAT solvers usually refer to a small unsatisfiable core, the returned variable partition is desirable because $|X_C|$ tends to be small. Even if a returned unsatisfiable core is unnecessarily large, the corresponding variable partition can be further refined by modifying the unit assumption to reduce the unsatisfiable core and reduce $|X_C|$ as well. The process can be iterated until the unsatisfiable core is minimal. We introduce an UNSAT core refinement process to further reduce the number of variables in X_C, the detailed description can be found in subsection 3.5.1.

After automatic variable partitioning, functions g and h can be derived through a construction similar to the decomposition problem with a given variable partition. The correctness of the overall construction can be asserted.

Theorem 3.2 For a function f decomposable under Ashenhurst decomposition, we have $f(X) = h(X_H, X_C, g(X_G, X_C))$ for functions g and h, and a non-trivial variable partition $X = \{X_H | X_G | X_C\}$ derived from the above construction.

Algorithm 2 Derive g and h without a given variable partition

```
Input: $f$
Output: $g$ and $h$
1: $(X_H, X_G, X_C) \Leftarrow FindAGoodPartition(f)$
2: $F \Leftarrow CircuitInstantiation(f, X_H, X_G, X_C)$
3: $(\varphi_A, \varphi_B) \Leftarrow CircuitPartition(F)$
4: $P(X_G^1, X_G^2, X_C) \Leftarrow Interpolation(\varphi_A, \varphi_B)$
5: $g(X_G, X_C) \Leftarrow Cofactor(P, a \in [\![X_G^1]\!])$
6: $h \Leftarrow FunctionalDependency(f, g)$
7: return $(g, h)$
```

Algorithm 2 summarizes the algorithms we used to derive the function g and h without a given variable partition. Line 1 is the core procedure to derive a valid and good partition result, the detailed algorithm of this step is described in Algorithm 3.

In Algorithm 3, the input is the f function itself, and output is a valid variable

Algorithm 3 *FindAGoodPartition*(f)

```
Input: $f$
Output: $VPBest$
$failcount \Leftarrow 0$
$TimeStart \Leftarrow Time()$
$VPBest \Leftarrow AllVarInCommon()$
while $(Time() - TimeStart) < 60$ do
  $SP \Leftarrow GetASeedPar()$
  $(VPCur, partitionable) \Leftarrow GetAValidPar(f, SP)$
  if $partitionable = true$ then
    if $disjointness(VPCur) < disjointness(VPBest)$ then
      $VPBest \Leftarrow VPCur$
      $failcount \Leftarrow 0$
    else if $disjointness(VPCur) = disjointness(VPBest)$ then
      if $balancedness(VPCur) < balancedness(VPBest)$ then
        $VPBest \Leftarrow VPCur$
        $failcount \Leftarrow 0$
  if $VPBest$ doesn't renew then
    $failcount + +$
  if $failcount \geq 1500$ then
    break
$VPBest \Leftarrow UNSATCoreRefinement(VPBest)$
```

partition $X = \{X_H|X_G|X_C\}$. Line 1-3 set some initial values, "VPBest" denotes the best variable partition we derived. In the beginning, the best variable partition is assigned to be with all variables in the common variable X_C. There are two stopping criterions in this algorithm. One is the time limit, we only allow this partition procedure be executed no more than 60 seconds, as line 4 shows. Another stopping criterion is in line 17-18. If both disjointness and balancedness[1] cannot be improved in consecutive 1500 seed partition trials, the partition procedure will be terminated.

Line 5-6 enumerate different seed partitions, and try to find a valid partition under this pre-specify seed partition by using the *unit assumption* technique. If the seed partition is partitionable, line 7-14 check the the improvement of disjointness and balancedness, if the current partition is a better result, the best partition will be replaced. Note that improve disjointness is more preferable than balancedness in our algorithm. Finally, there is an UNSAT core refinement procedure in line 19 to get a more disjointness and balancedness partition.

3.2 Multiple-Output Ashenhurst Decomposition

So far we considered single-output Ashenhurst decomposition for a single function f. We next show that the algorithm is extendable to multiple-output Ashenhurst

[1] The detailed definition of disjointness and balancedness can be found in Chapter 4.

decomposition for a set $\{f_1, \ldots, f_m\}$ of functions.

Proposition 2.1 in the context of Ashenhurst decomposition of a set of functions can be formulated as satisfiability solving as follows.

Proposition 3.2 A set $\{f_1(X), \ldots, f_m(X)\}$ of completely specified Boolean functions can be expressed as

$$f_i(X) = h_i(X_H, X_C, g(X_G, X_C))$$

for some functions h_i and g with $i = 1, \ldots, m$ if and only if the Boolean formula

$$\begin{aligned}
&(\bigvee_i f_i(X_H^1, X_G^1, X_C) \not\equiv f_i(X_H^1, X_G^2, X_C)) \wedge \\
&(\bigvee_i f_i(X_H^2, X_G^2, X_C) \not\equiv f_i(X_H^2, X_G^3, X_C)) \wedge \\
&(\bigvee_i f_i(X_H^3, X_G^3, X_C) \not\equiv f_i(X_H^3, X_G^1, H_C)) \qquad (3.6)
\end{aligned}$$

is unsatisfiable.

We assume that every f_i shares the same g, so in every matrix of the decomposition chart of a set of f_i functions, it is allowed to have at most two different kind of column patterns. Note that every element in the decomposition chart has m output assignments $(v_1, \ldots, v_m)$ for every output function f_i. Formula 3.6 checks whether there are more than two different kind of column patterns in the decomposition chart of a set $\{f_1, \ldots, f_m\}$ of functions.

Since the derivation of functions g and h_i, and automatic variable partitioning are essentially the same as the single-output case, we omit the detailed explanations.

3.2.1 Shared Variable Partition

In Formula 3.6, we assume that every f_i shares the same variable partition $X = \{X_H|X_G|X_C\}$. Actually, take $i = 2$, which is two-output Ashenhurst decomposition problem as an example, there can be 7 kinds of variable partitions. More specifically, a variable is said to be in X_g if the variable is used only by g, in X_{h1} if the variable is used only by h_1, in X_{h2} if the variable is used only by h_2, in $X_{h1,h2}$ if the variable is shared by h_1 and h_2, in $X_{g,h1}$ if the variable is share by g and h_1, in $X_{g,h2}$ if the variable is share by g and h_2, and in $X_{g,h1,h2}$ if the variable is share by g, h_1, and h_2.

In our formulation, for a multi-output decomposition problem, the variables are partitioned into 3 groups X_H, X_G, and X_C. However this grouping does not force a variable v which belongs to one of the 3 groups to be shared by all functions f_i. Since in our derivation we use functional dependency to construct the h_i functions, for a variable v is assigned to the X_H partition, the corresponding identity function of this variable become one of the candidate base function of functional dependency problem. However the constructed dependency function may not depend on all base functions. Once the dependency function h_1 depends on variable v but h_2 do not, the variable v is now in X_{h1} partition. In other word, we do not have to partition the variables into 7 groups before deriving the sub-functions, the variable partition $X = \{X_H|X_G|X_C\}$ we used in the algorithm is sufficient to cover 7 partition groups. X_{h1}, X_{h2}, and $X_{h1,h2}$ can be covered by the partition X_H; $X_{g,h1}$, $X_{g,h2}$, and $X_{g,h1,h2}$

can be covered by partition X_C; and X_g can be covered by partition X_G.

3.3 Beyond Ashenhurst Decomposition

In case we wish to extend our algorithm to general functional decomposition, the following question arises.

Is the above algorithm extendable to general functional decomposition, namely, for $k > 1$?

$$f(X) = h(X_H, X_C, g_1(X_G, X_C), \ldots, g_k(X_G, X_C))$$

The answer is yes, but with prohibitive cost. Taking $k = 2$ for example, we need 20 copies of f to assert the non-existence of five different column patterns for every matrix of a decomposition chart, in contrast to the six for Ashenhurst decomposition shown in Figure 3.1. This number grows in $2^k(2^k + 1)$. Aside from this duplication issue, the derivation of functions g_1, ..., g_k, and h may involve several iterations of finding satisfying assignments and performing cofactoring. The number of iterations varies depending on how the interpolation is computed and can be exponential in k. Therefore we focus mostly on Ashenhurst decomposition.

3.4 Decomposition under Don't-Cares

In Lemma 3.1, we have proved that for φ_A is unsatisfiable under some $c \in [\![X_C]\!]$, there is only one column pattern in the matrix of c of the decomposition chart of f. In this case, the X_G variables will be the don't-care condition under some $c \in [\![X_C]\!]$. We have known that the indices of the column pattern in each matrix can be treated as the input valuations of the g function, and the grouping of these column indices according to their column patterns can be used to distinguish the onset and offset minterms of the g function.

If there is only one column pattern in a matrix with respect to some valuation $c \in [\![X_C]\!]$, we cannot distinguish the onset or offset minterms of the g function in this matrix. Under such $c \in [\![X_C]\!]$, no matter what different valuations of X_G variables are, the output value of the f function are still the same, so this can be treated as a don't-care condition of f function. Since the existence of the don't-care of f function can be used to further simplify the f function, so quickly characterize all $c \in [\![X_C]\!]$ which the corresponding matrix of c has only one column pattern will become the future study.

3.5 Implementation Issues

3.5.1 Minimal UNSAT Core Refinement

Note that, if the final conflict clause returned by the SAT solver does not correspond to a *minimal* unsatisfiable core, for variable x_i being able to be placed in either of X_G and X_H under some variable partition, the valuation of the control variables $(\alpha_{x_i}, \beta_{x_i})$ needs not be $(1, 1)$. Similarly, under non-minimality, variables being able to be placed in X_G or X_H can be misplaced in X_C.

Since a final conflict clause returned by a SAT solver may not reflect a minimal UNSAT core, very likely we can further refine the corresponding variable partition. Suppose that the variable partition is $X = \{X_H|X_G|X_C\}$ before the refinement. We iteratively and greedily try to move a common variable of X_C into X_G or X_H, if available, making the new partition more balanced as well. The iteration continues until no such movement is possible. On the other hand, for a variable x with control variables $(\alpha_x, \beta_x) = (1, 1)$, indicating x can be placed in either of X_H and X_G, we put it in the one such that the final partition is more balanced.

3.5.2 Balanced Partition

When balanced variable partition is needed, randomize the IDs of input variable results in more balanced variable partition. Since SAT solver tend to make decisions in a descending priority order based on variable IDs. So if the input variable IDs have some pattern rules between different copies of f, SAT solving for the variable partition may check conditions of X_G or X_H with a higher priority than other variable group. So appropriately change the order of input variable IDs between different copies of f get more balanced partition result.

3.5.3 Elimination of Equality Constraint Clauses

When dealing with the equality of variables $x_i \in X$ in different copy of f, the current implementation first instantiate variables into needed copies then add equality constraints to claim they have to be the same value. However this procedure may increase the clause number as well as the run-time of SAT solving. Most important of all, current implementation may enlarge the size of interpolant circuit. If the function g obtained from interpolant is large, the derivation of h related to this g will be difficult. So remove the equality constraint by using the same variable ID can reduce the number of clauses when input variable is large.

3.6 A Complete Example

In this section, we provide a complete example for the algorithm, discussing how the aspects described before are taken into account in the execution of the method. Figure 3.3 shows the example circuit we use in the following discussion.

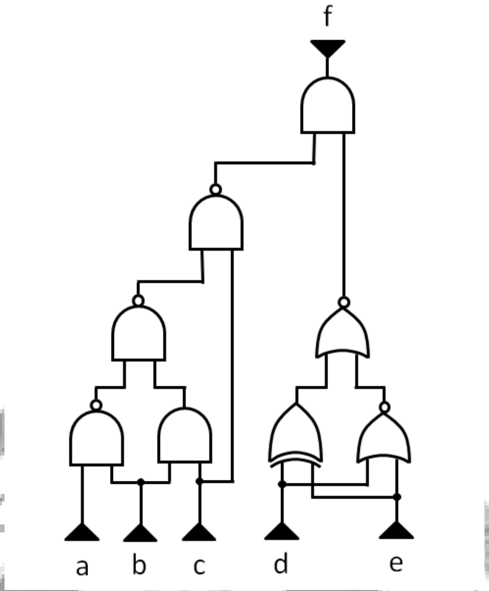
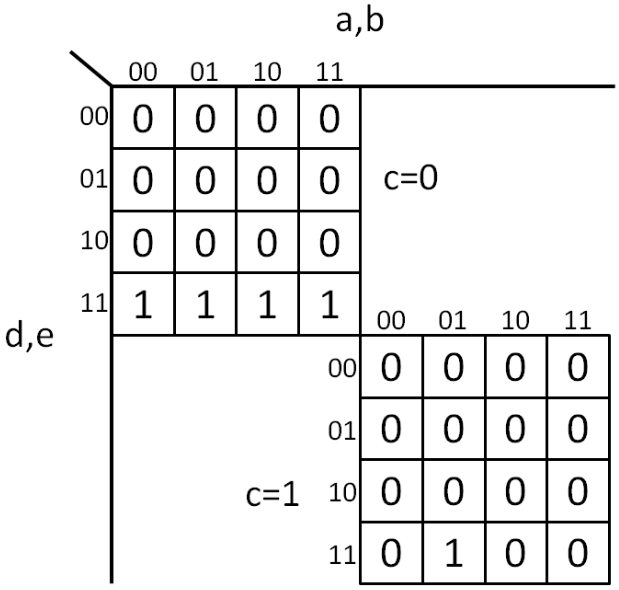


Figure 3.3: Example circuit used in the complete example

The first step is to derive a valid partition. In order to prevent the trivial partition, we force two arbitrary input variables to be strictly in X_G, say, a and b. One other variable to be strictly in X_H, say, d. The rest variables c and e are in X_C partition. These conditions can be specified by setting control variables $(\alpha_a, \beta_a) = (\alpha_b, \beta_b) = (0, 1)$, $(\alpha_d, \beta_d) = (1, 0)$, and $(\alpha_c, \beta_c) = (\alpha_e, \beta_e) = (0, 0)$. When SAT solving the conjunction of Formulas (3.4) and (3.5), if the *unit assumptions* on the control variables results in an unsatisfiable result, that means the partition exists and the returned conflict clause corresponds to a valid variable partition. For example, suppose that the returned conflict clause is $(\alpha_a + \alpha_b + \alpha_c + \beta_c + \beta_d + \beta_e)$. It in turn suggests that $c \in X_C$, $a, b \in X_G$, and $d, e \in X_H$.



c=0

d,e \ a,b	00	01	10	11
00	0	0	0	0
01	0	0	0	0
10	0	0	0	0
11	1	1	1	1

c=1

d,e \ a,b	00	01	10	11
00	0	0	0	0
01	0	0	0	0
10	0	0	0	0
11	0	1	0	0

Figure 3.4: The decomposition chart of the example circuit

The second step is to derive the g function using the variable partition we derived from step one. Since the partition we used is a valid partition, so the conjunction of Formula (3.2) and Formula (3.3) must be unsatisfiable, hence an interpolant exists. Figure 3.5 shows the relation the interpolant characterized. The red and blue groups indicate different column patterns. Note that when $c = 0$, from Figure 3.4 shows, there is only one kind of column pattern in the corresponding matrix. Both Formula (3.2) and Formula (3.3) are unsatisfiable since they cannot distinguish two different column patterns. when $c = 0$, the interpolant could be anything, depends on the proofs of the unsatisfiable SAT instance. When $c = 1$, it characterize all different column patterns just as the decomposition chart in Figure 3.4. Then we cofactor the variable (a^1, b^1) of the interpolant by value $(0, 0)$, hence we derive a legal implementation of g function after changing variables from (a^2, b^2, c^2) to (a, b, c). Note that different value we use to cofactor the variables (a^1, b^1) may result in different g function, it is not unique.

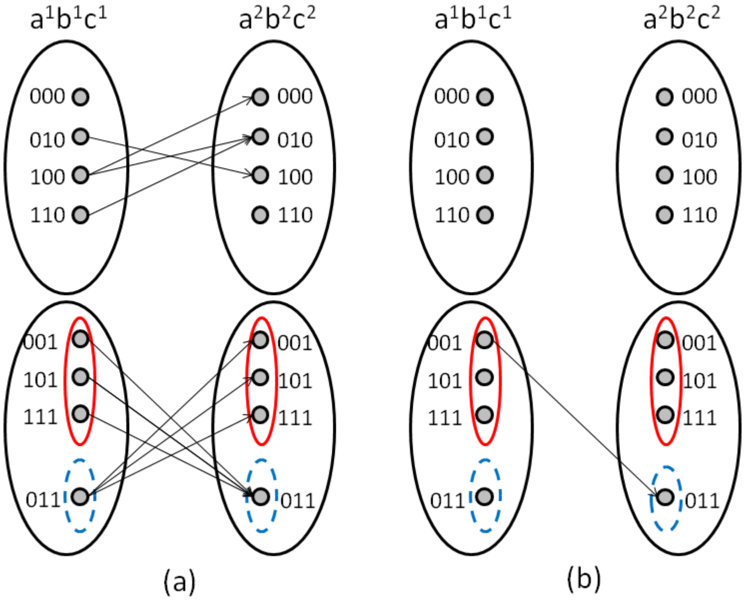


Figure 3.5: (a) Relation of the derived interpolant function, (b) Relation after cofactor by $(a^1, b^1) = (0, 0)$

The third step is to use functional dependency to derive the h function. We use the g function we just derived and identity functions which correspond to each variable in X_H and X_C partition to be the base functions, just as Figure 3.6 (a) shows. In other hand, the original f function in Figure 3.3 be the target function. If we do so, the returned dependency function by the functional dependency procedure is the h function we need. The derived h is shown in Figure 3.6 (b). Note that we have 8 gates in the original example circuit f, but only 5 gates in g and h circuits.

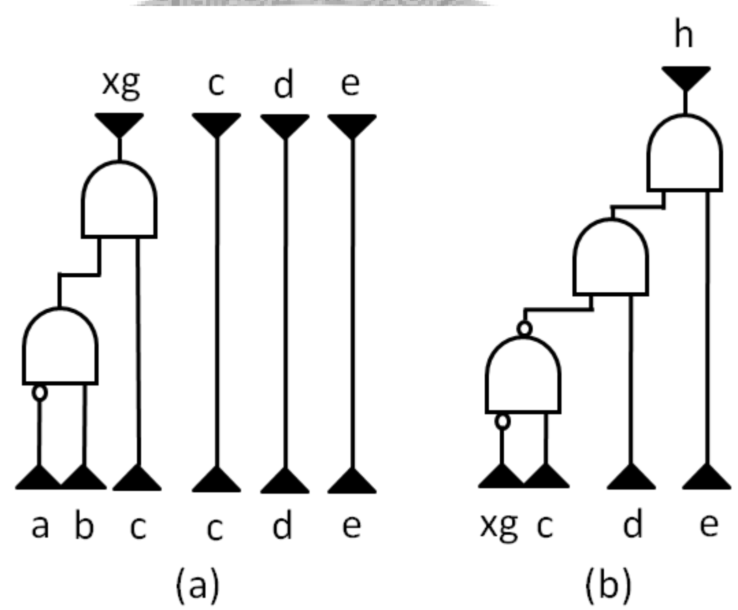


Figure 3.6: (a) The base functions, (b) *h* function derived by functional dependency

Chapter 4

Experimental Results

The proposed approach to Ashenhurst decomposition was implemented in C++ within the ABC package [2] and used MiniSAT [10] as the underlying solver. All the experiments were conducted on a Linux machine with Xeon 3.4GHz CPU and 6Gb RAM.

4.1 Single- and Two-Output Ashenhurst Decomposition

We choose large ISCAS, MCNC and ITC benchmark circuits to evaluate the proposed algorithm. Only large transition and output functions with equal to or more than 50 inputs in the transitive fanin cone were considered. In this section, we eval-

Table 4.1: Single-output Ashenhurst decomposition

circuit	#func	#var	#fail	#SAT_TO	#succ	#var_succ	#VP_avg	rate_valid-VP	time_avg (sec)	mem (Mb)
b14	153	50–218	0	108	45	50–101	1701	0.615	144.22	90.01
b15	370	143–306	0	51	319	143–306	1519	0.917	96.62	107.20
b17	1009	76–308	0	148	861	76–308	1645	0.904	87.12	125.84
C2670	6	78–122	0	1	5	78–122	1066	0.835	83.80	58.91
C5315	20	54–67	0	4	16	54–67	3041	0.914	50.90	51.34
C7552	36	50–194	0	2	34	50–194	1350	0.455	64.38	36.65
s938	1	66–66	0	0	1	66–66	3051	0.726	19.03	24.90
s1423	17	51–59	0	0	17	51–59	3092	0.723	13.66	25.34
s3330	1	87–87	0	0	1	87–87	3336	0.599	58.30	27.75
s9234	13	54–83	0	0	13	54–83	3482	0.857	37.86	35.33
s13207	3	212–212	0	0	3	212–212	569	0.908	70.26	50.62
s38417	256	53–99	6	72	178	53–99	1090	0.523	103.33	136.04
s38584	7	50–147	0	0	7	50–147	1120	0.924	47.13	51.56

Table 4.2: Two-output Ashenhurst decomposition

circuit	#pair	#var	#fail	#SAT_TO	#succ	#var_succ	#VP_avg	rate_valid-VP	time_avg (sec)	mem (Mb)
b14	123	50–223	18	65	40	50–125	1832	0.568	96.86	226.70
b15	201	145–306	0	31	170	145–269	1176	0.845	113.86	224.07
b17	583	79–310	0	88	495	79–308	676	0.824	103.12	419.35
C2670	5	78–123	0	1	4	78–123	254	0.724	66.95	55.71
C5315	11	56–69	0	2	9	56–69	370	0.594	59.20	60.05
C7552	21	56–195	0	2	19	56–141	188	0.465	89.57	78.67
s938	1	66–66	0	0	1	66–66	3345	0.720	61.24	34.77
s1423	14	50–67	0	0	14	50–67	3539	0.591	55.34	45.66
s3330	1	87–87	0	0	1	87–87	1278	0.423	66.83	47.43
s9234	12	54–83	0	0	12	54–83	2193	0.708	48.11	55.15
s13207	3	212–228	0	0	3	212–228	585	0.700	93.36	118.03
s38417	218	53–116	13	30	175	53–116	689	0.498	109.06	319.48
s38584	9	50–151	0	0	9	50–151	1656	0.713	46.17	207.78

uated both single-output and two-output Ashenhurst decompositions. For the latter case, we simultaneously decomposed a pair of functions with similar input variables. For a circuit, we heuristically performed pairwise matching among its transition and output functions to find function pairs having more common input variables for decomposition. Only function pairs with joint input variables equal to or more than 50 were considered to be decomposed. Note that the experiments target the study of scalability, rather than comprehensiveness as a synthesis methodology.

Tables 4.1 and 4.2 show the decomposition statistics of single-output and two-output decompositions, respectively. In these tables, Column 1 lists the circuits to be decomposed. Columns 2 lists the numbers of instances (i.e., functions for single-output decomposition and function pairs for two-output decomposition) with no less than 50 inputs. Column 3 lists the ranges of the input sizes of these instances. Column 4 lists the numbers of instances that we cannot find any successful variable partition within 60 seconds or within 1500 seed variable partitions. Column 5 lists the numbers of instances that are decomposable but spending over 30 seconds in SAT solving for the derivation of function g or h. Columns 6 and 7 list the numbers of successfully decomposed instances and the ranges of the input sizes of these successful instances, respectively. Columns 8 and 9 list the average numbers of tried seed partitions in 60 seconds and the average rates hitting valid seed partitions. Column 10 shows the average CPU times spending on decomposing an instance. Finally, Column 11 shows the memory consumption. As can be seen, our method can effectively decompose functions or function pairs with up to 300 input variables.

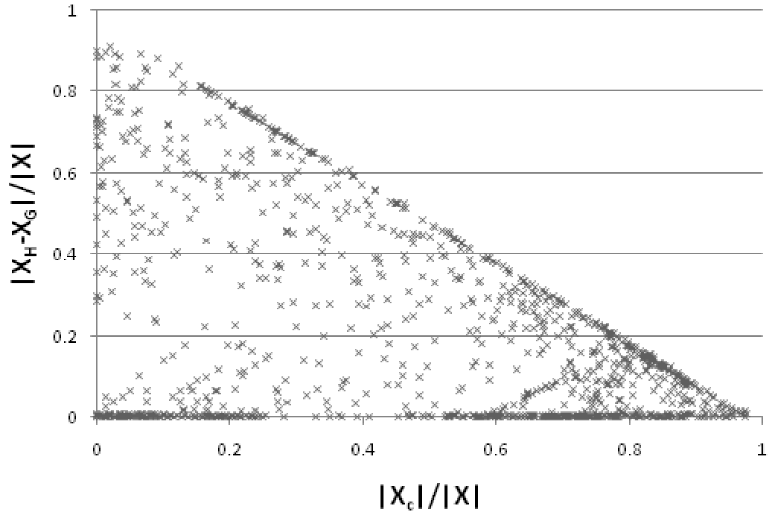


Figure 4.1: Best variable partition found in 60 seconds – without minimal UNSAT core refinement

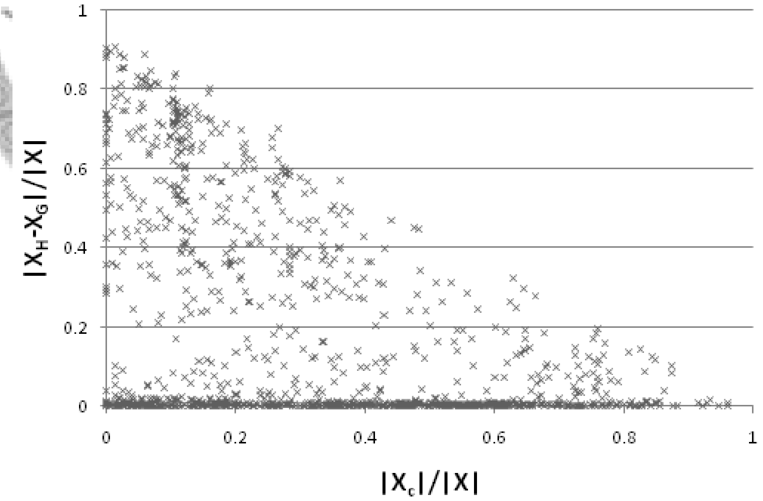


Figure 4.2: Best variable partition found in 60 seconds – with minimal UNSAT core refinement

4.2 Quality of Variable Partition

We measure the quality of a variable partition in terms of disjointness, indicated by $|X_C|/|X|$, and balancedness, indicated by $||X_G| - |X_H||/|X|$. The smaller these two values are, the better a variable partition is. Figures 4.1 and 4.2 show the disjointness and balancedness information for the best found variable partition within 60 seconds[1] for each decomposable instance. Note that when enumerating different seed variable partitions to find a better partition results, we emphasis on improving disjointness than balancedness.

Each spot on these two figures corresponds to a variable partition result for a decomposition instance who is decomposable. Figure 4.1 and Figure 4.2 show the variable partition results *without* and *with* further minimal unsatisfiable (UNSAT) core refinement process, respectively. Comparing Figures 4.1 and 4.2, we see that minimal UNSAT core refinement indeed can substantially improve the variable partition quality. Specifically, the improvement is 38.81% for disjointness and 17.70% for balancedness.

Figure 4.3 compares the qualities of variable partitioning under four different efforts. In the figure, "1st" denotes the first-found valid partition and "$t$sec" denotes the best-found valid partition within t seconds. The averaged values of $|X_C|/|X|$ and

[1] The search for a best variable partition may quit before 60 seconds if both disjointness and balancedness cannot be improved in consecutive 1500 trials.

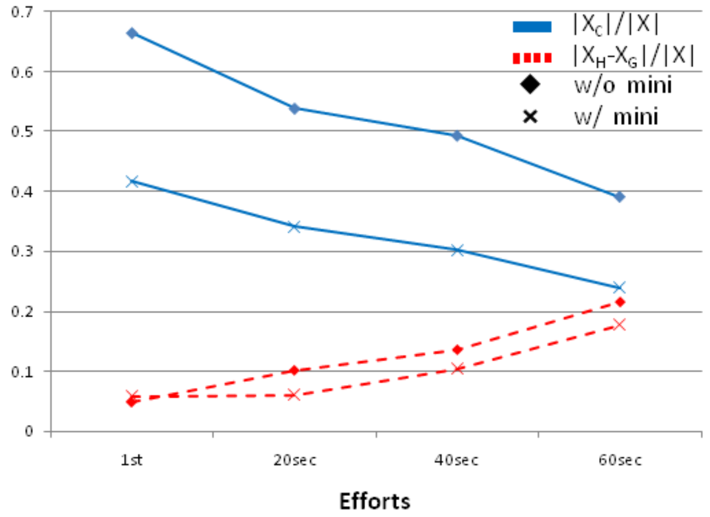


Figure 4.3: Variable partition qualities under four different efforts

$||X_G| - |X_H||/|X|$ with and without minimal UNSAT core refinement are plotted in this figure. In our experiments, improving disjointness is preferable to improving balancedness. These two objectives, as can be seen, are usually mutually exclusive. Disjointness can be improved when sacrificing balancedness, and vice versa. The figure also demonstrates that our proposed minimal UNSAT core refinement process can effectively improving the disjointness. It is interesting to note that, on average, 1337 seed partitions are tried in 60 seconds, in contrast to 3 seed partitions tried to identify the first valid one.

We observed that in many cases the run-time of variable partitioning dominates the total run-time of the algorithm. A further investigation suggests that the first-found valid variable partition may help reducing the run-time due to its first-found natural.

4.3 Fast Variable Partitioning

The next step we try to do is replacing the best found valid partition within 60 seconds by the first-found valid partition to reduce the run-time of variable partition step.

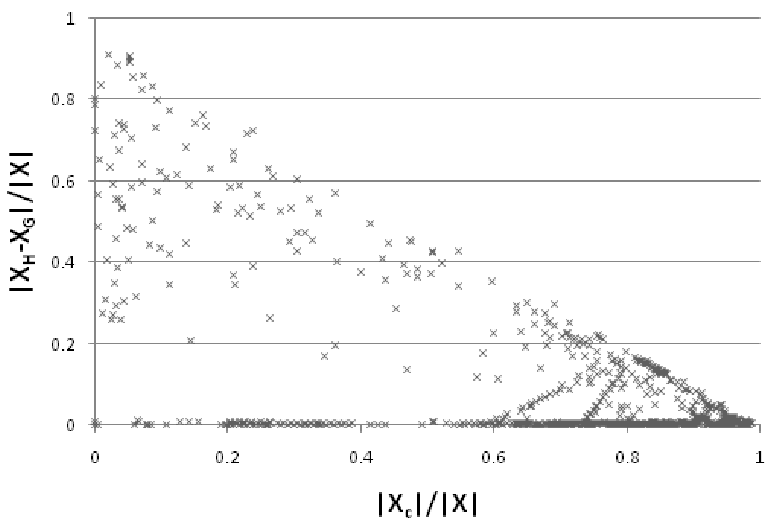


Figure 4.4: First-found valid partition – without minimal UNSAT core refinement

Figures 4.4 and 4.5 show the disjointness and balancedness of the first-found valid partition *without* and *with* the minimal UNSAT core refinement process. Every spot in the figures corresponds to a test case from the single- and two-output Ashenhurst decomposition in our experiments. As can be seen, for the partition results without

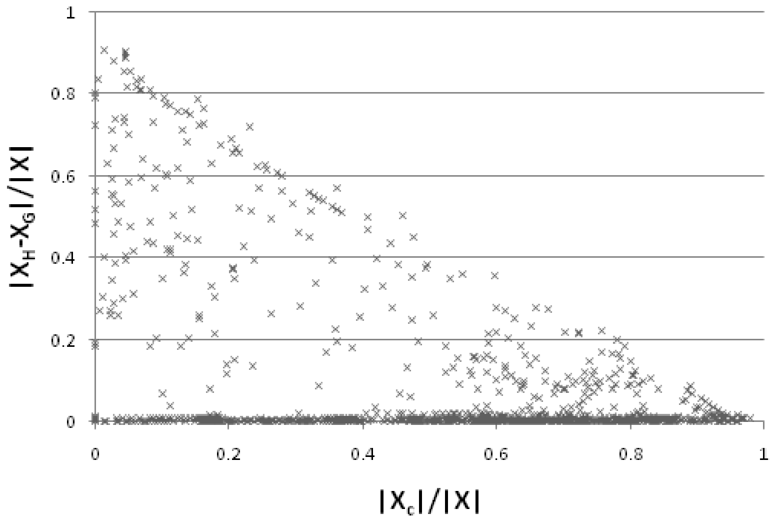


Figure 4.5: First-found valid partition – with minimal UNSAT core refinement

minimal UNSAT core refinement, the results of first-found valid partition has fewer points close to the origin compare to the best found variable partition in 60 seconds. It reveals that our partition technique without minimal UNSAT core refinement may not catch the local optimal solution in the first-found valid partition.

By introducing the minimal UNSAT core refinement process, there are more points close to the origin. That means the partition results are more balancedness and disjointness. After apply the minimal UNSAT core refinement process to the first-found valid partition, there are 37.23% improvement on disjointness but 19.44% balancedness loss (balancedness from 4.89% to 5.84%). Even though the ratio of balancedness loss seems high, but the actual value of balancedness is still very low. Since the minimal UNSAT core refinement process improved the quality of first-found valid partition, we are now interested in comparing the partition results of first-found valid partition with minimal UNSAT core process with the results of best found partition within 60 seconds without minimal UNSAT core process. One fact we observed is, for partition results of first-found valid partition with minimal UNSAT core process, which shows in Figure 4.5, the quantity of those points close to the origin is even more than the quantity of those in Figure 4.1, which corresponds to the best found partition in 60 seconds without minimal UNSAT core refinement process. It reveals that these two partition efforts have comparable qualities.

We next analyze the number of SAT solving needed between first valid partition with UNSAT core refinement and best partition within 60 seconds without UNSAT core refinement. Recall that in average, we need 3 seed partitions tried to find the first valid partition, in contrast to 1337 seed partitions tried in 60 seconds. In the other hand, the number of SAT solving we need in minimal UNSAT core refinement process is exactly the number of X_C variables. The maximum cardinality of X_C is no more than the number of input variables (in our experiments, the maximum number of input variables is 308).

The above analysis shows that the disjointness and balancedness of the first-found valid partition with minimal UNSAT core refinement process are comparable to the results of best found partition in 60 seconds without minimal UNSAT core operation. However the number of SAT solving needed is four times less. Hence if the timing cost of the partition process is asked to be small, the minimal UNSAT core refinement process is a non-sacrificed procedure in the partition step.

Table 4.3: Variable distribution in different partition efforts

effort	X_G	X_H	X_{GH}	X_C
first	0.013	0.087	0.259	0.641
first_mini	0.128	0.186	0.261	0.425
60	0.049	0.291	0.261	0.399
60_mini	0.141	0.342	0.262	0.255

Table 4.3 shows the average variable distribution of first-found valid partition and best found partition in 60 seconds. For example, the average X_G distribution is calculated as $average|X_G|/average|X|$. Note that the term "mini" here indicates the results with minimal UNSAT core refinement process. The column X_G, X_H, and X_C indicate the percentage of variables which belong to the variable partition X_G, X_H and X_C, respectively. The column X_{GH} indicates the percentage of variables which belong to either X_G or X_H partition. From the results, there are more variables strictly in X_H than in X_G no matter what different partition efforts we applied. There are almost one fourth of variables can be either in X_H or X_G, these free variables can be used to establish a more balanced partition. One thing we have to mention here is that, in our experiment, when evaluating disjointness and balanced, we let each X_{GH} variable be either in X_H or X_G such that the variable partition will be more balanced.

From Table 4.3, we know the fact that for average disjointness, the first-found valid partition with minimal UNSAT core operation and the best found partition in 60 seconds without minimal UNSAT core operation have almost the same value. The difference is less than 2.6%. We further use Figure 4.6 to demonstrate that not only for average disjointness but also for the disjointness for every test case, no any partition effort dominates another. The x-axis indicates the disjointness of first-found valid partition with minimal UNSAT core operation. The y-axis is the value for best found partition in 60 seconds without minimal UNSAT core operation. Also, every spot in the figure corresponds to a test case from the single- and two-output Ashenhurst decomposition in our experiments.

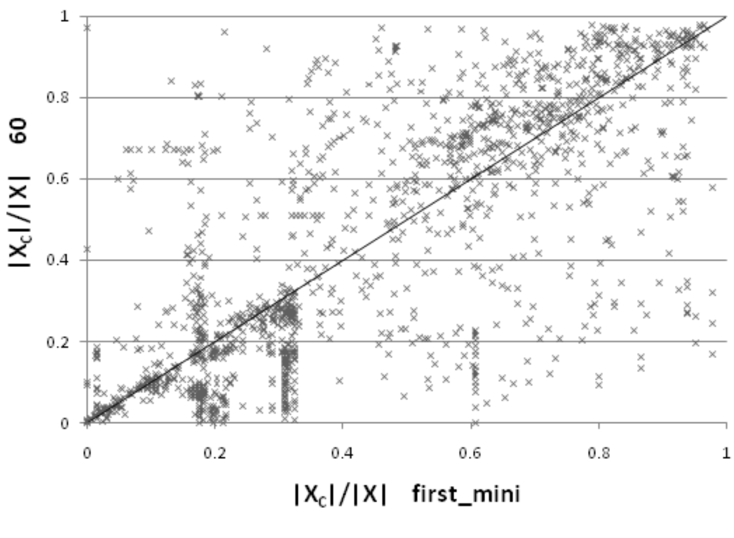


Figure 4.6: Comparison of disjointness between different partition efforts

We have mentioned that if the timing cost is asked to be small, the minimal UNSAT core refinement process is a non-sacrificed operation in the partition step.

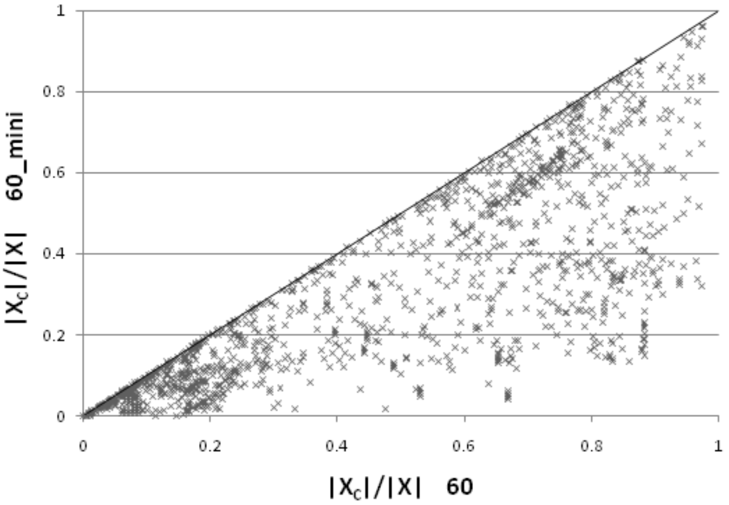


Figure 4.7: Comparison of disjointness between different partition efforts

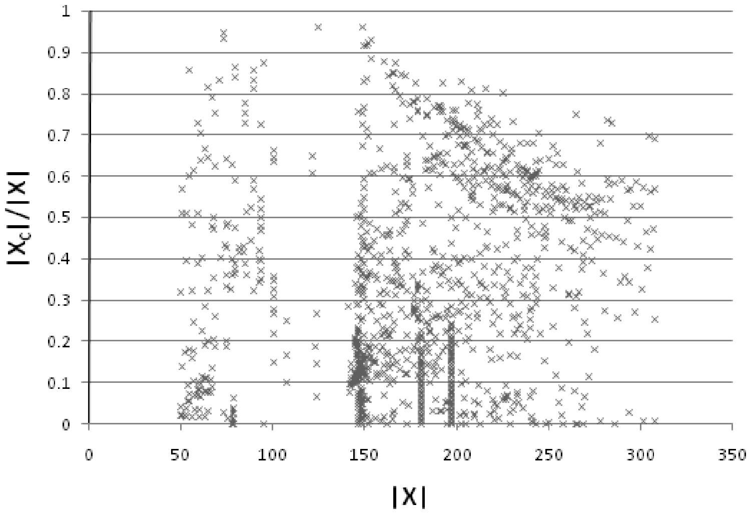


Figure 4.8: Disjointness versus total variables of the best found partition in 60 seconds with minimal UNSAT core operation

Although the first-found valid partition with minimal UNSAT core operation results in a similar partition quality compares to the best found partition in 60 seconds without minimal UNSAT core operation. On the other hand, if we further apply the minimal UNSAT core operation to the best found partition in 60 seconds, there are additional improvements in both disjointness and balancedness. Figure 4.7 shows the improvement of disjointness for every test cases in our single- and two-output Ashenhurst decomposition experiments. Furthermore, since in our experiments scalability is an important issue, Figure 4.8 shows the disjointness versus the total input variables for every test cases under "60_mini" partition effort. It shows that not only small circuits but also large circuits with more than 300 input variables have chances to find a more disjoint variable partition using our proposed method.

4.4 n-Output Ashenhurst Decomposition

We have shown that the proposed algorithm can handle single- and two-output Ashenhurst decompositions. Here, we choose a benchmark circuit s38584 to demonstrate that out proposed algorithm can successfully decompose functions with more than two outputs. Figure 4.9 shows the average run time of multi-output Ashenhurst decomposition. Just as the test case generation step in the two-output function decomposition, we heuristically choose n output functions that have more common input variables from the benchmark circuit s38584 to be the test cases. Note that for each n-output test case, every output function shares the same g function. As can be seen, the average runtime is proportional to the number of the outputs.

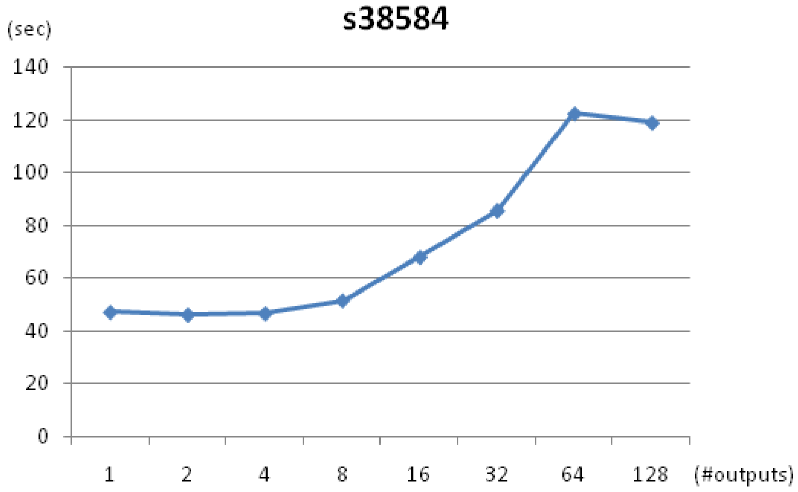


Figure 4.9: Runtime of multi-output Ashenhurst decomposition

4.5 Quality of Composite Functions

We take benchmark circuit s9234 as an example to show the circuit information of functions g and h derived from Ashenhurst decomposition. The results are shown in Table 4.4.

The first column indicates the different efforts we applied to the test cases before we feed it into the algorithm. “Sin” and “Dul” denote the single- and two-output Ashenhurst decomposition, respectively. “Clp” denotes ABC commands **collapse** and **strash** are further applied to collapse and structure hash a circuit. “Syn” de-

Table 4.4: Circuit information of s9234

effort	original			g			g_Clp			h			h_Clp		
	#n	#l	#sup	#n	#l	#sup	#n	#l	#sup	#n	#l	#sup	#n	#l	#sup
Sin	227	29	62	186	23	27	83	12	27	136	22	34	65	13	34
SinClp	130	16	54	147	25	28	51	12	27	234	30	33	97	14	33
SinClpSyn	130	16	54	270	35	28	63	12	27	212	28	34	88	14	34
SinSyn	128	14	54	135	20	22	37	9	21	169	25	36	99	14	36
Dul	242	29	65	155	28	20	56	10	20	294	39	49	122	16	49
DulClp	166	18	59	611	69	23	75	11	21	239	36	49	116	16	48
DulClpSyn	165	18	59	633	64	25	89	12	25	304	41	46	112	16	45
DulSyn	148	15	59	195	23	21	50	11	21	254	29	48	116	15	48

notes ABC command **resyn** is further applied to synthesize a circuit. The next three columns show the information of the test cases with different efforts as the pre-processing. The next six columns indicate the average number of nodes, levels, and support variables of the derived g function before and after applying the command **collapse** to the derived g function. The last six columns are the results for h function. Note that functions g and h in different rows may not have the same functionalities. Although the test cases in different rows have the same functionalities, they can have different circuit structures. The different circuit structures of the test cases result in different variable partitions as well as the functionality of derived functions g and h.

As can be seen, by ABC command **collapse** the size and level of the functions g and h are reduced. The same fact happens in the preprocessing step. One fact we can observe from the table is that, for function g without further optimized by **collapse**, the generated g in row “Dul” is smaller than the generated g in row “DulClp”.

It is interesting that smaller test cases(with **collapse**) may not generate smaller composite functions. The reason why this happens is because the derivation of the interpolant function is directly based on the refutation proof of an UNSAT instance. So for a given function with different circuit structures, the circuit with smaller size may not generate smaller refutation proof and so do the interpolant functions. However the sizes of g and h after command **collapse** are greatly reduced to sizes comparable to original test cases.

Chapter 5

Conclusions and Future Work

In this thesis, we have proposed a SAT-based algorithm to handle single- and multiple-output Ashenhurst decompositions. Our proposed method can handle not only disjoint decomposition, but also non-disjoint decomposition in a natural way. We formulated the problem into SAT solving; by using Craig interpolation and functional dependency, the decomposed sub-functions can be derived. Variable partition process can be automated and integrated into the algorithm by introducing the control variables for each input variable. To prevent trivial variable partition from happening, we enforced some seed partitions to achieve this goal. Furthermore, we enumerated different seed partitions to find a good variable partition which is more balanced and disjoint. In addition, the variable partition results can be further greatly improved by the proposed UNSAT core refinement process.

In our method, the formulation that solves the problem of single-output decomposition can be easily extended to deal with the multiple-output decomposition problem. Also, in comparison with the prior BDD-based algorithms, the bound set variables of our method need not be small. Functions with hundreds of variables, which cannot be represented by BDD, can be handled by our SAT-based algorithm. The scalability of our proposed method was justified by experimental results. The results showed that we can successfully decompose functions with up to 300 input variables. Because the experimental results showed that the method we proposed can handle large design instances, our approach may be applied at a top level of hierarchical decomposition in logic synthesis, which may provide a global view on optimization. For example, our algorithm may be used on chip-level decomposition to implement a large design instance using many FPGA cores. Also our algorithm can be a step forward towards topologically constrained logic synthesis.

For future work, how to effectively deal with general functional decomposition remains future investigation. The application of our approach to FPGA Boolean matching and multilevel hypergraph partitioning [13] are interesting subjects to study. In contrast to multilevel hypergraph partitioning, which focuses on structure-based manipulations, our Ashenhurst decomposition is a function-based algorithm. This is the major difference between the two subjects, although both of them can partition a circuit into two parts. Furthermore, for the characterization of don't-care conditions, how to quickly identify all sub-matrices that have only one kind of column pattern, and how to use these don't-care conditions to minimize the circuit will be the future study.

Bibliography

[1] R. L. Ashenhurst. The decomposition of switching functions. *Computation Lab*, Harvard University, Vol. 29, pp.74-116, 1959.

[2] Berkeley Logic Synthesis and Verification Group. *ABC: A system for sequential synthesis and verification.* `http://www.eecs.berkeley.edu/∼alanmi/abc/`

[3] J. Cong and Y.-Y. Hwang. Boolean matching for LUT-based logic blocks with applications to architecture evaluation and technology mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 20(9):1077-1090, 2001.

[4] J. Cong and K. Minkovich. Improved SAT-based Boolean matching using implicants for LUT-based FPGAs. In *Proc. ACM International Symposium on Field Programmable Gate Arrays*, pp.139-147, 2007.

[5] S. A. Cook. The complexity of theorem proving procedures. *Proc. Third Annual ACM Symposium on the Theory of Computing*, pp.151-158, 1971.

[6] W. Craig. Linear reasoning: A new form of the Herbrand-Gentzen theorem. *J. Symbolic Logic*, 22(3):250-268, 1957.

[7] H. A. Curtis. *New Approach to the Design of Switching Circuits.* Van Nostrand, Princeton, NJ, 1962.

[8] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201-215, 1960.

[9] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394-397, 1962.

[10] N. Eén and N. Söensson. An extensible SAT-solver. In *Proc. International Conference on Theory and Applications of Satisfiability Testing*, pp.502-518, 2003.

[11] J.-H. R. Jiang and R. K. Brayton. Functional dependency for verification reduction. In *Proc. International Conference on Computer Aided Verification*, pp.268-280, 2004.

[12] J.-H. R. Jiang, J.-Y. Jou, and J.-D. Huang. Compatible class encoding in hyper-function decomposition for FPGA synthesis. In *Proc. ACM/IEEE Design Automation Conference*, pp.712-717, 1998.

[13] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel Hypergraph Partitioning: Application in VLSI Domain. In *Proc. ACM/IEEE Design Automation Conference*, pp.526-529, 1997.

[14] R. M. Karp. Functional decomposition and switching circuit design. *J. Soc. Ind. Appl. Math.* 11(2):291-335, 1963.

[15] Y.-T. Lai, M. Pedram, and S. B. K. Vrudhula. BDD based decomposition of logic functions with applications to FPGA synthesis. In *Proc. ACM/IEEE Design Automation Conference*, pp.642-647, 1993.

[16] C.-C. Lee, J.-H. R. Jiang, C.-Y. Huang, and A. Mishchenko. Scalable exploration of functional dependency by interpolation and incremental SAT solving. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pp.227-233, 2007.

[17] R.-R. Lee, J.-H. R. Jiang, and W.-L. Hung. Bi-decomposing large Boolean functions via interpolation and satisfiability solving. In *Proc. ACM/IEEE Design Automation Conference*, pp.636-641, 2008.

[18] H.-P. Lin, J.-H. R. Jiang, and R.-R. Lee. To SAT or Not to SAT: Ashenhurst Decomposition in a Large Scale. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pp.32-37, 2008.

[19] A. C. Ling, D. P. Singh, and S. D. Brown. FPGA technology mapping: A study of optimality. In *Proc. ACM/IEEE Design Automation Conference*, pp.427-432, 2005.

[20] K. L. McMillan. Interpolation and SAT-based model checking. In *Proc. International Conference on Computer Aided Verification*, pp.1-13, 2003.

[21] P. Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *J. Symbolic Logic*, 62(3):981-998, September 1997.

[22] J. P. Roth and R. M. Karp. Minimization over Boolean graphs. *IBM Journal*, pp.227-238, 1962.

[23] C. Scholl. *Functional Decomposition with Applications to FPGA Synthesis.* Kluwer Academic Publishers, 2001.

[24] H. Sawada, T. Suyama, and A. Nagoya. Logic synthesis for look-up table based FPGAs using functional decomposition and support minimization. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pp.353-358, 1995.

[25] T. Stanion and C. Sechen. A Method for Finding Good Ashenhurst Decompositions and Its Application to FPGA Synthesis. In *Proc. ACM/IEEE Design Automation Conference*, pp.60-64, 1995.

[26] S. Safarpour, A. G. Veneris, G. Baeckler, and R. Yuan. Efficient SAT-based Boolean matching for FPGA technology mapping. In *Proc. ACM/IEEE Design Automation Conference*, pp.466-471, 2006.

[27] G. S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, pp.466-483, 1970.

[28] K.-H. Wang and C.-M. Chan. Incremental learning approach and SAT model for Boolean matching with don't cares. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pp.234-239, 2007.

[29] B. Wurth, K. Eckl, and K. Antreich. Functional Multiple-Output Decomposition: Theory and an Implicit Algorithm. In *Proc. ACM/IEEE Design Automation Conference*, pp.54-59, 1995.

[30] B. Wurth, U. Schlichtmann, K. Eckl, and K. Antreich. Functional multiple-output decomposition with application to technology mapping for lookup table-based FPGAs. *ACM Trans. on Design Automation of Electronic Systems*, 4(3):313-350, 1999.