國 立 交 通 大 學

電 子 工 程 學 系　電 子 研 究 所 碩 士 班

碩　士　論　文

以圖型自同構為基礎之自動接線驗證樣本產生器

On Automatic Pattern Generation for Interconnect Verification Based on Graph Automorphism

研 究 生：周貞伶

指導教授：周景揚 博士

中華民國 九十三 年 六 月

以圖型自同構為基礎之自動接線驗證樣本產生器

On Automatic Pattern Generation for Interconnect Verification Based on Graph Automorphism

研 究 生：周貞伶	Student: Chen-Ling Chou
指導教授：周景揚 博士	Advisor: Dr. Jing-Yang Jou

國立交通大學
電子工程學系 電子研究所碩士班
碩士論文

A Thesis
Submitted to Institute of Electronics
College of Electrical Engineering and Computer Science
National Chiao Tung University
In Partial Fulfillment of the Requirements
For the Degree of
MASTER OF SCIENCE
In
Electronics Engineering
June 2004
Hsinchu, Taiwan, Republic of China

中華民國　九十三　年　六　月

以圖型自同構為基礎之自動接線驗證樣本產生器

研究生：周貞伶　　　指導教授：周景揚　博士

國立交通大學

電子工程學系 電子研究所碩士班

摘　　要

在大型系統晶片 (system-on-a-chip, SoC) 設計中，嵌入式核心(embedded cores)的使用正大量的增加。高複雜度的系統晶片設計，使得設計驗證對系統整合者來說是個很大的挑戰。為了降低以嵌入式核心為基礎的設計驗證的複雜度，連接埠順序障礙模型 (port-order-fault，POF) 已經被提出，其對應的驗證向量產生器也已經發展好了。在本論文中，為了解決大型系統晶片連接驗證的問題，我們在連接埠順序障礙模型之下提出了一個以圖形自同構 (graph automorphism) 為基礎的演算法來改善這個自動驗證向量產生器的效率。此外，這個演算法也可以運用於計算電路的輸入對稱的最大集合。我們測試了一些 ISCAS-85 和有較多輸入連接埠的 MCNC 電路。由實驗結果顯示，這個演算法可以在更少的時間下產生更有效的驗證向量。

On Automatic Pattern Generation for Interconnect Verification Based on Graph Automorphism

Student: Chen-Ling Chou Advisor: Dr. Jing-Yang Jou

Department of Electronics Engineering &
Institute of Electronics
National Chiao Tung University

Abstract

Embedded cores are being increasingly used in the design of large system-on-a-chip (SoC). High complexities of SoC designs lead the design verification to be a challenge for system integrators. To reduce the verification complexity, the port-order-fault (POF) model has been proposed and the corresponding verification pattern generation has been developed for verifying core-based designs. This thesis proposes a graph automorphism-based algorithm to improve the efficiency of the automatic verification pattern generation (AVPG) for SoC interconnect verification based on the POF model. Furthermore, this algorithm can be applied to compute maximal sets of symmetric inputs of circuits. We conduct the experiments on ISCAS-85 and some MCNC benchmarks with large inputs of circuits. The experimental results demonstrate that our approach generates more efficient patterns with less CPU time.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor, Professor Jing-Yang Jou (周景揚), for his insightful suggestion and patient guidance throughout the course of this work in the two years of my graduate life. I am still very grateful for his help in many other matters, academically and socially. I am also indebted to Chun-Yao Wang (王俊堯) and Geeng-Wei Lee (李耿維), for their great help and constructive suggestions on my research. Thanks also go to all members in the Electronic Design Automation Lab for their friendship and all members of the intramural badminton team. Finally I have to show my deepest gratefulness to my parents, 周乾德 and 蔣慧月, my sister, 周芳俞, and my boyfriend, Hung-Chih Lai (賴鴻志). Without their support and love, this work would not be complete.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Spurred by process technology leading to the availability of more than 1 million gates per chip, and more stringent requirements upon time-to-market and performance constraints, system level integration and platform-based design [3] are evolving as a new paradigm in system designs. A multitude of components that are needed to implement the required functionality make it hard for a company to design and manufacture an entire system in time and within reasonable cost. Hence, design reuse and reusable building blocks (cores) trading are becoming popular in the system-on-a-chip (SoC) era. However, present design methodologies are not enough to deal with cores which come from different design groups and are mixed and matched to create a new system design. In particular, verifying whether a design satisfies all requirements is one of the most difficult tasks.

Usage of cores divides the IC design community into two groups: core providers and system integrators. In traditional system-on-board (SoB) design, the components that go from provider to system integrator are ICs, which are designed, verified, manufactured, and tested. The system integrator verifies the design by using these components as fault-free building blocks. SoB verification is limited to detecting faults in the interconnection among the components. Similarly, in SoC design, the components are cores. The system integrator verifies the design by using the cores as design error-free building blocks. The focus of this core-based design verification should be on how the cores communicate with each other [13]. However, before the interface verification, the interconnection between the cores in an SoC have to be verified first. This is because the SoC integrator has to connect a large number of ports (hundreds or even thousands of ports) in an SoC. The likelihood of interconnection misplacements between the cores is high. Furthermore, the correct interconnection between the cores is the minimum requirement to verify the interface protocols. In other words, if the interconnections between the cores are misplaced, the process of the verification on the interface between the cores will be in vain. Thus, the interconnection verification can be conducted as the first step to the interface verification between the cores in an SoC.

Most previous work in testing interconnection focused on the development of deterministic tests for interconnection between chips at the board level [7, 8]. The main purpose is to test if the interconnections are connected properly (neither short nor open). In the interconnection testing phase, the basic assumption for a system under test is that the system design is correct, and the faults are due to manufacturing defects on interconnection among components. For the core-based SoC design verification, however, the system is not fully verified yet and the most of system design errors are due to the incorrect interconnection among predesigned cores. The incorrect interconnections are normally introduced by the misinterpretation of port description of IP cores, and this misinterpretation is usually caused by some factors, such as ambiguous or cryptic port names, Big Endian or Little Endian byte order of address bus, etc. Therefore, the extension of these board level testing methods is inadequate for connectivity-based design verification. Fig. 1.1 (a) and (b) shows the schemes to demonstrate the processes of interconnection testing and interconnection verification, respectively. In the interconnection testing, the engineers focus on the success of implementation of interconnected wires between block1 and block2. The testing patterns and corresponding responses are applied and observed at the ends of the interconnects to check whether the interconnects are manufactured correctly. On the other hand, in the interconnection verification, the system integrators verify whether the interconnections between block1 and block2 are located in the correct ports. They apply the verification patterns to primary inputs (PIs) of the integrated design, then observe the corresponding responses in primary outputs (POs) of the integrated design, and match them against the specification instead.

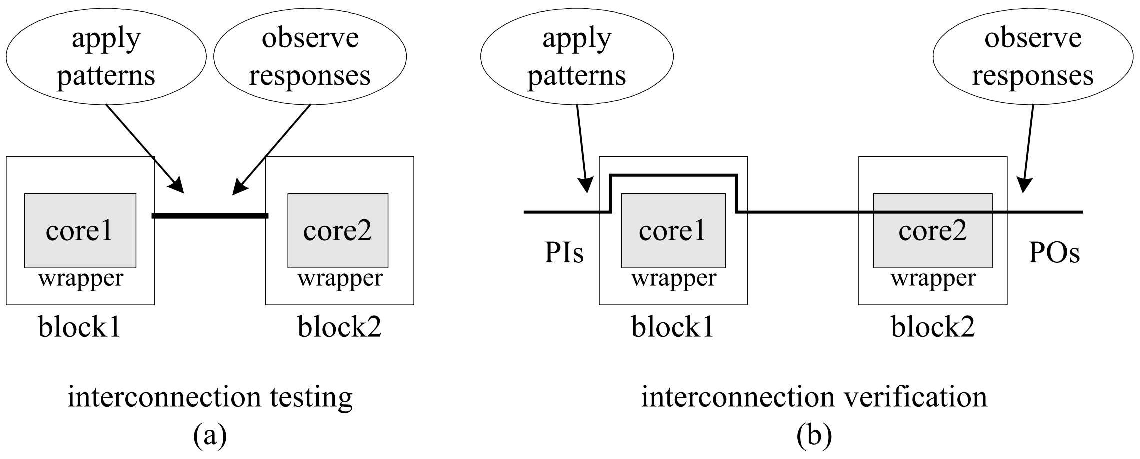


Figure 1.1. The schemes of interconnection testing and interconnection verification.

By creating the testbenches at a high level, a connectivity-based design fault model, port-order-fault (POF), proposed in [17], is used for reducing the time on core-based design verification [18]. And the superset of all automorphism (SAA) technique is proposed in [19] to accelerate the AVPG and reduce the size of verification pattern set.

In this thesis, we propose a graph automorphism-based algorithm to further reduce the size of verification pattern set. This algorithm also can compute maximal sets of symmetric inputs of circuits, which are important to design verification and diagnosis [9, 10, 12, 16], as well as technology mapping [5, 11, 14].

In general, the manufacturing defects are called faults. Since this approach is conducted for design verification rather than chip testing, we rename the POF model as port-order-error (POE) without changing the model definition.

We exploit the IEEE P1500 wrappers and user defined Test Access Mechanisms (TAMs) to propagate the verification patterns from PIs to the wrappers in the predecessor of the core under verification (CUV) and to propagate responses of the CUV to POs. The integration verification mechanism with the IEEE P1500 standard-under-development is the same with that introduced in [18].

The remainder of this thesis is organized as follows: the POE model and previous work are introduced in Chapter 2. Chapter 3 describes the graph automorphism (GA) technique in the AVPG. The experimental results are shown in Chapter 4 and the applications of symmetric detection are shown in Chapter 5. Chapter 6 concludes the thesis.

Chapter 2

Preliminary

2.1. POE Model

The POE model belongs to the group of pin-errors models [1], which assumes that an erroneous cell has at least two I/O ports misplaced. It also assumes that the components are error free and only the interconnections among the components could be erroneous. There are three types of POEs [17].

Definition 2.1: The type I POE is at least an output misplaced with an input. The type II POE is at least two inputs misplaced. The type III POE is at least two outputs misplaced. It has been proven that the type II POEs dominate the other two types of

POEs [18]. Hence, in this thesis, the AVPG focuses on the type II POEs solely.

Definition 2.2: A port sequence is an input port number permutation that indicates the relative positions among these input ports.

Definition 2.3: The error-free-port-sequence (EFPS) is a port sequence that none of the input ports were misplaced. For an N-input core, its input variables are numbered from 1 to N. The number of the input variable permutations is N! and these N! permutations represent the N! port sequences of the core. Except the error-free-port-sequence, the remaining (N!-1) port sequences represent those corresponding cores with particular POEs and are called erroneous-port-sequences (EPSs). In this thesis, the POEs and the EPSs are used exchangeably.

Given a 4-input core, the input ports are numbered from 1 to 4. Any input port numbers permutation is a port sequence of the core. It has 4! port sequences totally. The only one EFPS is 1234, the remaining (4!-1) port sequences are EPSs. The EPS 1423 represents the port 4 is connected to the location of port 2, the port 2 is connected to the location of port 3, and the port 3 is connected to the location of port 4. The schematic representation of EFPS 1234 and EPS 1423 are shown in Fig. 2.1 (a) and Fig. 2.1 (b), respectively.

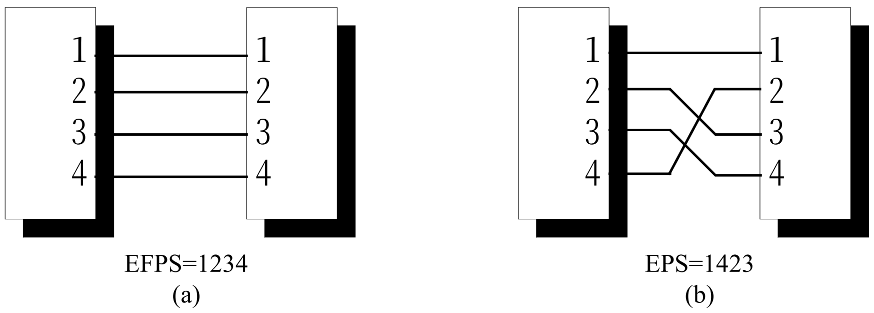


Figure 2.1. The schematic representation of EFPS 1234 and EPS 1423.

2.2. Undetected Port Sequences (UPSs) Representation

Typically, the automatic pattern generator for functional errors, such as transition faults [6], or manufacturing faults, such as stuck-at-fault (SAF) [15], builds fault list explicitly first to explore how many faults have to be detected, and then generates random patterns and deterministic patterns to detect these faults in the fault list. For the POE-based AVPG, the error list is not enumerated explicitly. This is because the total number of POEs in an N-input core is (N!-1). This number grows rapidly when N increases, for instance, as N = 70, $N!-1 \approx 1.2\times10^{100}$. Instead, an implicit representation is used to indicate the remaining undetected port sequences (UPSs). According to this implicit representation, we can realize exactly what the remaining UPSs are and the further verification patterns can be generated accordingly in the verification pattern generation. The following example demonstrates this implicit UPSs representation.

Given an 8-input core, the input ports are numbered from 1 to 8. The UPSs representation (12345678) represents the UPSs that caused by all possible misplacements among the port numbers in the same group, i.e. port 1 to port 8. The number of undetected POEs is 8!-1, and the 1 in the 8! accounts for the error free port sequence. The UPSs representation (125)(4)(3678) indicates the UPSs that caused by all possible misplacements among the port numbers 1, 2 and 5 and/or all possible misplacements among the port numbers 3, 6, 7 and 8. The number of the undetected POEs is $3! \times 1! \times 4!-1$. Please note that the port number 4 is the only one element in the second group. It means that the port sequences whose port number 4 in the wrong position are not represented by this UPSs representation. The order of the groups in the UPSs representation is irrelevant, neither is the order of the numbers in each UPSs group. For example, the UPSs (125)(4)(3678) can also be expressed as (4)(215)(8763). The UPSs (12)(3)(4)(5)(6)(78) contain 4 port sequences and they are 12345678, 21345678, 12345687 and 21345687. The UPS representation (1)(2)(3)(4)(5)(6)(7)(8) has eight groups and each group has only one element; therefore, no misplacement could be occurred in each group. The number of the undetected POEs is $1!\times1!\times1!\times1!\times1!\times1!\times1!\times1!-1=0$. Hence, (1)(2)(3)(4)(5)(6)(7)(8) represents 8!-1 POEs are all detected. If the UPSs representation is induced from (12345678) to (1)(2)(3)(4)(5)(6)(7)(8), all POEs are detected.

2.3. Previous Work

The AVPG for verifying the interconnect in a core-based design is proposed in [18]. Its flowchart is shown in Fig. 2.2 again. The AVPG reads the combinational core and generates heuristic patterns. The patterns simulation results determine the valid verification patterns. Then the UPSs are calculated in UPSs_Calculation stage, so that more further verification patterns can be generated accordingly. When the error coverage (E_C) reaches 100%, i.e., the verification patterns for detecting all EPSs are generated, or the iterations are over the bound, the AVPG will be terminated.

The UPS's calculation (UPSs_Calculation) procedure shown in Fig. 2.2 determines what the remaining UPSs are and guides the further pattern generation. If the results of UPS's calculation are not precise enough, some of the further verification patterns could be redundant and the processing time to reach the desired error coverage will increase. In [18], the characteristic vector (CV) approach of determining the remaining UPSs encountered this weakness. However, the superset of all automorphisms (SAA) technique proposed in [19] improves the weakness mentioned and indeed increases the AVPG efficiency. This fact can be seen in Fig. 2.3. Fig. 2.3 is a drawing from [19], which shows the hierarchical relations on computing remaining UPSs between the CV approach [18] and the SAA approach [19]. It has five levels from the center to the boundary. The first level represents the set of real remaining UPSs explicitly. The second level is the implicit UPS representation of the first level. The third level represents the UPSs that are obtained by the automorphism approach. The SAA approach and CV approach are shown in the fourth and fifth levels, respectively. Since the UPSs in the inner levels are the subset of the outer levels, the SAA approach really gets better results than CV approach.

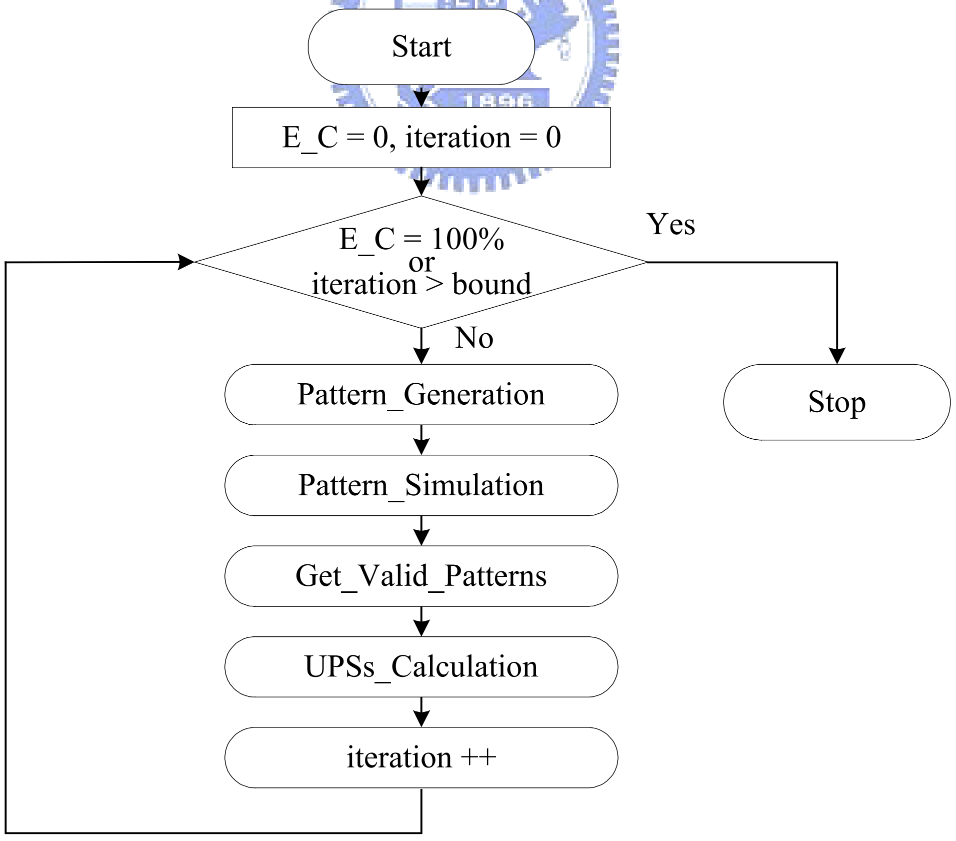


Figure 2.2. The flowchart of the POE-based AVPG.

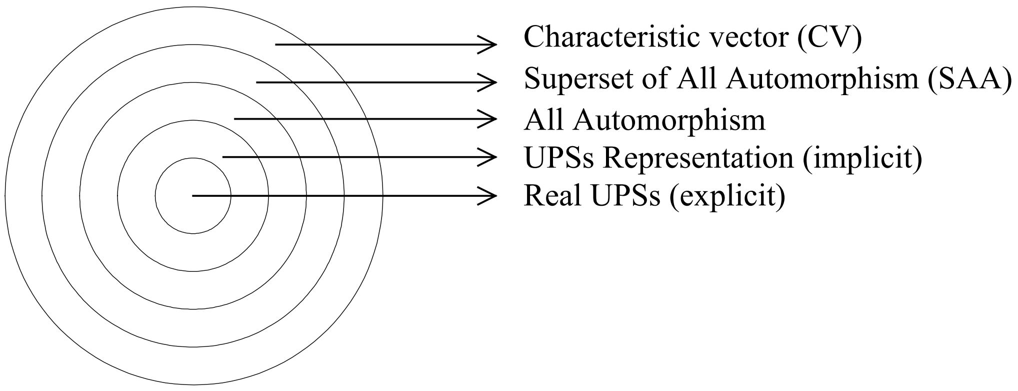

Characteristic vector (CV)
Superset of All Automorphism (SAA)
All Automorphism
UPSs Representation (implicit)
Real UPSs (explicit)

Figure 2.3. Hierarchical relations among the different approaches.

Chapter 3

Graph Automorphism–based AVPG

We observe that the stages which profoundly influence the efficiency of AVPG are Pattern_Generation and UPSs_Calculation. Thus, in addition to advancing the UPSs_Calculation stage like [19], this thesis proposes a new pattern generation procedure as well, which corresponds to the proposed UPSs_Calculation. In this chapter, we first explain the new Pattern_Generation procedure. Then we introduce the graph automorphism algorithm. The new UPSs_Calculation procedure based on the graph automorphism algorithm is described in the last section.

3.1. Pattern Generation

Definition 3.1 : For an N-input combinational core, the exhaustive pattern set is defined as Φ^N. The size of Φ^N is the number of patterns in Φ^N, and is denoted as $|\Phi^N|$ and $|\Phi^N|$ equals 2^N.

Definition 3.2 : The set that consists of all patterns with m 1s and (N-m) 0s is denoted as θ_m^N ,where m$\in$ [0, 1, 2, . . ., N -1, N]. The size of θ_m^N is the number of patterns in θ_m^N and is denoted as $|\theta_m^N|$ and $|\theta_m^N|$ equals C_m^N.

Example 3.1: For a 4-input core, Φ^4 is the exhaustive pattern set with 4 bits. $|\Phi^4| = 2^4 = 16$. θ_0^4 ={0000}, $|\theta_0^4| = C_0^4 =1$. θ_1^4={1000, 0100, 0010, 0001}, $|\theta_1^4| = C_1^4 =4$. θ_2^4= {1100, 1010, 1001, 0110, 0101, 0011}, $|\theta_2^4| = C_2^4 = 6$. θ_3^4= {1110, 1101, 1011, 0111}, $|\theta_3^4| = C_3^4 =4$. θ_4^4= {1111}, $|\theta_4^4| = C_4^4 =1$.

The target of AVPG is to generate valid verification patterns such that all N!-1 POEs are detected. To detect a POE, the error effect has to be activated first. If the error effect is not activated, it surely cannot be propagated out for detection. Thus, all N!-1 POEs have to be activated during the Pattern_Generation stage. For general cases, all remaining POEs have to be activated in each iteration. To activate a POE, the logic assignments of the corresponding input ports cannot be all the same. For example, to activate the EPS 1243, the assignments of port 3 and port 4 have to be different, either port 3 is assigned 0, port 4 is assigned 1, or vice versa. Furthermore, to increase the AVPG efficiency, the generated verification pattern cannot be repeated.

Lemma 3.1: Given a pattern T $\in \theta_m^N$, there are *m* 1s and (*N*-*m*) 0s in T. If the pattern

T turns to T' after a EPS λ, then T' $\in \theta_m^N$ and is a permutation of T by λ.

Theorem 3.1: θ_m^N can activate all (N!-1) POEs where m $\in$[1, 2, …, N-2, N-1]. [18]

Proof: θ_m^N contains all patterns with m 1s and (N-m) 0s. According to Lemma 3.1, there must exist a pair of patterns T and T' $\in \theta_m^N$ that corresponds to the original pattern T and the activated pattern T' for each POE. Thus, (N!-1) POEs are all activated by θ_m^N for m $\in$[1, 2, …, N-2, N-1]. Q.E.D.

Theorem 3.2: For an N-input core, the pattern set $\in$ $\theta_{m_1}^{n_1}$ X $\theta_{m_2}^{n_2}$ X … X $\theta_{m_k}^{n_k}$ with $n_1 + n_2 + \ldots + n_k = N$, $0 \leqq m_p \leqq n_p$ where p = 1,2,…,k and there being at least one group with different logic assignments can activate all $(n_1!) \times (n_2!) \times \cdots \times (n_k!)-1$ errors in the UPSs $(a_1a_2\cdots a_{n_1})(b_1b_2\cdots b_{n_2})\cdots(r_1r_2\cdots r_{n_k})$.

Proof: According to Theorem 3.1, $\theta_{m_1}^{n_1}$ can activate (n_1!-1) POEs where $m_1 \in$ [1, 2,…, n_1-1]. Obviously, $\theta_{m_1}^{n_1}$ X $\theta_{m_2}^{n_2}$ is to repeat $\theta_{m_1}^{n_1}$ $C_{m_2}^{n_2}$ times and to repeat $\theta_{m_2}^{n_2}$ $C_{m_1}^{n_1}$ times. For each pattern in $\theta_{m_2}^{n_2}$ with a set of $\theta_{m_1}^{n_1}$ can activate (n_1!)-1 POEs, $\theta_{m_1}^{n_1}$ X $\theta_{m_2}^{n_2}$ can activate $(n_1!)\times(n_2!)-1$ POEs. We can extend this to k groups, therefore, each pattern set $\in$ $\theta_{m_1}^{n_1}$ X $\theta_{m_2}^{n_2}$ X…X $\theta_{m_k}^{n_k}$ can activate $(n_1!)\times(n_2!)\times\cdots\times(n_k!)-1$ errors, same as the number of POEs in the UPSs $(a_1a_2\cdots a_{n_1})(b_1b_2\cdots b_{n_2})\cdots(r_1r_2\cdots r_{n_k})$.

Q.E.D.

The simulation results of the verification patterns are observed to determine which activated POEs are propagated to POs. The error effects are propagated to POs if there exists different responses among these verification patterns.

Example 3.2: For UPSs (12)(345), we know that the assignment θ_1^3 in (345) can activate 3!-1 errors according to Theorem 3.1. When 1 and 2 are involved, we can assign 10 or 01 for the (12) and combine with the assignment in (345) as shown in Fig. 3.1. The error activation of 21435 can be obtained from 12435 by comparing 10010 and 01010. All other EPSs can be activated by the similar process as well. Therefore, the POEs in (12)(345) are all activated by the $\theta_1^2 \times \theta_1^3$.

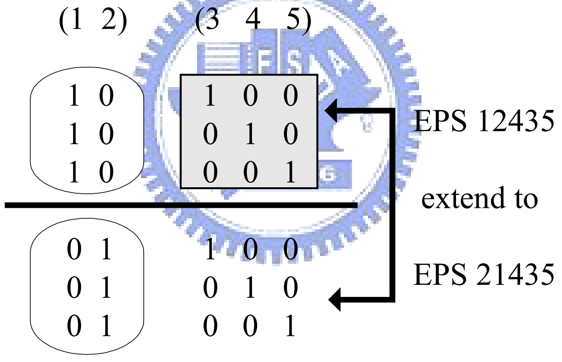


Figure 3.1. Error activation.

We use an example to demonstrate the details of pattern generation stage (error activation and error propagation). Given a 5-input core, assume the UPSs currently are (123)(45) and θ_1^5, θ_4^5 have been simulated. In this case, the further pattern sets come from $\theta_{m_1}^3 \times \theta_{m_2}^2$ where $m_1 \in$ [0, 1, 2, 3] and $m_2 \in$ [0, 1, 2]. Therefore, the possible further pattern sets are A_1 ~A_{12} as listed in Table 3.1 and A_5, A_6 are shown in Fig. 3.2.

Table 3.1. All possible pattern sets.

Possible further pattern set	UPSs = (123)(45)	Explanation
A_1	$(m_1=0,m_2=0)$	$\in \theta_0^5$, cannot activate remaining POEs
A_2	$(m_1=0,m_2=1)$	$\in \theta_1^5$, have been simulated
A_3	$(m_1=0,m_2=2)$	$\in \theta_2^5$, cannot activate remaining POEs
A_4	$(m_1=1,m_2=0)$	$\in \theta_1^5$, have been simulated
A_5	$(m_1=1,m_2=1)$	$\in \theta_2^5$
A_6	$(m_1=1,m_2=2)$	$\in \theta_3^5$
A_7	$(m_1=2,m_2=0)$	$\in \theta_2^5$
A_8	$(m_1=2,m_2=1)$	$\in \theta_3^5$
A_9	$(m_1=2,m_2=2)$	$\in \theta_4^5$, have been simulated
A_{10}	$(m_1=3,m_2=0)$	$\in \theta_3^5$, cannot activate remaining POEs
A_{11}	$(m_1=3,m_2=1)$	$\in \theta_4^5$, have been simulated
A_{12}	$(m_1=3,m_2=2)$	$\in \theta_5^5$, cannot activate remaining POEs

Further pattern set A5
$(m_1=1,m_2=1)$

```
(1 2 3) (4 5)
 1 0 0   1 0
 0 1 0   1 0
 0 0 1   1 0
 1 0 0   0 1
 0 1 0   0 1
 0 0 1   0 1
```

Further pattern set A6
$(m_1=1,m_2=2)$

```
(1 2 3) (4 5)
 1 0 0   1 1
 0 1 0   1 1
 0 0 1   1 1
```

Figure 3.2. Possible further pattern sets in A5 and A6.

Note that A_2 and A_4 are $\in \theta_1^5$ as well as A_9 and A_{11} are $\in \theta_4^5$. They have already been simulated before, thus, they are skipped from further verification pattern sets. Furthermore, A_1, A_3, A_{10}, and A_{12} have the same logic assignments in each group. They cannot be the verification pattern sets either. Consequently, the remaining verification pattern sets are $A_5 \sim A_8$, which are $\in \theta_2^5$ and θ_3^5 respectively. We can choose only θ_2^5, or θ_3^5, or both θ_2^5 and θ_3^5 for further verification sets since they all activate the remaining EPSs and have the same opportunity to reduce the UPSs. To reduce the complexity of pattern generation, we choose one θ_m^5 at a time. In this example, either θ_2^5 or θ_3^5. Table 3.2 shows the selected verification patterns of θ_2^5 and their corresponding outputs which are represented in symbolic output representation. For example, the pattern set A_5 contains six patterns, {10010, 01010, 00110, 10001, 01001, 00101} and their outputs are a1, a2, a1, a3, a3, and a3, respectively. The patterns in A_5 can be grouped into three groups {10010, 00110}, {01010}, and {10001, 01001, 00101} according to their outputs. To select a group of patterns as the valid verification patterns, we always choose the group with the smaller size if it indeed can detect new EPS. Thus, in this example, {01010} and {10010, 00110} are selected as the valid verification patterns. When we apply these patterns {01010}, {10010, 00110} to verify the interconnections, we expect the outputs to be a2 and a1, respectively. If the real outputs are not a2 and a1, then the misplaced interconnects are detected. For the same reason, the gray patterns in A_7 can be the valid verification patterns. After having these valid verification patterns, we have to figure out what new EPSs are detected and determine the remaining UPSs for further pattern generation.

Table 3.2. θ_2^5 pattern sets.

Verification pattern sets θ_2^5	UPSs =(123)(45)	Outputs	Description
A_5	1 0 0 1 0	a1	m_1=1, m_2=1, target on the both groups; {01010}, {10010, 00110} are valid verification pattern sets
	0 1 0 1 0	a2	
	0 0 1 1 0	a1	
	1 0 0 0 1	a3	
	0 1 0 0 1	a3	
	0 0 1 0 1	a3	
A_7	1 1 0 0 0	b1	m_1=2, m_2=0, target on the first group; {11000} is a valid verification pattern set
	1 0 1 0 0	b2	
	0 1 1 0 0	b2	

The UPSs_Calculation stage is achieved by the proposed graph automorphism algorithm. Thus, we first introduce this algorithm, then apply it on the UPSs_Calculation.

3.2. Graph Automorphism Algorithm

The Graph Automorphism (GA) problem is a well-known and well- studied problem. However, it is not known to be either in P or NP-complete [2, 4]. Here we propose a heuristic to solve the GA problem.

Definition 3.3 : A graph G(V,E) with n vertices and m edges consists of a vertex set V(G)={V_1,…,V_n} and an edge set E(G)={E_1,…,E_m}. Each edge consists of two vertices called its endpoints. (U,V) is an edge with endpoints U and V. A graph is undirected if there is no ”direction” on the edges. A graph is weighted if there are positive integer weights on the edges. The weight of the edge (U,V) is denoted as W((U,V)).

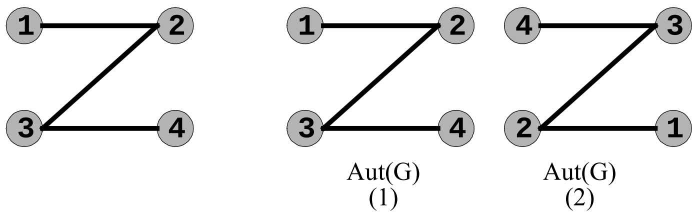


Figure 3.3. An example of graph G and its automorphisms.

Definition 3.4 : An automorphism of graph G is a permutation of V(G) that preserves adjacency [20]. The automorphisms of G is denoted as Aut(G), and the cardinality of this set of automorphisms is denoted as |Aut(G)|.

Example 3.3: An undirected graph G = (V, E) as shown in Fig. 3.3 has V = {1,2,3,4} and E = {(1,2), (2,3), (3,4)}. Its automorphisms are also shown in Fig. 3.3 which are {1234 ,4321}, and |Aut(G)| = 2.

Definition 3.5 : A graph G is disconnected if we can partition its vertices into at least two nonempty sets, R and S, such that no vertex in R is adjacent to any vertex in S. That is, G is the disjoint union of the two subgraphs induced by R and S.

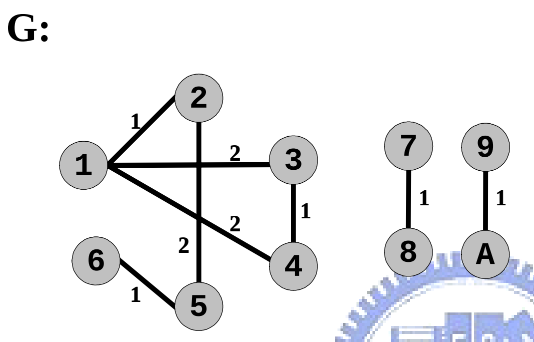
Definition 3.6 : For any two sets π_1 and π_2, its intersection $\pi_1 \cap \pi_2$ is a set that contains elements in both sets.

We propose an automorphism representation that is similar to the UPS representation mentioned in Chapter 2.2 to record the automotphisms of a graph G. Vertices in the same group in this automorphism representation imply that any permutation of these vertices is an automorphism.

Example 3.4: For an automorphism representation π_1 = (12)(34). It contains 2! × 2! automorphisms and they are {1234, 1243, 2134, 2143}. If π_2 = (2)(134), it contains 1! ×

3! automorphisms and they are {1234, 1243, 3214, 3241, 4213, 4231}, then $\pi_1 \cap \pi_2$ = {1234, 1243}, which can be expressed as (1)(2)(34) in automorphism representation.

Now, there are four steps in finding Aut(G) in our algorithm. Given an undirected graph G with N vertices, the initial automorphism representation is expressed as (123…N).



$$\mathrm{Adj(G)} = \begin{bmatrix} 0111000000 \\ 1000100000 \\ 0001000000 \\ 1010000000 \\ 0100000000 \\ 0000100000 \\ 0000000100 \\ 0000001000 \\ 0000000001 \\ 0000000010 \end{bmatrix}$$

(a) (b)

Figure 3.4. An undirected graph G and its Adj(G).

Step 1(Disjoint Graph - DG): If the graph is composed by t disconnected subgraphs, then the automorphism representation is divided into t groups such that each group corresponds to one subgraph. If the graph exists two or more subgraphs with the same number of vertices, then merge the corresponding subgroups into one.

Since swapping any two vertices coming from disconnected subgroups with different size cannot be an automorphism, we can divide the automorphism representation of G respect to the graph connectivity of different sizes directly.

Example 3.5: Given a graph G as shown in Fig. 3.4(a). The graph G can be divided into three subgraphs. Since the subgraphs induced by vertices 7 and 8, and by vertices 9 and 10 are the same size, therefore, by Step 1, the updated automorphism representation π_1= (123456)(789A). That means any vertex exchanged from these two groups cannot be an automorphism.

Definition 3.7 : The Degree Vector (DV) of a graph is a vector that contains each vertex's degree such that DV[i] = degree of the ith vertex.

Step 2(Degree Vector - DV): Calculate the DV of the graph G. Then group the vertices with the same degrees into one group.

Since exchanging of two vertices with different degrees cannot be an automorphism, we calculate the degree of each vertex and group the vertices with the same degrees into the same group. The grouping results are the automorphisms and are complied with the proposed automorphism representation.

An automorphism has to satisfy the properties described in Step 1 and Step 2, so the updated automorphisms can be obtained from the intersection of automorphisms derived by Step 1 and Step 2.

Example 3.6: From Fig. 3.4(a), the DV of G is 5 3 3 3 3 1 1 1 1 1. By grouping the degree of each vertex, the automorphism representation is π_2 = (1)(2345)(6789A). Thus, the updated automorphism representation by the intersection of Step 1 and Step 2 is π_3 = (1)(2345)(6)(789A) which comes from (123456)(789A)∩(1)(2345)(6789A).

Definition 3.8 : The partial vector (PV) of vertex V_i in G is the ith row of adjacency matrix of G, Adj(G).

Definition 3.9 : A single element group (SEG) is a group that contains only one vertex in the automorphism representation.

If updated automorphism representation contains SEG, go to Step 3, otherwise go to Step 4.

Step 3(Repeated Automorphism - RA): Grouping the partial vector of each SEG vertex and intersect this grouping result with the updated automorphism representation. If the updated automorphism representation has newly generated SEG, repeat Step 3, otherwise go to Step 4.

Since the automorphisms do not relate to the vertex in SEG, partition the partial vector of SEG vertices can determine the automorphisms. That is, only the neighbors with the same degree to the SEG vertices can be swapped as automorphisms.

Example 3.7: In the updated automorphism representation π_3(1)(2345)(6)(789A), there are two vertices, 1 and 6, in SEG, respectively. We have known that in Fig. 3.4(a), vertex 1 is connected to vertices 2, 3 and 4, but is not connected to vertex 5. Thus, vertex 5 is different to vertices 2, 3 and 4. Besides, the weight of (1,2) is different to the weight of (1,3) and (1,4). We can use partial vector V_1 to reduce (2345) to (2)(34)(5). So as targeting on vertex 1, the partial vector V_1 is 0122000000. It can be seen from the 1st row of Adj(G). Its corresponding V_1_g, is (156789A)(2)(34). Then $\pi_3 \cap V_1$_g = (1)(2)(34)(5)(6)(789A) = π_4. We find that vertices 2 and 5 are new vertices in SEG. Therefore, we conduct Step 3 for vertices 2, 5 and 6. As targeting on vertex 6, the partial vector V_6 is 0000100000, V_6_g is (12346789A)(5). $\pi_4 \cap V_6$_g = (1)(2)(34)(5)(6)(789A) = π_5. As targeting on vertex 5, the partial vector V_5 is 0200010000, V_5_g is (1345789A)(2)(6). $\pi_5 \cap V_5$_g = (1)(2)(34)(5)(6)(789A) = π_6. As targeting on vertex 2, the partial vectors V_2 is 1000200000, V_2_g is (1)(5)(2346789A), $\pi_6 \cap V_2$_g = (1)(2)(34)(5)(6)(789A) = π_7. Now, no more new vertex in SEG is generated, then go to Step 4.

Step 4: For the group with more than one element in it under automorphism representation, we observe its corresponding subgraph. If the subgraph is disconnected, give the joinder "——— "on those vertices which are connected, respectively. If the subgraph is a cycle, then give the " [ ] " on these vertices. If the subgraph is a complete graph, then give the " { } " on these vertices. Observe those vertices in some subgraph under automorphism representation with just two elements in it. Record the partial vector if it really can reduce automorphism representation.

After Step 1 ~ Step 3, we can assure that if more than one vertex in a group, they must have the same degrees and may be the combination of smaller subgraphs with the same number of vertices. Their corresponding subgraphs have three conditions. One is that it is just the combination of smaller subgraphs with same vertices, e.g. G1={(1,2), (3,4)} shown in Fig. 3.5(a). The number of |Aut(G1)| has special solution for it. We will introduce this in the next paragraph. Another condition is that it is a connected subgraph and is also a complete subgraph, e.g. G2={(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)} shown in Fig. 3.5(b). So we can change the order of these vertices in the subgraph and remain identical under automorphism representation. The other condition is that it is a connected graph and it is a cycle shown, e.g. G3={(1,2),(1,3),(2,4),(3,4)} shown in Fig. 3.5(c). The number of |Aut(G3)| has special solution for it. We will also introduce this in the next paragraph. And if there are just two vertices in a subgraph, we have to observe the neighbors of these two vertices. If they have no neighbors in common, then the partial vectors of these two vertices are useful. Otherwise, the partial vectors of these two vertices are useless because these two vertices can be exchanged mutually.

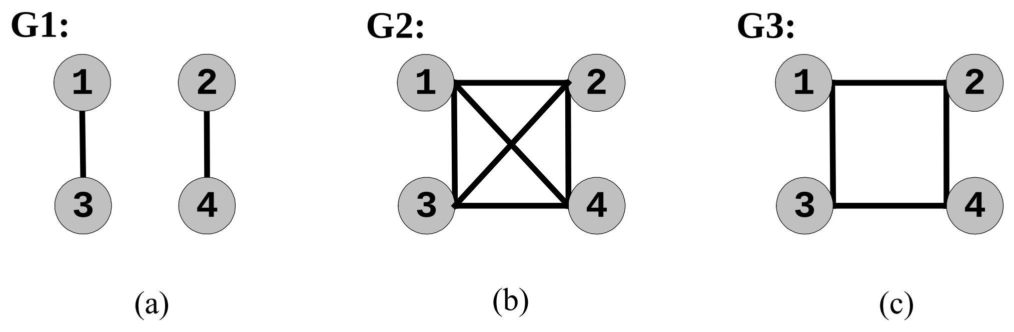


Figure 3.5. Disjoint graph, complete graph and cycle graph.

Example 3.8: For the updated automorphism representation π_7 (1)(2)(34)(5)(6)(789A), there are six groups. Only the 6th group (789A) corresponds to the disconnected subgraph. Observe the vertices 7, 8, 9, and A, vertex 7 is connected with 8 while vertex 9 is connected with A. The group (789A) under automorphism representation now becomes $(\overline{789A})$. Thus, the updated automorphism representation π_7 is now (1)(2)(3 4)(5)(6) $(\overline{789A})$.

After Step 1 ~ Step 3, we can find the Aut(G) and |Aut(G) | can be obtained based on the updated automorphism representation. For updated automorphism representation $(a_1a_2\cdots a_{n_k})(b_1b_2\cdots b_{n_k})\cdots(r_1r_2\cdots r_{n_k})$, the |Aut(G)| = ($n_1$!) × ($n_2$!) × ··· × ($n_k$!). If the joinders are shown in the automorphism representation, such as $(\overline{a_1a_2\cdots a_{n_1}}\ \overline{b_1b_2\cdots b_{n_1}}\cdots\overline{r_1r_2\cdots r_{n_1}})$, |Aut(G)| = {($n_1$!) × ($n_1$!) × ··· × ($n_1$!) × (the number of joinder)!}. For example in Fig. 3.5(a), Aut(G1)={1234, 1243, 2134, 2143, 3412, 4312, 3421, 3421}. |Aut(G1)| = 2! × 2! × 2! = 8. If the braces are shown in the automorphism representation, such as ($\{a_1a_2\cdots a_{n_1}\}$), |Aut(G)| = n_1!. For example in Fig. 3.5(b), Aut(G2)={1234, 1243, 1324, 1342, 1423, 1432, 2134, 2143, 2314, 2341, 2413, 2431, 3124, 3142, 3214, 3241, 3412, 3421, 4123, 4132, 4213, 4231, 4312, 4321}.

C_2^n

|Aut(G2)| = 4! = 32. If the square brackets are shown in the automorphism representation, such as ($[a_1 a_2 \cdots a_{n_1}]$), |Aut(G)| = (n_1) × 2. For example in Fig. 3.5(c), Aut(G3)={1234, 1324, 2143, 2413, 3142, 3412, 4231, 4321}. |Aut(G3)| = 4 × 2 = 8. For this example, the updated automorphism representation π_7 is (1)(2)(34)(5)(6) ($\overline{78}\,\overline{9A}$). Thus, |Aut(G)| = 1! × 1! × 2! × 1! × 1! × (2! × 2! × 2!) = 16. We can enumerate these 16 automorphisms explicitly and they are {123456789A, 12345678A9, 123456879A, 12345687A9, 1234569A78, 123456A978, 1234569A87, 123456A987, 12435678 9A, 12435678A9, 124356879A, 12435687A9, 1243569A78, 124356A978, 1243569A87, 124356A987}.

3.3. UPSs Calculation with Automorphism Technique

From Chapter 3.1, we get the verification pattern sets. Now, we conduct the UPSs_Calculation stage with the graph automorohism (GA) technique to determine the remaining UPSs.

For the example in Fig. 3.6(a), we assume the verification pattern set S1 has four patterns P_1, P_2, P_3 and P_4.

To solve the problem of calculating the remaining UPSs of the pattern set S1 with n bits, an undirected, weighted graph G(V,E) is constructed, which corresponds to the set S1 with |S1| patterns, P_1 to $P_{|S1|}$. $P_i[j]$ in S1 denotes the j^{th} bit in P_i where i = 1 ~ |S1| and j = 1 ~ n. The vertex V*k* in G corresponds to the *k*th input variable/port in S1. For all patterns P_1 to $P_{|S1|}$ in S1, when P_i [*k*] = P_i [*k'*] = 1, an edge V*k*V*k'* is added into G and W(V*k*V*k'*)=1 where (*k*, *k'*) are all C_2^n bit pairs. If the edge V*k*V*k'* has existed in G, W(V*k*V*k’*) is increased by one.

Example 3.9: For $P_1[1:7]$ in S1 shown in Fig. 3.6(a), 1010001, since $P_1[1] = P_1[3] = P_1[7] = 1$, edges V1V3, V1V7, and V3V7 are added into G, respectively. For $P_2[1:7]$, 0100110, edges V2V5, V2V6, and V5V6 are added into G, respectively, and so on. The constructed graph G is an undirected weighted graph and is shown in Fig. 3.6(b). Its corresponding adjacency matrix, Adj(G), is shown in Fig. 3.6(c). In Adj(G), if there is no edge between V*k* and V*k’* in G, Adj(G)[*k*][*k’*] = Adj(G)[*k’*][*k*] = 0; otherwise Adj(G)[*k*][*k’*] = Adj(G)[*k’*][*k*] = W(V*k*V*k’*).

The problem of calculating the remaining UPSs in S1 is now transformed to finding all automorphisms of G. The effectiveness of this problem transformation is that the position relations of digit 1s in each pattern in S1 are transformed to the connectivity relations in G. Finding the port misplacements that maintain S1 to be invariant (calculating UPSs) is equivalent to finding all automorphisms of G.

Verification pattern set S1

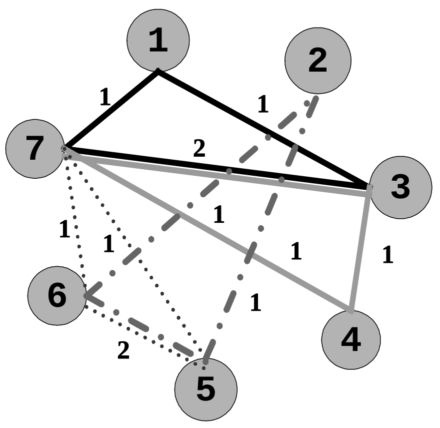
(1 2 3 4 5 6 7)

$P_1[1:7]$=1 0 1 0 0 0 1

$P_2[1:7]$=0 1 0 0 1 1 0

$P_3[1:7]$=0 0 1 1 0 0 1

$P_4[1:7]$=0 0 0 1 1 1 1



$$\mathrm{Adj(G)} = \begin{bmatrix} 0010001 \\ 0000110 \\ 1001002 \\ 0010001 \\ 0100021 \\ 0100201 \\ 1021110 \end{bmatrix}$$

(a) **(b)** **(c)**

Figure 3.6. UPSs_Calculation with GA technique.

We demonstrate the UPSs_Calculation stage by the proposed GA algorithm using the following example.

The initial UPSs is π_0 = (1234567). In Step1, the graph in Fig. 3.6(b) is a connected graph and cannot be divided into subgraphs. Thus, the UPS remains unchanged. In Step 2, the degree vector DV[1:7] is 2 2 4 2 4 4 6. It groups the three 2s in one group, three 4s in another group, and one 6 in the third group. The grouped results can be represented as (222)(444)(6) and its corresponding UPSs are (124)(356)(7). Then the updated UPS $\pi_1 = \pi_0 \cap$(124)(356)(7) = (1234567)$\cap$(124)(356)(7) = (124)(356)(7). In Step 3, there is one SEG vertex 7. As targeting on vertex 7 in updated UPSs π_1, the partial vector V_7 is 1021110, V_7_g, (1456)(27)(3). Then $\pi_2 = \pi_1 \cap V_7$_g =(14)(2)(3)(56)(7). SEGs of vertices 2 and 3 are newly generated. Therefore, we have to repeat Step 3 for vertices 2 and 3, respectively. As targeting on vertex 2, the partial vector V_2 is 0000110, V_2_g is (12347)(56). $\pi_3 = \pi_2 \cap V_2$_g = (14)(2)(3)(56)(7). As targeting on vertex 3, the partial vector V_3 is 1001002, V_3_g is (14)(2356)(7). $\pi_4 = \pi_3 \cap V_3$_g = (14)(2)(3)(56)(7). Now, no new vertex in SEG is generated, Step 3 is terminated. For the updated UPS π_4 (14)(2)(3)(56)(7), there are five groups. The partial vector of the first group and the 4th group is useless because the neighbors of each group are the same. Thus, the updated UPS π_4 remains unchanged. The GA technique is now finished. The summary of this example is shown in Fig. 3.7.

	Initial UPSs (1 2 3 4 5 6 7)
	⇩
Step (1) DG : (1 2 3 4 5 6 7)	(1 2 3 4 5 6 7)
	⇩
Step (2) DV : (1 2 4)(3 5 6)(7)	(1 2 4)(3 5 6)(7)
	new isolated vertex (7)
Step (3) RA:	⇩
Target on 7	
PV : (1 4 5 6)(2 7)(3)	(1 4)(2)(3)(5 6)(7)
	new isolated vertices (2)(3)
Target on 2	⇩
PV : (1 2 3 4 7)(5 6)	(1 4)(2)(3)(5 6)(7)
Target on 3	⇩
PV : (1 4)(2 3 5 6)(7)	(1 4)(2)(3)(5 6)(7)

Figure 3.7. Graph Automorphism (GA) technique.

In the same example, the remaining UPSs obtained from the CV approach are (124)(356)(7) and from the SAA approach are (124)(3)(56)(7) in [19]. However, it can be further reduced to (14)(2)(3)(56)(7) by the GA approach and the real remaining UPSs are (14)(2)(3)(56)(7). These results demonstrate that the GA approach gets more precise remaining UPSs than that of the CV approach and the SAA approach.

In this example in Fig. 3.7, Step 4 seems useless for reducing UPSs. However, it also provides information for reducing UPSs later on. Hence we do not conduct Step 4 actually in the AVPG for preserving the UPS representation concisely. But its impact on reducing UPSs is still remained by using partial vector concept.

Definition 3.10: The partial vector of vertex t is called automorphism message of t, denoted as AM_t, if vertex t does not belong to SEG in the UPS representation.

Giving UPSs (1234)(56)(78)(9A)(BC), assume that the valid verification pattern set is {10010101010, 010010100101, 001001011010, 000101010101}, as shown in Fig.

3.8(a) and its adjacency matrix is shown in Fig. 3.8(b). If we apply the GA technique of Step 1 ~ Step 3, we can find that the updated UPS is still (1234)(56)(78)(9A)(BC). Then we keep this verification pattern set and calculate AM for each vertex t. For example, AM_1 defined as the partial vector of vertex 1 is 000010101010, AM_C defined as the partial vector of vertex C is 010111110200 and they are shown in Fig. 3.8(c). AM_1 implies that if vertex 1 is in SEG after some iteration later, the vertices 5, 7, 9, 11 (vertices with assignment 1 in AM_1) can also be in SEGs as well. Thus if we assume vertex 1 in SEG indeed in the next UPSs_Calculation process, then we can reduce UPSs from (1234)(56)(78)(9A)(BC) to (1) (234)(5)(6)(7)(8)(9)(A)(B)(C) according to AM_1. Other AMs have the similar effect on reducing UPSs.

Verification pattern set

(1234)	(56)	(78)	(9A)	(BC)
1000	10	10	10	10
0100	10	10	01	01
0010	01	01	10	10
0001	01	01	01	01

(a)

AM_1: 000010101010

AM_C: 010111110200

(c)

$$\mathrm{Adj}(G) = \begin{bmatrix} 000010101010 \\ 000010100101 \\ 000001011010 \\ 000001010101 \\ 110000201111 \\ 001100021111 \\ 110020001111 \\ 001102001111 \\ 101011110020 \\ 010111110002 \\ 101011112000 \\ 010111110200 \end{bmatrix}$$

(b)

Figure 3.8. Verification pattern set with automorphism message (AM).

Now, the complete AVPG Flow with graph automorphism (GA) technique is shown in Fig. 3.9, where DG means Disjoint Graph in Step 1, DV means Degree Vector in Step 2, RA means the Repeated Automorphism in Step 3, and AM means Automorphism Message in Step 4. The pseudo code of AVPG with GA technique is shown in Fig. 3.10. In line 7, 9, it generates patterns and the output simulation in line 11. The effectiveness of the simulated patterns is determined in line 12. The steps of GA technique are in line 14 ~ 16, 20, 23, respectively. In line 17, 21, it includes the valid patterns into the verification pattern set. At the end of the algorithm, the verification pattern set, error coverage and UPSs are returned.

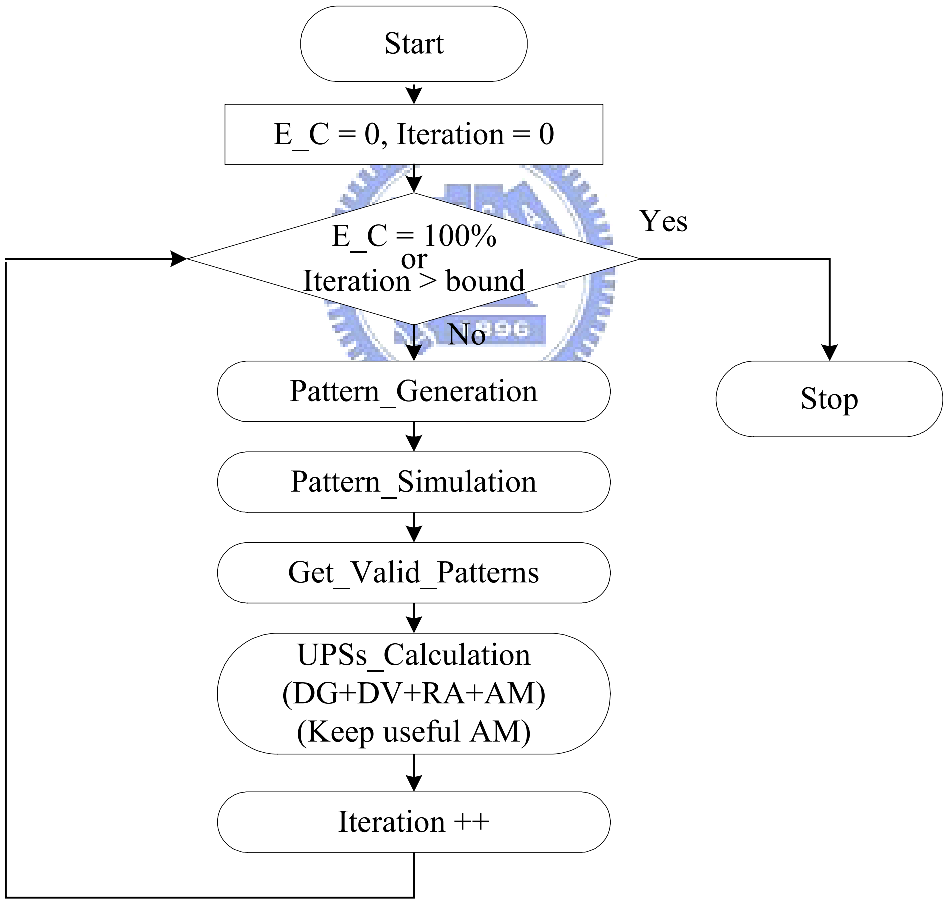


Figure 3.9. Complete AVPG flow.

```
Algorithm: Automatic Verification Pattern Generation with Automorphism Technique
Input: CUV ( N-input, M-outtput )
Output: Verification Pattern set (Veri_P_S) , error coverage (E_C) , UPS
{
01      variable i          ←  0 ;
02      UPS                 ←  (1 2 3 ... N) ;
03      E_C                 ←  0 ;
04      simulated[N-1]      ←  {0,0, ... ,0} ;
05      Veri_P_S            ←  {$\phi$} ;
06      Temp_P_S            ←  {$\phi$} ;

        do
        {
            if (i==0)
                {
07                  verification patterns ← $C_1^N , C_{N-1}^N$ ;
08                   i ← i+1;
                }
            else
09              verification patterns ← Pattern_generation (UPS , simulated[N-1]) ;

10          Update (simulated[N-1]);
11          outputs ← Simulation (verification patterns);
12          {valid verification patterns} ←output_analysis(outputs);
13          G(V,E) ← Transition ({valid verification patterns }) ;
14          {UPS_tmp} ←DG ( G , UPS) ;
15          {UPS_tmp} ←DV ( G , UPS_tmp) ;
16          {UPS_tmp} ←RA ( G , UPS_tmp) ;

            if ( UPS_tmp $\neq$ UPS )
                {
17                  Veri_P_S ← Veri_P_S $\cup$ {valid verification patterns } ;
18                  E_C ← E_C_calculation (UPS_tmp) ;
                }

19          UPS ← UPS_tmp ;
20          UPS_tmp2 ← Use_AM (Temp_P_S,UPS) ;

            if ( UPS_tmp2 $\neq$ UPS )
                {
21                  Veri_P_S ← Veri_P_S $\cup$ Temp_P_S ;
22                  UPS ← UPS_tmp2 ;
                }

23          {Temp_P_S} ← Keep_AM (G,UPS);

            if (E_C == 1) break ;
            if (simulated[N-1]=={1,1, ... ,1}) break ;
        }
        while (E_C !=1 ) ;
24      return (Veri_P_S , E_C , UPS) ;
}
```

Figure 3.10. Pseudo code of AVPG with GA technique.

Chapter 4

Experimental Results

The GA-based algorithm is implemented in Programming Language Interface (PLI) environment. Experiments are conducted over a set of ISCAS-85 and some MCNC benchmarks with large number of inputs. The benchmarks are in Verilog HDL format. Since only the simulation information of these benchmarks is needed to conduct the experiments, arbitrary levels of design description can be used for generating verification pattern set. Table 4.1 summarizes the experimental results of the comparison of SAA approach in [19] and our graph automorphism technique algorithm of AVPG. The first five columns show the parameters of each benchmark, including name, |PI|, |PO|, the number of literals (lits.) and the number of POEs. The |PI|

represents the number of inputs and the size of the POEs set is |PI|!-1. The |PO| represents the number of outputs and influences on the probability of error effect propagation. The number of literals indicates the complexity of a benchmark. The remaining columns show the number of verification patterns (pats.), error coverage (E_C) and the CPU time. The error coverage is defined as 1- (# of undetected POEs / # of all POEs). The ratio is defined as (the patterns of GA approach / the patterns of SAA approach). The iteration bound is set to 100. The CPU time is measured in second on a Sun Sparc II Workstation. The algorithm will be terminated automatically if iterations are over the bound or the error coverage reaches 100%, and the verification pattern set and the error coverage are returned. For example, in c880 benchmark, the number of verification patterns is decreased from 130 to 73 and the ratio of pattern size is 0.56. We can see that for most circuits, the number of verification patterns is reduced by the proposed techniques – a new pattern generation algorithm and the GA technique. Furthermore, the CPU time is acceptable.

Table 4.1. Comparisons on experimental results of SAA approach and GA approach.

Bench	Parameters			SAA approach		GA approach			
	|PI|	|PO|	Lits.	Pats.	E_C(%)	Pats.	E_C(%)	Time(s)	Ratio
c17	5	2	12	5	100	4	100	0.23	0.80
c880	60	26	703	130	99.999	73	99.999	5.59	0.56
c1355	41	32	1032	51	100	39	100	0.91	0.74
c1908	33	25	1497	45	100	41	100	0.93	0.91
c432	36	7	382	35	100	35	100	0.41	1
c499	41	32	616	40	100	37	100	0.72	0.93
c3540	50	22	2934	89	100	70	100	10.70	0.75
c5315	178	123	4369	222	100	174	100	53.42	0.78
c2670	233	140	2043	351	99.999	237	99.999	92.71	0.67
c7552	207	108	6098	448	99.999	312	99.999	87.45	0.69
c6288	32	32	4800	30	99.999	30	99.999	2.84	1
des	256	245	7412	255	100	255	100	5.47	1
alu4	14	8	1278	17	100	14	100	0.51	0.82
apex6	135	99	904	187	99.999	154	99.999	12.88	0.82
i9	88	63	1453	107	100	92	100	1.12	0.86
i8	133	81	4626	204	100	156	100	13.65	0.76
i7	199	67	1311	240	100	194	100	12.77	0.81
i6	138	67	1037	138	100	130	100	10.12	0.94
i5	133	66	556	133	100	125	100	10.65	0.94
rot	135	107	1424	247	99.999	150	99.999	17.40	0.61
x3	135	99	1816	165	99.999	145	99.999	9.34	0.88
x4	94	71	1040	141	99.999	117	99.999	5.14	0.83
pair	173	137	2667	186	100	120	100	21.38	0.65
Total				3466		2704			
Ratio				1		0.78			

Chapter 5

Applications

We use this graph automorphism-based algorithm for computing maximal sets of symmetric inputs of circuits. It can be used to identify nonsymmetric inputs in a circuit and enhance the efficiency of input matching, library binding (technology mapping), as well as logic verification problems.

5.1. About Symmetric

Two inputs x_i an x_j of a single/multi-output logic function f(X) are said to be symmetric, if exchanging x_i and x_j does not change f, i.e., $f(x_1\ldots, x_i,\ldots, x_j,\ldots x_n) = f(x_1,\ldots, x_j,\ldots, x_i,\ldots, x_n)$[34].

Symmetric input sets of a logic function are subsets of inputs such that any permutation of the inputs within a subset leaves the function invariant [33, 34]. The problem of finding maximal symmetric input sets is formulated as following [33, 34]: Given a single/multi-output function f(X), find maximal subsets of inputs $X_1, X_2, \ldots, X_k \subseteq X$, such that $X_1 \cup X_2 \cup \ldots \cup X_k = X$ and the inputs in every subset X_i can be permuted in any fashion without changing the function. The symmetry relation is an equivalence relation, and the maximal symmetric input sets are its equivalence classes. To find the maximal symmetric input sets, it is therefore sufficient to find all pairs of symmetric inputs and then take the unions of all the pairs having nonempty intersection.

This symmetry is known as the classical symmetry or the nonskew nonequivalence symmetry [35]. The definition of this symmetry translates into the following requirements for the cofactors of function $f_{Xi\overline{Xj}} = f_{\overline{Xi}Xj}$ with respect to any two inputs x_i and x_j.

Symmetric input sets, and the particular maximal symmetric input sets, are important in problems of design verification and diagnosis [12, 21, 23, 25-27, 29-32, 40-44] and in problems of technology mapping, where it is required to find a library macro to implement a given logic block [24, 28, 36, 39].

The goal of design verification is to check whether a given implementation follows the specification for which it was designed. If implementation is incorrect, diagnosis may be used for gate-level implementations [12, 25, 29-31, 40, 41] to locate and then correct the design. It is typically assumed that the correspondence between specification and implementation inputs is known. However, due to design errors, implementation inputs may be interchanged, and hence it may be necessary to compute the actual correspondence between specification and implementation inputs. This problem was studied in [12], where it was observed that finding the symmetric input sets can save the effort of trying different permutations of inputs belonging to the same symmetric set, since all these permutations are equivalent.

In technology mapping applications [24, 28, 36, 39], it is required to check whether a logic block in the given circuit exists in a macro library. The problem that arises is to find a correspondence between the inputs of the logic block and the inputs of a library macro and a correspondence between the outputs of the logic block and the outputs of the library macro, such that under the two correspondences the library macro is logically equivalent to the logic block. The library macro can then be used to implement the logic block. For this problem, too, finding the symmetric input sets of the logic block and the library macro can save the effort of trying different permutations of inputs belonging to the same symmetric set.

The problem of finding maximal subsets of symmetric inputs has been studied for single-output functions, for which the set of all minterms that set the function to 1 is available (or can be derived) [33]. An efficient method to find subsets of symmetric inputs, applicable to large multi-output circuits, is studied here. In parallel to our work on this problem [35, 37, 38], methods using BDD's [22] are investigated in [35, 37]. However, these BDD-based algorithms are not applicable to the designs that described in behavior level or RT-level, e.g., soft Intellectual Property, or that do not have compact BDD representation. Thus, simulation-based approach [38] was proposed to compute the maximal sets of symmetric inputs. [38] establishes two steps to accomplish the input symmetric identification. The first step uses heuristic to identify inputs that do not belong to the same symmetric input set. Although the signature-based heuristic used in [24, 36, 39] can be used for this purpose, the heuristics in [24, 36, 39] were applied to circuits having compact BDD representation. The second step uses test generation techniques to further identify the inputs that were not distinguished by the heuristic. The efficiency of [38]-like approach for computing the maximal sets of symmetric inputs in circuits without compact BDD representation depends on the "ability" of heuristics. Good heuristics can distinguish more nonsymmetric inputs prior to entering computation intensive test generation step. Thus, we focus on the first step of [38]-like approach by using graph automorphism-based heuristic.

5.2. Symmetric with GA Algorithm

We exchange the UPS representation to the Symmetric-ASymmetric Inputs (SASIs) to represent the maximal symmetric inputs sets. For an N-input circuit, we number its inputs from 1 to N. Initially, we assume that all inputs are symmetric, the corresponding SASIs representation is (1 2 3 … N), i.e., all inputs are placed in one group. If we claim that input *i* is asymmetric to the other inputs by our methods, the input *i* is isolated from original group and can be expressed as (*i*)(1 2 … *i*-1 *i*+1 … N). Please note the order of the groups in the SASIs representation is irrelevant, neither is the order of the number in each SASIs group. By the SASIs representation, if any two inputs are not placed in the same group, then they are nonsymmetric inputs. Otherwise they are "possibly" symmetric. Obviously, according to SASIs representation, we can realize exactly which inputs are asymmetric to the other inputs. Thus, further efforts ought to be put onto the groups that contain undistinguished inputs. The following example demonstrates the details of the SASIs representation.

Given an 8-input circuit, the inputs are numbered from 1 to 8. The SASIs representation (12345678) represents all inputs are symmetric. The SASIs representation (125)(4)(3678) indicates that inputs 1, 2 and 5 are symmetric and inputs 3, 6, 7 and 8 are symmetric. Input 4 is the only one element in the second group. It means that input 4 is asymmetric to the other inputs. The SASIs (125)(4)(3678) can also be expressed as (4)(215)(8763). The SASIs representation (1)(2)(3)(4)(5)(6)(7)(8) has eight groups and each group has only one element; therefore, these inputs are nonsymmetric inputs. The goal of the proposed heuristic is to induce SASIs representation from (12345678) to (1)(2)(3)(4)(5)(6)(7)(8) if these inputs are asymmetric indeed.

Then, we change the AVPG to Automatic Computing Symmetric Procedure (ACSP). Its flow chart is shown in Fig. 5.1. This flow chart is similar to the AVPG flow shown in Fig. 3.8. The ACSP reads a combinational circuit and generates heuristic patterns. The patterns simulation results provide information to SASIs_Calculation stage for figuring out nonsymmetric inputs in SASIs representation. Then further heuristic patterns are generated according to the updated SASIs in the next iteration. When all inputs are identified as nonsymmetric or the iterations are over the bound, the ACSP will be terminated and the maximal symmetric input sets are returned. The pseudo code of ACSP with GA technique is shown in Fig. 5.2. In line 5, 7, it generates patterns and the output simulation in line 9. The valid generated pattern sets are determined in line 10. The steps of GA technique are in line 11 ~ 17. At the end of the algorithm, the SASIs are returned.

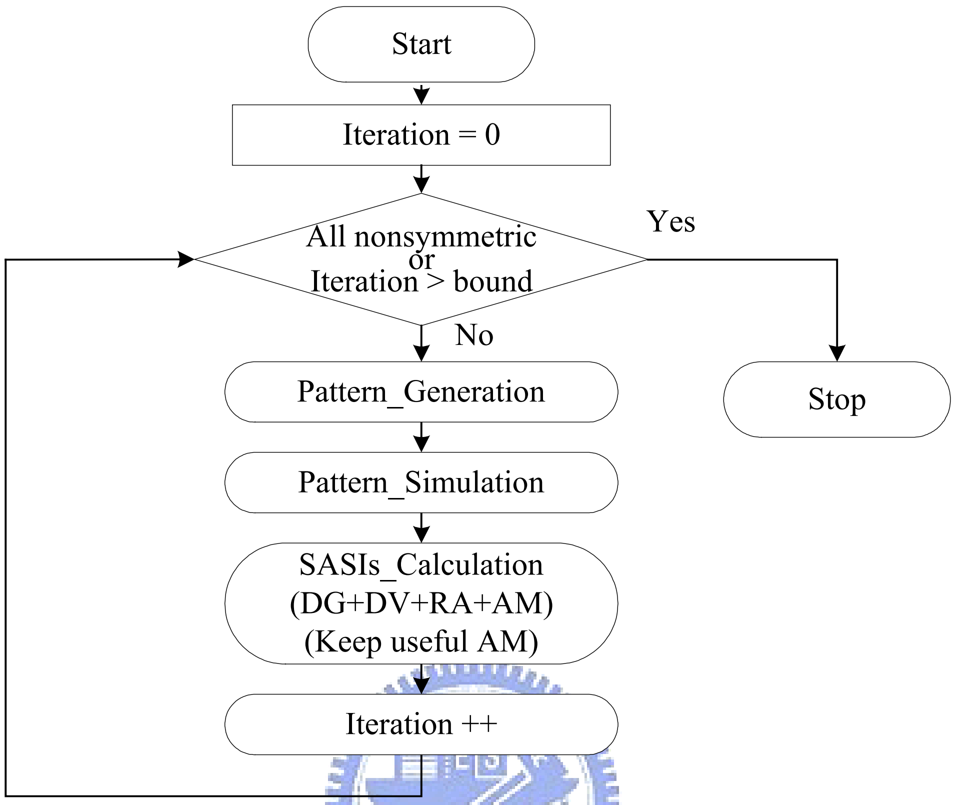


Figure 5.1. Complete ACSP flow.

```
Algorithm: Automatic Computing Symmetric Procedure with GA Technique
Input: circuit ( N-input, M-output )
Output:  SASI
{
01      variable i          ← 0 ;
02      SASI                ← (1 2 3 ... N) ;
03      simulated[N-1]      ← {0, 0, ... , 0} ;
04      Intending_AM        ← {$\phi$} ;
        do
        {
            if (i==0)
                {
05                  generated pattern sets ← $C_1^N, C_{N-1}^N$ ;
06                  i ← i+1;
                }
            else
07              generated pattern sets ← Pattern_Generation (SASI , simulated[N-1]) ;

08            Update (simulated[N-1]) ;
09              outputs      ← Simulation (generated pattern sets) ;
10            {valid generated pattern sets} ← outputs_analysis (outputs) ;
11              G(V,E)       ← Transformation ({valid generated pattern sets}) ;
12            {SASI_tmp} ← DG ( G, SASI) ;
13            {SASI_tmp} ← DV ( G, SASI_tmp) ;
14            {SASI_tmp} ← RA ( G, SASI_tmp) ;
15            {SASI_tmp} ← Use_AM (Intending_AM , SASI_tmp) ;
16                SASI     ← SASI_tmp ;
17          {Intending_AM} ← Keep_AM (G , SASI);

            if (simulated[N-1]=={1,1,...,1}) break ;
        }
        while (all nonsymmetric) ;
18      return (SASI) ;
}
```

Figure 5.2. Pseudo code of ACSP with GA technique.

5.3. Experimental Results

The GA-based algorithm is implemented in Programming Language Interface (PLI) environment. Experiments are conducted over a set of ISCAS-85 and some MCNC benchmarks. Table 5.1 summarizes the experimental results of the previous approach in [38] and our GA-based algorithm. The first four columns show the parameters of each benchmark, including name, |PI|, |PO| and the number of literals (lits.). The |PI|

represents the number of inputs. The |PO| represents the number of outputs. The number of literals indicates the complexity of a benchmark. This number is derived from its gate level description. The remaining columns show the sets of inputs that cannot be distinguished. In fact, these input sets are possibly symmetric inputs sets. They are expressed by pairs (*size*, number of sets), where *size* is the size of an input set and the following is the number of sets of that size. For example, for c880, three sets of size two means that they are possible symmetric inputs and the other sets of size one means that the 54 inputs are asymmetric to other inputs. The iteration bound is set to 100. The CPU time is measured in second on a SUN Sparc II workstation. The algorithm will be terminated automatically if iterations are over the bound or all inputs are nonsymmetric, and SASIs are returned. According to Table 5.1, we can find that in previous approach, c499, c1355, and c1908 still have potentially symmetric input sets, but our approach can distinguish each input as a nonsymmetric input. Also for c2670 and c6288, our approach distinguishes more nonsymmetric inputs than that of [38]. Please note for c6288, our approach does not distinguish each input as a nonsymmetric input while [38] does. This is because our heuristic patterns inherently cannot distinguish each input of a multiplier such as c6288. [38] uses test generation technique to conquer it instead. Table 5.2 also shows the results on some MCNC benchmarks. Most nonsymmetric inputs are distinguished efficiently as usual for most benchmarks. These results demonstrate that our approach acts as a good filter to identify nonsymmetric inputs as many as possible in a circuit such that significant efforts can be saved for other succeeding applications. Our approach is applicable to sequential circuits if circuit states can be fully specified. The experiments on sequential circuits are progressing.

Table 5.1. Comparisons on experimental results of previous approach in [38] and GA approach.

Bench	Parameters			Previous approach in [38]		GA approach	
	|PI|	|PO|	Lits.	(size, number of sets)	Time (s)	(size, number of sets)	Time (s)
c17	5	2	12	(1,5)	0.04	(1,5)	0.23
c880	60	26	703	(1,54)(2,3)	2.86	(1,54)(2,3)	1.74
c1355	41	32	1032	(1,32)(9,1)	3.16	(1,41)	0.91
c1908	33	25	1497	(1,22)(5,1)(6,1)	1.81	(1,33)	0.93
c432	36	7	382	(1,36)	0.24	(1,36)	0.41
c499	41	32	616	(1,32)(9,1)	1.07	(1,41)	0.72
c3540	50	22	2934	(1,50)	19.52	(1,50)	10.70
c5315	178	123	4369	(1,178)	33.95	(1,178)	33.42
c2670	233	140	2043	(1,221)(2,2)(8,1)	59.95	(1,223)(2,2)(6,1)	42.55
c7552	207	108	6098	(1,166)(2,6)(3,1)(5,2)(4,4)	5514	(1,183)(2,8)(3,1)(5,1)	191.33
c6288	32	32	4800	(1,32)	6.53	(2,16)	2.84

Table 5.2. Results of some MCNC benchmarks.

Circuit	|PI|	|PO|	Lits.	(size, number of sets)	Time(s)
9symml	9	1	277	(9,1)	0.95
b1	3	4	17	(1,1)(2,1)	0.24
b9	41	21	236	(1,31)(2,5)	3.88
cm138a	6	8	35	(1,4)(2,1)	0.29
cm162a	14	5	58	(1,12)(2,1)	0.37
cm163a	16	5	53	(12,1)(4,1)	0.36
cm82a	5	3	26	(1,3)(2,1)	0.39
cmb	16	4	62	(4,2)(8,1)	0.98
count	35	16	174	(1,33)(2,1)	0.58
frg1	28	3	130	(1,26)(2,1)	0.41
lal	26	19	223	(1,16)(2,5)	1.87
pm1	16	13	85	(1,9)(3,1)(4,1)	0.31
term1	34	10	625	(1,32)(2,1)	0.48
x1	51	35	2141	(1,49)(2,1)	20.97
x2	10	7	71	(1,8)(2,1)	0.49
x3	135	99	1816	(1,133)(2,1)	9.37
x4	94	71	1040	(1,92)(2,1)	5.14
z4ml	7	4	77	(2,2)(3,1)	0.77
alu4	14	8	1278	(1,14)	0.51
apex6	135	99	904	(1,135)	12.88
des	256	245	7412	(1,256)	5.47
i5	133	66	556	(1,133)	10.65
i6	138	67	1037	(1,138)	10.12
i7	199	67	1311	(1,199)	12.77
i8	133	81	4626	(1,133)	13.65
i9	88	63	1453	(1,88)	1.12
pair	173	137	2667	(1,173)	21.38
rot	135	107	1424	(1,115)(2,5)(3,2)(4,1)	17.4

Chapter 6

Conclusions

In the SoC era, the embedded cores are mixed and integrated to create a system chip. The verification of the core-based system design should be focused on how the cores communicate with each other. However, before the interface verification, the interconnections between the cores in an SoC have to be verified first. System integrators integrate those cores manually and have the possibility of incorrect integration due to the misplaced I/O ports. Therefore, we adopt the connectivity-based POE model to raise the abstraction level of the design verification and to reduce the time on functional verification in core-based design methodology.

In this thesis, we present a new pattern generation algorithm to activate all remaining POEs and the GA technique to improve the UPSs_Calculation procedure. These two approaches get more precise remaining UPSs and therefore accelerate the AVPG and generates a more efficient verification pattern set for verifying core-based designs.

Besides, we use this GA technique for computing maximal sets of symmetric inputs of circuits. It can be used to identify nonsymmetric inputs in a circuit and enhance the efficiency of input matching, library binding (technology mapping), as well as logic verification problems. The experimental results demonstrate that our approach distinguishes more nonsymmetric inputs than that of previous work.

References

[1] M. Abramovici, M. A. Breuer, and A. D. Friedman, “Digital Systems Testing and Testable Design,” *Computer Science Press*, 1990, pp. 95.

[2] M. Agrawal and V. Arvind, “A Note on Decision Versus Search for Graph Automorphism,” in *Eleventh Annual IEEE Conference Computational Complexity*, 1996, pp. 272–277.

[3] H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly, and L.Todd, *Surviving the SOC Revolution—A Guide to Platform-Based Design*. Norwell, MA: Kluwer, 1999.

[4] R. Chang, W. Gasarch, and J. Toran, “On Finding the Number of Graph Automorphism,” in *IEEE Structure in Complexity Theory Conference*, 1995, pp. 288–298.

[5] D. I. Cheng and M. Marek-Sadowska, “Verifying Equivalence of Functions with Unknown Input Correspondence,” in *Proceedings European Design Automation Conference (EDAC),* 1993, pp. 81-85.

[6] K.-T. Cheng, and J.-Y. Jou, ”A Functional Fault Model for Sequential Machines,” *IEEE Transactions on Computer-Aided Design*, Sep. 1992, pp.1065-1073.

[7] P. Goel and M. T. McMahon, “Electronic Chip-in-place Test,” in *Proceedings IEEE International Test Conference*, Oct. 1982, pp. 83–90.

[8] A. Hassan, V. K. Agarwal, B. Nadeau-Dostie, and J. Rajski, “BIST of PCB Interconnects Using Boundary-scan Architecture,” *IEEE Transactions Computer-Aided Design*, Oct. 1992, pp. 1278–1288.

[9] H.-T. Liaw, J.-H. Tsaib and C.-S. Lin, “Efficient Automatic Diagnosis of Digital Circuits,” *International Conference On Computer-Aided Design,* Nov. 1990, pp.464-467.

[10] J. C. Madre, O. Coudert and J. P. Billon, “Automating the Diagnosis and the Rectification of Design Errors with PRIAM,” in *Proceedings International Conference on Computer-Aided Design,* Nov. 1989, pp. 30-33.

[11] J. Mohnke and S. Malik, “Permutation and Phase Independent Boolean Comparison,” in *Proceedings European Design Automation Conference (EDAC),* 1993, pp. 86-92.

[12] I. Pomeranz and S. M. Reddy, “On Diagnosis and Correction of Design Errors,” in *Proceedings International Conference On Computer-Aided Design,* Nov 1993.

[13] J. A. Rowson and A. Sangiovanni-Vincentelli, “Interface-Based Design,” in *Proceedings Design Automation Conference*, June 1997, pp. 178–183.

[14] U. Schlichtmann, F. Brglez and M. Hermann, “Characterization of Boolean Functions for Rapid Matching in EPGA Technology Mapping,” in *Proceedings Design Automation Conference,* June 1992, pp. 374-379.

[15] M. H. Schulz, E. Trischler, and T. M. Sarfert, ”SOCRATES: a Highly Efficient Automatic Test Pattern Generation System,” IEEE Transactions on Computer-Aided Design, Jan. 1988, pp.126-137.

[16] K.A. Tamura, “Locating Functional Errors in Logic Circuits,” in *Proceedings Design Automation Conference,* 1989, pp. 185-191.

[17] S.-W. Tung and J.-Y. Jou, “A Logic Fault Model for Library Coherence Checking,” *Journal of Information Science and Engineering*, Sept. 1998, pp. 567–586.

[18] C.-Y. Wang, S.-W. Tung, and J.-Y. Jou, “On Automatic-verification Pattern Generation for SoC with Port-order Fault Model,” *IEEE Transactions on Computer-Aided Design*, vol. 21, Apr. 2002, pp. 466–479.

[19] C.-Y. Wang, S.-W. Tung, and J.-Y. Jou, “An Automorphic Approach to Verification Pattern Generation for SoC Design Verification Using Port-Order Fault Model,” *IEEE Transactions on Computer-Aided Design*, vol. 21, Oct. 2002, pp. 1225–1232.

[20] D. B. West, ”Introduction to Graph Theory,” Upper Saddle River, New Jersey, Prentice-Hall, Incorporation, 1996.

[21] M. S. Abadir, J. Ferguson, and T. E. Kirkland, “Logic Design Verification via Test Generation,” *IEEE Transactions on Computer*, Jan. 1988, pp. 138-148.

[22] R. E. Bryant, “Graph-based Algorithms for Boolean Function Manipulation,” *IEEE Transactions on Computer*, Aug. 1986, pp. 677-691.

[23] P. Camurati and P. Prinetto, “Formal Verification of Hardware Correctness: Introduction and Survey of Current Research,” *IEEE Computer*, July 1988, pp. 8-19.

[24] D. I. Cheng and M. Marek-Sadowska, “Verifying Equivalence of Functions with Unknown Input Correspondence,” in *Proceedings of European Design Automation Conference (EDAC)*, 1993, pp.81-85.

[25] P. Y. Chung and I. N. Hajj, “ACCORD: Automatic Catching and Correction of Logic Design Errors in Combinational Circuits,” in *Proceedings of International Test Conference*, 1992, pp. 742-751.

[26] S. Devadas, H. K. T. Ma, and A. R. Newton, “On the Verification of Sequential Machinesat Differing Levels of Abstraction,” *IEEE Transactions on Computer-Aided Design*, June 1988, pp. 713-722.

[27] J. Jain, J. Bitner, D. S. Fussell, and J. A. Abraham, “Probabilistic Design Verification,” in *Proceedings of International Conference Computer-Aided Design*, Nov. 1991, pp. 468-471.

[28] K. Keutzer, “DAGON: Technology Binding and Local Optimization by DAG Matching,” in *Proceedings of Design Automation Conference*, 1987, pp. 341-347.

[29] S. Y. Kuo, “Locating Logic Design Errors via Test Generation and Don’t-care Propagation,” in *Proceedings of European Design Automation Conference (EDAC)*, Sep. 1992, pp. 466-471.

[30] H. T. Liaw, J. H. Tsaih, and C. S. Line, “Efficient Automatic Diagnosis of Digital Circuits,” in *Proceedings of International Conference Computer-Aided Design*, Nov. 1990, pp. 464-467.

[31] J. C. Madre, O. Coudert, and J. P. Billon, “Automating the Diagnosis and the Rectification of Design Errors with PRIAM,” in *Proceedings of International Conference Computer-Aided Design*, Nov. 1989, pp. 30-33.

[32] F. Maruyama and M. Fujita, “Hardware Verification,” *IEEE Computer*, Feb. 1985, pp. 22-32.

[33] E. J. McCluskey, “Detection of Group Invariance or Total Symmetry of a Boolean Function,” *Bell System Tech, J.*, Nov. 1956, pp. 1445-1453.

[34] E. J. McCluskey, *Logic Design Principles with Emphasis on Testable Semicustom Circuits*, Prentice-Hall, 1986.

[35] Alan Mishchenko, “Fast Computation of Symmetries in Boolean Functions” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 22, No. 11, Nov. 2003, pp. 1588-1593.

[36] J. Mohnke and S. Malik, “Permutation and Phase Independent Boolean Comparison,” in *Proceedings of European Design Automation Conference (EDAC)*, 1993, pp. 86-92.

[37] D. Moller, J. Mohnke, and M. Weber, “Detection of Symmetry of Boolean Functions Represented by ROBDDs,” in *Proceedings of International Conference Computer-Aided Design*, Nov. 1993, pp. 608-684.

[38] I. Pomeranz and S.M. Reddy, “On Determining Symmetries in Inputs of Logic Circuits,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* Vol. 13, No. 11, Nov. 1994, pp. 1428-1434.

[39] U. Schlichtmann, F. Brglez., and M. Hermann, “Characterization of Boolean Functions for Rapid Matching in EPGA Technology Mapping,” in *Proceedings of Design Automation Conference*, June 1992, pp. 347-379.

[40] K. A. Tamura, “Locating Functional Errors in Logic Circuit,” in *Proceedings of Design Automation Conference*, 1989, pp. 185-191.

[41] M. Tomita, H. H. Jiang, T. Yamamoto, and Y. Hayashi, “An Algorithm for Locating Logic Design Errors,” in *Proceedings of International Conference Computer-Aided Design*, Nov. 1990, pp. 468-471.

[42] R. L. Wadsack, “Design Verification and Testing of the WE 32100 CPUs,” *IEEE Design and Test*, Aug. 1984, pp. 66-75.

[43] R, S. Wei and A. L. Sangiovanni-Vincentelli, “PROTEUS: A Logic Verification System for Combinational Circuits,” in *Proceedings of International Test Conference*, Sep. 1986, pp. 350-359.

[44] A. S. Wojcik, “Formal Design Verification of Digital Systems,” in *Proceedings of Design Automation Conference*, June 1983, pp. 228-234

Vita

Chen-Ling Chou was born in Taipei, Taiwan on August 2, 1980. She received the M.S. degree in Electrical and Control Engineering from National Chiao Tung University in June 2002 and entered the Institute of Electronics, National Chiao Tung University in September 2002. Her major studies were Electronics Design Automation (EDA) and VLSI design. She received the M.S. degree from NCTU in June 2004.