Threshold Logic Properties and Methods: Applications to Post-CMOS Design Automation and Gene Regulation Modeling

by

Tejaswi Linge Gowda

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved September 2011 by the
Graduate Supervisory Committee:

Sarma Vrudhula, Chair
Aviral Shrivastava
Karamvir Chatha
Seungchan Kim

ARIZONA STATE UNIVERSITY
May 2012

ABSTRACT

Threshold logic has been studied by at least two independent group of researchers. One group of researchers studied threshold logic with the intention of building threshold logic circuits. The earliest research to this end was done in the 1960's. The major work at that time focused on studying mathematical properties of threshold logic as no efficient circuit implementations of threshold logic were available. Recently many post-CMOS (Complimentary Metal Oxide Semiconductor) technologies that implement threshold logic have been proposed along with efficient CMOS implementations. This has renewed the effort to develop efficient threshold logic design automation techniques. This work contributes to this ongoing effort. Another group studying threshold logic did so, because the building block of neural networks – the Perceptron, is identical to the threshold element implementing a threshold function. Neural networks are used for various purposes as data classifiers. This work contributes tangentially to this field by proposing new methods and techniques to study and analyze functions implemented by a Perceptron

After completion of the Human Genome Project, it has become evident that most biological phenomenon is not caused by the action of single genes, but due to the complex interaction involving a system of genes. In recent times, the 'systems approach' for the study of gene systems is gaining popularity. Many different theories from mathematics and computer science has been used for this purpose. Among the systems approaches, the Boolean logic gene model has emerged as the current most popular discrete gene model. This work proposes a new gene model based on threshold logic functions (which are a subset of Boolean logic functions). The biological relevance and utility of

this model is argued illustrated by using it to model different *in-vivo* as well as *in-silico* gene systems.

DEDICATION

To Kamakshi Linganna, without whom life would not have been the same.

ACKNOWLEDGMENTS

I would like to thank all the people who made this thesis, and the years of work that led to this possible. I first list all the professional help I received and then the people who made the emotional ride possible. Much of the following names overlap the two categories and are listed in no particular order.

First, I'd like to thank my advisor Sarma Vrudhula, my committee members Seungchan Kim, Aviral Shrivastava and Karamvir Chatha. I'd also like to thank Goran Konjevod, and Michael Bittner for their inputs.

Personally I'd like to thank all the people who have supported me in this endeavor. Here is an incomplete list – Viveck Cadambe, Guruprasad Jakka, Dustin Hurtt, Vinay Hanumaiah, Niranjan Kulkarni, Brandon Menc, Blake Quinn among many others.

I would also like to express my deepest thanks for the funding received from the National Science Foundation under award CCF-070283, the Science Foundation Arizona SFAZ-SBC and the Stardust Foundation, the Consortium for Embedded Systems, and the Department of Computer Science and Engineering through principal investigator Dr. Sarma Vrudhula, which provided my salary as a full time research assistant, tuition, travel expenses, and additional benefits from 2005 to 2010.

Finally and most importantly I'd like to thank my wife Ariana Gowda for her unwavering love, encouragement and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

BACKGROUND

Electronic circuits and modern genetics have revolutionized the world in the last few decades. Both these fields continue to unravel new developments, which will impact our lives in the foreseeable future. The work done as part of this Ph.D dissertation contributes to both these efforts. Semiconductor circuits have been around for many years now, but as will be discussed later in this chapter new developments require new paradigms of design. Genetics on the other hand has recently become a quasi-computer-science based field [26]. The completion of the Human Genome Project (HGP) [19] and the advent of micro-array technology [32, 82] has allowed for the increasing use of computational techniques to study, analyze and understand complex biological phenomenon through the analysis of genes involved. This chapter gives a brief introduction to both fields and how this dissertation contributes to each.

Information processing, paradigms of computation and limits of these paradigms have been studied by different fields of computer science [87]. Electronic circuit design borrows these results from computer science to design electrical circuits for practical applications while exploiting the properties of different physical materials [69]. The very first computers were built exploiting properties and characteristics of electric current flowing through vacuum [75]. But this was eventually replaced by circuits built using the properties of semiconductors. The unique properties of semiconducting materials has lead to reduction in the size and speed of electronic circuits [91].

Circuits built from semiconducting materials have been continually improved by shrinking the features of transistors [75]. This paradigm of scaling

is inevitably leading to problems involving manufacturing processes employed [9]. So far, these bottlenecks have been overcome by the use of ingenuous engineering solutions [9]. Thus the prediction made by Gordon Moore on the rate of scaling of circuits (commonly referred to as the Moore's Law [71]) has been true thus far [70]. Scaling has produced the reductions in size and cost, at the same time increasing the performance of electronic chips seen in the past [91]. However, this approach of scaling is now approaching limits imposed on physical properties of the materials used to build transistors [98]. As the feature size of transistors entered the nano-scale, it has become difficult to produce reliable circuits using existing fabrication techniques [86].

Currently circuits are fabricated using the Complementary Metal-Oxide Semiconductor (CMOS) technology [69]. Even though this particular transistor design style has been the most dominant, there have been other design styles proposed over the years [58]. Important among them is the Bipolar Junction Transistor (BJT) [75] and the now emerging futuristic devices like the Resonant Tunneling Diode (RTD) [34], Quantum Cellular Automata (QCA) [7], Single Electron Transistor (SET) [64] and Carbon Nano-tube Field Effect Transistor (CNT-FET) [4]. For reasons of fabrication economics, ease of design and because of the inherent advantages of CMOS (most importantly low-power, which was instrumental in it being preferred over BJTs) CMOS has trumped all other design styles [75]. The natural abstraction to design CMOS circuits is a network of AND/OR (and other simple Boolean gates) that implements the required logic [69]. This is a direct consequence of the advantages of organizing transistors as what is popularly known as pull-up-pull-down logic [75]. However, by using complex gates as design

primitives a more efficient (which in this context roughly means circuits with less gates and lesser depth) can be obtained [58]. This is however not possible with the current pull-up-pull-down design paradigm [75].

An alternative paradigm for designing digital circuits is threshold logic [73]. Threshold logic is a sub-set of Boolean logic [24] that is identical to the Perceptron used in neural network design [79]. A threshold function is a Boolean function that can be implemented using a threshold gate [24].

A *threshold element* or *gate*, has n binary inputs, $x_1, x_2, \cdots, x_n$, and a single binary output y. Its internal parameters are a threshold T and weights $w_1, w_2, \cdots w_n$. Every input x_i is associated with its respective weight w_i. The values of the threshold T and the weights w_i $(i \in \{1, \cdots, n\})$ may be any real, finite, positive or negative numbers [73].

The input output relation of a threshold gate is defined as:

$$y = \begin{cases} 1 & \text{if} \sum_{i=0}^{n} w_i x_i \geq T \\ 0 & \text{otherwise} \end{cases} \tag{1.1}$$

where the sum and product are arithmetic. The sum $\sum_{i=0}^{n} w_i x_i$, is called the *weighted sum* of inputs to the threshold element.

Example: Figure 1.1 shows a threshold gate that implements the function $y = a'b(c+d)$. Input a, b, c, d are assigned the weights -2, 2, 1 and 1 respectively and the threshold of this gate (T) is 3. The gate's output will be 1 when the weighted sum of inputs exceeds 3 (The value of T), and will be 0 otherwise.

Threshold circuits comprise of a network of interconnected threshold

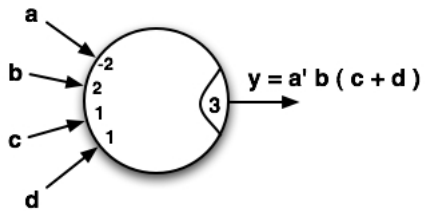


Figure 1.1: A threshold element implementing the function $y = a'b(c+d)$

logic gates that implement the required function. All the basic gates (such as AND, OR and their variants) are indeed threshold functions, and hence an existing circuit of AND/OR gates is also a threshold logic circuit [98]. However this is a simplistic form of threshold circuit and does not make use of the advantages offered by threshold logic which is the availability of complex gate primitives [97]. It is important to note here that threshold logic is still a very small subset of Boolean logic and that not all Boolean functions are threshold [58].

Even though currently the most dominant circuit design style is CMOS based Boolean AND/OR circuit, there is a convergence of two historic trends that are making threshold logic a viable alternative. Threshold logic has been studied mostly as a theoretical curiosity and has never been widely used due to the absence of efficient implementations. Incidentally, futuristic nano-devices which were originally investigated to find alternatives to CMOS (for an era when further scaling of CMOS transistors will not be possible), inherently implement threshold logic similar to the way pull-up-pull-down logic is a natural design style for for CMOS circuits [98]. Furthermore, more and

more efficient CMOS implementations of threshold circuits are being proposed [5]. This has motivated many researchers to look more closely into threshold logic circuits over the past few years [3, 88, 98]. This work is focused on aiding this effort when threshold logic circuits may come to dominate, or at least be a viable alternative to CMOS in the next decade. This argument is pictorially depicted in Figure 1.2.

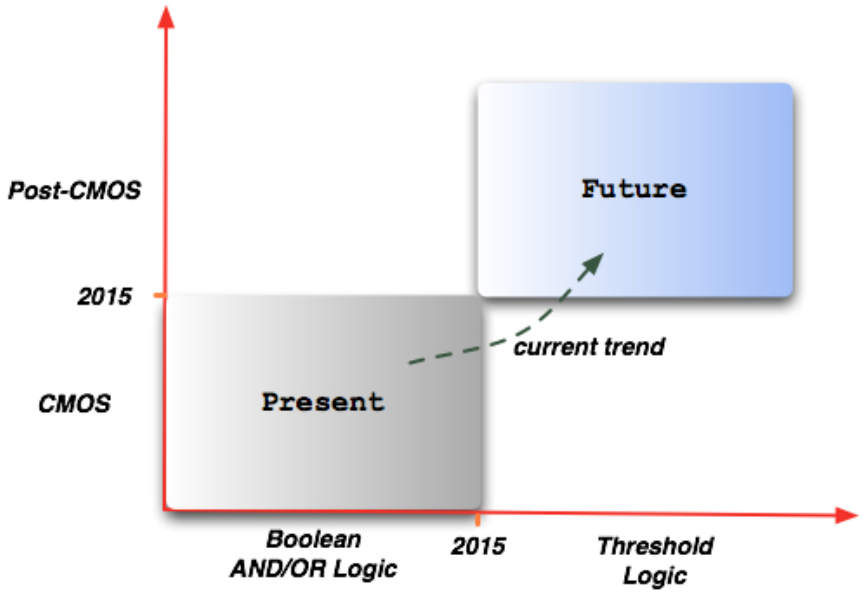


Figure 1.2: Trends in semiconductor devices and circuit design styles.

Irrespective of the design style for reasons of increasing complexity of digital circuits, shorter time-to-market, and a push for reducing costs the process of designing digital circuits is either completely automated or software tools have been developed that aid designers to analyze and optimize the circuits they engineer. The process of designing a digital circuit broadly falls into four phases (as described by Giovanni Micheli [69]; see Figure 1.3). These phases are similar to the phases of design employed in other fields – testing follows design and optimization, which is then followed by fabrication before the product is packaged and shipped. Each of these phases involves

many sub-steps, which are facilitated by the use of design automation tools. The design automation tools work with the abstraction of a circuit before making the design specific to the underlying manufacturing technology. This is known as technology independent design. This work focuses on technology independent design automation techniques for threshold logic circuits.

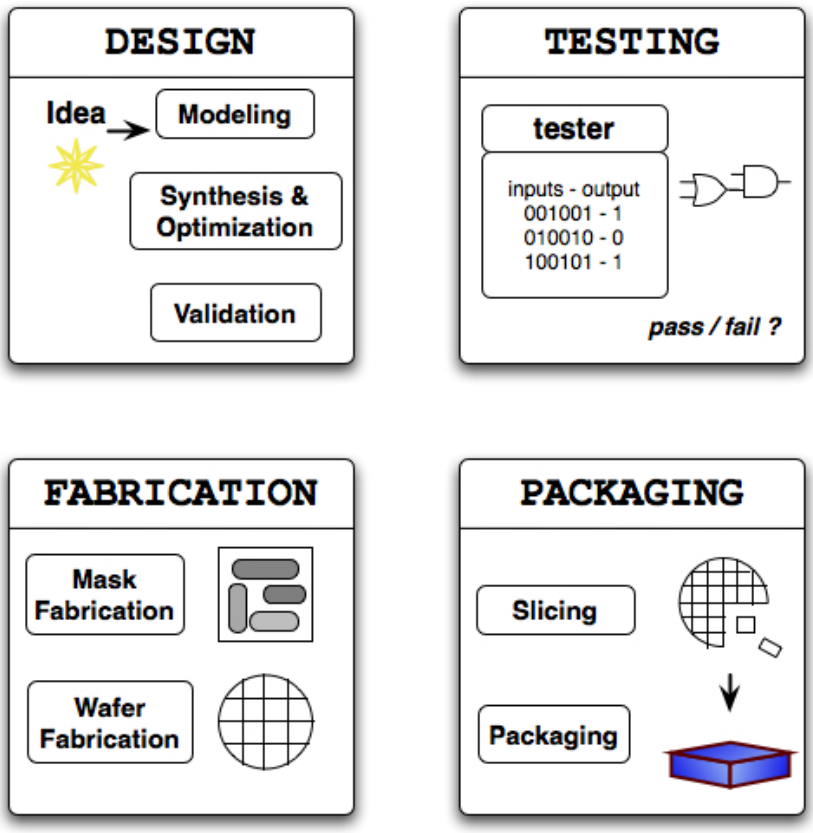


Figure 1.3: Different phases in the design of a digital circuit.

The four major problems that this work attempts to address (with regards to threshold design automation) are the problem of synthesis of threshold circuits, verification of threshold circuits, identification/characterization of threshold functions, and efficient implementation of state machines using threshold logic. These problems lay at the heart of some of the core problems in threshold design automation.

Another focus of this work is the development of accurate gene models. Although on the surface this appears to be different from the earlier described work on threshold circuit design automation, it is closely tied because it uses the properties and techniques developed for threshold design automation. In recent years the circuit model of gene action is gaining popularity [68] and Boolean logic has been extensively used to study both the qualitative and quantitative characteristics of gene action [55, 56]. Not only that many subsets of Boolean functions (like canalizing [72] and post-class functions [85]) have been proposed to better capture gene action than generic Boolean logic. To this end the current work explores the possibility of using threshold logic to explain the action of genes in creating complex biological phenomenon. This is done by both qualitatively arguing the biological relevance of threshold logic and by demonstrating the same by using the proposed approaches to model both *in-silico* and *in-vivo* gene systems.

To further motivate the work on threshold gene models a review of earlier work done in designing theoretical gene models, and the relevance of this work in light of the current state-of-affairs in biology is in order. For a long time biology has been an empirical, theory-neutral science [27]. But with the advent of modern genetics [90] and the central dogma of molecular biology [21] this is beginning to change. The central dogma of molecular biology essentially is that genes code for proteins (see Figure 1.4) and all the complex biological phenomena are caused by the action of proteins. This is a very powerful paradigm and has the potential of being the Atomic Theory of biology. Prior to the atomic theory, chemistry was an empirical, experimental science. But the deduction of sub-atomic particles and a theory of their organization and interaction led to the formulation of the theoretical

underpinnings of modern chemistry [8]. Biology is now at that juncture, where with the advent of new technology, ever more complex experiments can be done on gene systems and this might yield a theoretical underpinning similar to the foundation the Atomic Theory provided for modern chemistry.

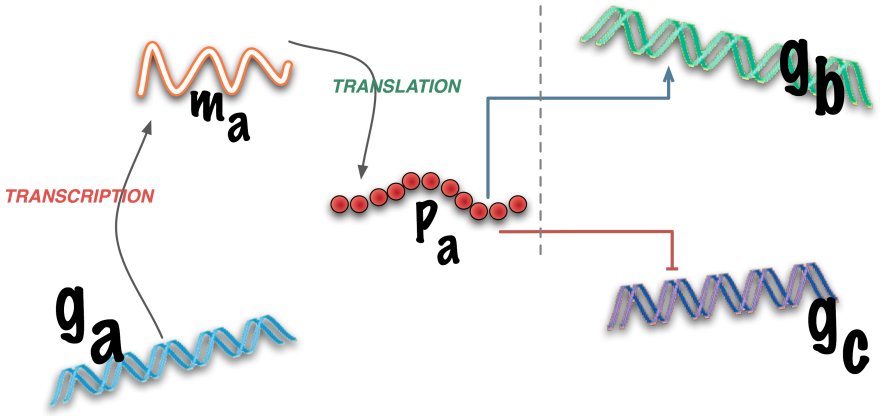


Figure 1.4: The process of transcription and translation through which DNA codes for proteins.

With this hope the Human Genome Project (HGP) was initiated. This project mapped the entire human genome [19]. This was possible because of the advances in biotechnology, as well as improved computers and sequencing algorithms. However, since the completion of the HGP, it has become clear that the one-gene-one-disease model is more of an exception than a rule [78]. Even though a single malfunctioning gene causes some genetic diseases, there are a plethora of inherited diseases, in which many genes have been found to be involved [6]. This has led to the current consensus that all gene action that causes complex biological phenomena is based on the non-trivial interaction of genes and their proteins [26]. This has resulted in the systems approach to study gene interaction. Since gene expression is inherently a computational process (four base pairs of DNA are read as a quaternary code, which when grouped together as triples, code for

the 20 amino acids that make up proteins [90]) computational techniques could be used in sequencing of genes. But now computer science is also used to construct complex models that can help study gene systems that are too large and too complex to be studied without such assistance [22]. Developing ever-accurate models of simple gene systems is one of the most important problems in computational biology [54].

Use of such computer tools to study real-world phenomenon is not new. Using physical laws, computer aided design and modeling tools are produced for designing cars, bridges etc. This is indeed possible because physical laws have been well understood for a very long time and can be accurately modeled within a computer [42]. But at this point in time we do not have good models to explain how gene systems create complex biological phenomena. One of the main reasons for this is that there is not enough biological data available to infer biological laws needed to develop accurate models [48]. At the same time as gene systems are complex it is difficult to plan experiments on them by intuition alone. This is a peculiar catch-22 that can be resolved as follows: using available biological data crude models can be built. These models can then provide specific insights and predictions on how the system should work. These can then be verified by actual experiments in the wet-lab thereby increasing the scope of biological knowledge. This symbiosis between model generation and biological experiments is depicted in Figure 1.5. Both these activities benefit from each other leading to increasing biological knowledge and more accurate models. The work as part of this dissertation intends to contribute to this ongoing effort.

Many gene regulatory models have been proposed by extending methodologies developed originally by mathematicians and computer

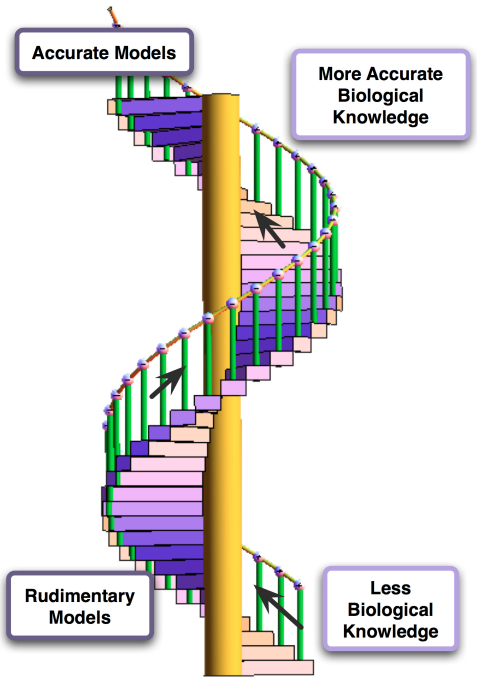


Figure 1.5: Experimental data drives modeling efforts and new models help generate relevant data. This symbiosis drives the research for more accurate biological modeling.

scientists. While each has its own specific characteristics they can be grouped broadly into two categories: the continuous models and the discrete models. Discrete models have some convenient features that make them more appealing. For example discrete models capture the behavior of the underlying systems in a crude manner, while retaining the essential behavior of the system. This makes these models easier to build than continuous models, which need accurate data to be built. However, gene expression data currently available is coarse-grained. Among the discrete models the Boolean logic gene model is the most popular one. This model has been shown to capture the qualitative behavior of gene interaction [45].

The Boolean gene model assumes that gene expression is digital, i.e.

each gene is either on (logic level 1) or off (logic level 0). The expression of each gene (henceforth called the target gene) is affected by a subset of other genes involved. In the Boolean gene model the rules by which a set of genes affect the target gene is represented by a Boolean function. The model is simulated in discrete time to observe how the gene system behaves over time. Although this does not happen in actual biological systems (gene interaction which is based on chemical interactions of proteins is asynchronous), the Boolean model can still capture the qualitative aspects of gene interaction like homeostasis and switch-like behavior by synchronous simulation alone. If starting with an initial state the model attains a steady state or a steady cycle, this is said to correspond to one of the equilibrium (homeostatic) states observed in biological systems.

Many extensions and variants of the Boolean gene model have been proposed using subsets of Boolean functions. Important among them are the canalizing and post-class functions. This work argues that using threshold functions (a subset of Boolean functions) gene regulation modeling can be accurately modeled. The biological relevance of threshold logic is argued by comparing the characteristics of threshold logic with the characteristics observed in actual *in-vivo* regulatory interactions. Additionally, the usefulness of this concept is demonstrated by using it to model different biological systems, like the genes involved in the embryonic patterning of *Drosophila* and genes implicated in skin cancer. The approach could also successfully model the dynamic properties of a *in-silico* gene system of 50 genes.

A brief overview of the specific problems addressed in this work is now given. These will be discussed in further detail in the subsequent chapters. Towards the end an annotated list of published work that discuss some of the

work described here in more detail is given. For more details the reader is referred to these.

The problem of design automation for threshold circuits is fundamentally different from the problem of design automation for Boolean CMOS circuits. As mentioned earlier a basic AND/OR Boolean circuit is also a threshold logic circuit, but it does not utilize the full repertoire of gates available. This work attempts to build a theoretical underpinning for threshold design automation, while borrowing from the now mature filed of CMOS design automation wherever relevant. The most important problem in developing design automation techniques for any design style is the problem of synthesis. Synthesis is the process of arriving at the required circuit when the function to be implemented is specified.

Earlier threshold synthesis algorithms are different variations that map the synthesized Boolean circuit onto a threshold circuit by selective recombination of nodes [98]. Other proposed algorithms have been an algorithm to synthesize feed-forward threshold circuits [3] and a two-level synthesis algorithm [88]. In contrast to these earlier approaches the proposed work attempts to develop a theoretical foundation based on manipulation of graphical representation of Boolean functions (mainly Boolean Decision Diagrams [13] and factor forms [40]).

Another closely related problem is the identification of threshold functions and characterization of the same. Identification involves deciding whether a Boolean function is threshold or not and characterization refers to assigning input weights and threshold to the threshold function. This work proposes a co-factoring based identification procedure which also

characterizes the identified threshold functions. This procedure yields a decomposition based synthesis approach. The advantage of the proposed approach is that it eliminates the ILP formulation to identify threshold functions. This improves all other threshold synthesis procedures that need to identify threshold functions.

Once threshold circuits are synthesized they need to be verified for accuracy. This problem is inherently hard for threshold circuits as in order to find the Boolean function that is implemented by a threshold gate all the one-points of that function should be determined. Detection of these one-points is an exponential complexity process (in the worst case) as equivalence of circuits for all combination of inputs need to be verified. As part of this work a polynomial-total-complexity algorithm is developed for the purpose of obtaining the Boolean function implemented by a threshold gate. This function can be used together with existing Boolean equivalence checking tools to verify the functional equivalence of threshold circuits.

Even though there now exists synthesis techniques proposed as part of this work (among others), there are many problems that need to be resolved before the true potential of threshold circuits is realized. Important among them is the implementation of state machines using a single layer of threshold logic gates. The problem involved in this is to identify if a single layer threshold implementation of a state machine exists. If such an implementation exists then the characterization of this circuit has to be done.

So far a brief outline of the problems addressed by this thesis with regard to threshold circuit design automation were discussed. Now a similar overview of the problems addressed by this thesis to develop better gene

regulation models based on threshold logic is given.

Since many subsets of threshold logic have been used before to model gene regulation models, a natural line of enquiry is to investigate if threshold logic is more suited for this purpose. Threshold logic has been used earlier to model the functioning of the brain using neural networks [2]. In some ways threshold logic is more suited for gene regulation modeling that generic Boolean logic. For example, the input weights and threshold naturally represents the degree of influence an input gene has on the target gene (since each input has its own weight). Positive and negative weights denoted if a gene has an activating or inhibitory effect on the target gene.

First, a simple proof-of-concept to demonstrate the ability of threshold logic to model complex biological phenomena was attempted. To do this one of the most well understood gene systems associated with fruit fly embryo development was selected. Using just a directed graph (known as the gene interaction graph) showing which genes affect any particular gene and how (activation/inhibition) a threshold logic gene model was developed. Since this system is well understood, this was relatively easy to do using just the gene interaction graph. The normal homeostatic state and the initial state were used to iteratively fine-tune the model. This model is extensively studied to generate predictions about the behavior of malfunctioning genes. Many of the predictions could be verified using existing literature, and many others are biologically meaningful, but could not be validated by existing literature. These could provide pointers for fruit fly geneticists to conduct future experiments. A detailed analysis of the state space generated by this model is also done. This thesis proposes two gene models that model (1) the anterior-posterior segmentation and (2) the dorsal-ventral germ layer formation that takes place

in the fruit fly embryo.

The approach used to model the fruit fly embryo development genes is based solely on the gene interaction graph. However, with the advent of micro-array technology [32, 82] most high-throughput data available today is from micro-array experiments. This is generally a time-series data where gene expression is measured in discrete time steps. Moreover, inferring the gene interaction graph is secondary since one or more gene interaction graphs can yield an accurate model of the underlying gene regulatory dynamics [95]. With the intent of developing a framework to infer the threshold gene model from micro-array data an automated procedure was developed. This procedure was tested on *in-silico* data provided as part of the Dialogue for Reverse Engineering Assessments and Methods (DREAM2) network inference challenge [74]. The method developed could accurately capture the dynamic properties of the underlying system, even though it had only moderate success in predicting the network structure (even so, it was chosen as one of the best performers in the DREAM2 challenge).

The specific problems discussed above are elaborated in the forthcoming chapters. This dissertation is intended to be complete and self-contained. However, the reader is referred to the annotated bibliography in the end as further reference.

Chapter 2

SYNTHESIS OF THRESHOLD LOGIC CIRCUITS

An important problem in regards to threshold design automation is the synthesis of threshold circuits. This thesis introduces 4 new algorithms for threshold logic synthesis. In this Chapter three of them will be introduced and the last algorithm will be discussed in Chapter 3. After a formal description of the synthesis problem, algorithms *3-ITC*, *n-ITC* and a tree-matching algorithm for threshold synthesis are presented.

2.1 Problem Formulation

The generic problem of threshold circuit synthesis can be stated as follows: *Starting with a given functional representation, generate a threshold circuit that implements the required Boolean function, as well as assign weights and threshold to each threshold element in the synthesized circuit.*

A threshold circuit is a directed graph in which each node represents a threshold gate. Every threshold gate has a weight corresponding to each input and a single threshold. The source nodes of this directed graph are primary inputs the output nodes form the sink. The generated threshold circuit and the given circuit specification are said to be functionally equivalent if for all input the circuit evaluates the output as specified in the circuit specification. A common circuit specification (which is the input to the synthesis algorithms presented in this Chapter) is the factored form representation. Starting with this representation the goal of these synthesis algorithms is to generate the functionally equivalent threshold circuit. This goal is achieved by the proposed algorithms by partitioning the factor form into a group of factor forms such that

each partitioned factor form is a threshold function. This partitioning of the factor forms can be done either by using inherent properties of threshold functions (the 3-ITC and n-ITC algorithms) or by pattern matching (the tree partitioning algorithm).

2.2 The non-ILP based Synthesis Method for 3-Input Threshold Logic Circuits

The 3-ITC algorithm take as input the maximally factored factor form of a function and generates a threshold circuit that implements the function using threshold gates of fanin not greater than 3. The algorithm makes use of the fact that that all the threshold functions that have fanin no more than 3 have one of the following seven factor form patterns:

- $a+b$.
- ab.
- $a+b+c$.
- $a+bc$.
- $a(b+c)$.
- $a(b+c)+bc, abc$.

Here, a, b and c are any three unique (positive or negative) literals. For e.g: the *pattern* $a+bc$ could represent $x'+yz$ or $p+q'r'$.

The input to the 3-ITC algorithm is represented as a factor tree. Each node of the factor tree is either a maximal POS or a maximal SOP[1]. This is

[1]A **Maximal Sum of Product (MSOP)** is a SOP which is not contained in any other SOP [40]. E.g. Consider the function $H = abc(x+y+z+d)+e$. Here $x+y$ is not a MSOP,

shown in Figure 2.1 (a). Consider the function $G = (a+b)(c+d+e')+xy'z'$. The factor tree for this is shown in Figure 2.1 (b). As shown in this Figure, the factor tree represents alternate layers of MPOSs and MSOPs. Note that single literals are both MSOPs and MPOSs. Every *AND* function is represented by a MPOS and an *OR* function represented by a MSOP.

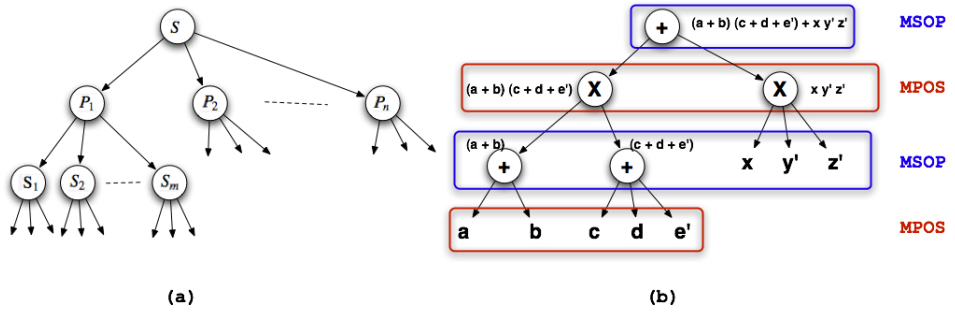


Figure 2.1: (a) Structure of a factor tree. (b) Factor Tree for $G = (a+b)(c+d+e')+xy'z'$.

The 3-ITC algorithm is first explained with the help of an example and then the complete algorithm is listed for reference. Consider the function $G = (a+b)(c+d+e')+xy'z'$. The 3-ITC algorithm considers single literals, two input *AND/OR* functions as trivial MPOS/MSOP. The 3-ITC algorithm represents these functions as the roots of the factor tree. The factor tree for G with this representation is shown in Figure 2.2(a). The only difference between this factor tree and the previous one (Figure 2.1 (b)) is that the node representing function $a+b$ is now a leaf. The 3-ITC algorithm works on this factor tree from the bottom up. For each node it visits it reduces the function of

since it is part of $x+y+z$ and $x+y+z+d$, but $x+y+z+d$ is a MSOP. A **Maximal Product of Sum (MPOS)** is a POS which is not contained in any other POS. E.g. Referring to the previous example, notice that $abc(x+y+z+d)$ is a MPOS in H.

the node to either a single literal or an *AND/OR* function of two literals. In this process, at each node the 3-ITC algorithm synthesizes threshold gates and assigns unique literals to the output of these gates.

So, going back to the example (function G), starting at the leaves, the 3-ITC groups the trivial MSOP/MPOS into a single literal, and generates a new gate for it. The 3-ITC algorithm has a set of individual literals and a set of two-input ORs at each MPOS node, and a set of literals and a set of two input ANDs at each MSOP node. This pairing of single literals with two input AND/OR functions identify the *patterns* $a+bc$ and $a(b+c)$. Another thing the algorithm does is that it groups three single literals together. This grouping helps identify the patterns $a+b+c$ (if it is a SOP node) and abc (if it is a POS node). Figure 2.2(b) shows the action of the 3-ITC algorithm at each node. At node $N1$ it combines the 3 individual literals c, d and e' into a single gate whose output is O_1. Similarly it combines three single literals at node $N3$ to form a single gate whose output is O_3. At node $N2$ the 3-ITC algorithm combines the single literal O_1 with the two-input OR, $a+b$, to synthesize a gate $O_2 = O_1(a+b)$. Node $N4$ has two single literals O_2 and O_3. These are combined to form a two input threshold gate $G = O_2O_3$. The complete threshold circuit synthesized by this procedure is shown in Figure 2.2(c). Since there are only seven patterns, the weights and thresholds are determined before hand.

In case of negative literals the weights can be adjusted using thw following well known result: If $f(X)$ is a threshold function and has a weight threshold assignment $[\{w_1, \cdots w_{a-1}, w_a, w_{a+1}, \cdots, w_n\}; T]$, then for $f(X; a \rightarrow a')$, $[\{w_1, \cdots w_{a-1}, -w_a, w_{a+1}, \cdots, w_n\}; T - w_a]$ is a feasible weight threshold assignment. (Theorem 3.2.2 page 58 in [73]). For example

$[w_{a'} = -2, w_b = 1, w_c = 1; T = 1] \equiv [w_a = 2, w_b = 1, w_c = 1; T = 3]$.

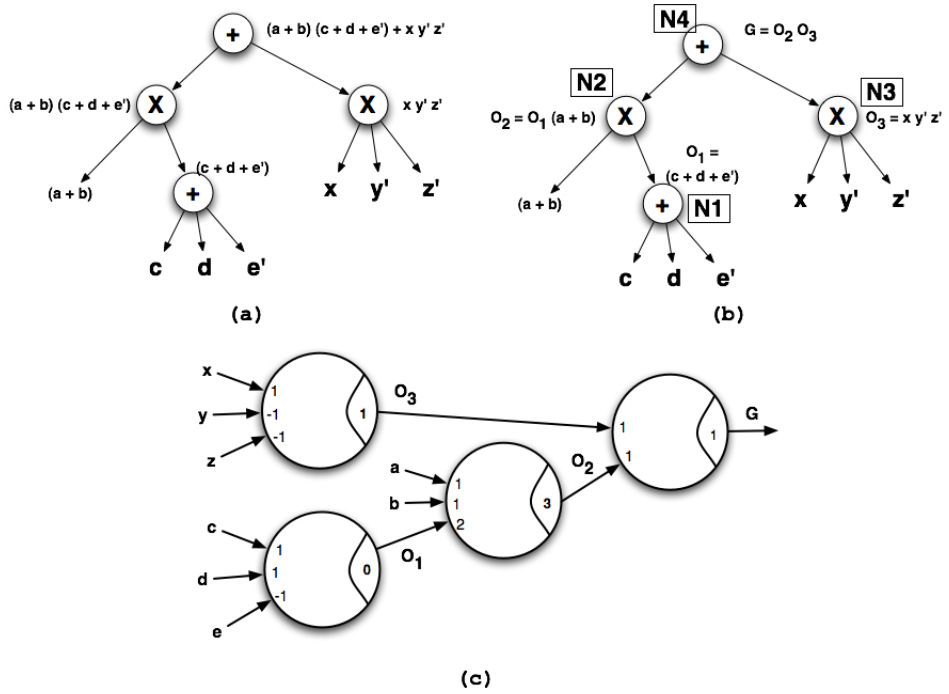


Figure 2.2: (a) Structure of a factor tree used as input to the 3-ITC algorithm (b) Working of the 3-ITC Algorithm. (c) The circuit synthesized for G, by the 3-ITC algorithm.

The example explains how the patterns $a+b+c$, $a+bc$, $a(b+c)$ and abc are detected. The patterns ab and $a+b$ are used only when a pair of literals are left in the root. This is an important feature of the 3-ITC algorithm. Since it uses a two input threshold gate only when a three input gate cannot be used, it can be shown the 3-ITC algorithm synthesizes a gate optimal circuit under certain conditions. This is discussed in detail later. As for the pattern $a(b+c)+bc$, when the 3-ITC algorithm is processing a MSOP node, it looks for these patterns before grouping the literals with trivial POS.

The algorithm is listed for reference. The algorithm $generate3ITC$ takes a factor form and synthesizes a 3-ITC implementing the factor form. It internally uses $SOPTGSyn$ and $POSTGSyn$ that call each other recursively. The function $Synthesize$ generates a threshold gate for the given function and assigns a unique output variable to the synthesized gate. The $combineLists$ function groups the literals with the trivial POSs/SOPs iteratively, until a single literal or a trivial SOP/POS is left. The $addToList$ function adds the argument to the $listofLiterals$ if the argument is a literal. If the argument is a trivial factor form it adds it to the $listofTrivialFFs$.

Algorithm 1: **generate3ITC:** Algorithm to generate a 3-ITC from a factored form

```
1:  INPUT: FF, a factored form that is not a trivial POS/SOP.
2:  if FF is a POS then
3:     returnFF = POSTGSyn(FF)
4:  else
5:     returnFF = SOPTGSyn(FF)
6:  end if
7:  if noofLiterals(returnFF) = 2 then
8:     returnFF = Synthesize(returnFF)
9:  end if
10: return  returnFF
```

Improving the Depth of Synthesized Circuits

While combining nodes (in *combineLists*), we can ensure that the increase in the number of levels is kept to a minimum. For this, depth information of each gate synthesized should be maintained. When generating a new gate, while popping out three elements from the $listOfOnes$, we can pop the elements with the least depth. This way the new gate generated will have the least depth, of all the gates we can generate in the current iteration of the algorithm.

Algorithm 2: **SOPTGSyn**

1: **INPUT: S**, a Sum-of-Product
2: listOfLiterals = [ ] listOfTrivialFFs = [ ]
3: **for all** Patterns $X(a+b)+ab$ is S (where X is a factored form) **do**
4: Y = Synthesize(*generate3ITC*(X)$(a+b)+ab$)
5: Replace $X(a+b)+ab$ in S, by Y.
6: **end for**
7: **for all** MPOS, P in S **do**
8: **if** P is a trivial POS **then**
9: addToList(P).
10: **else**
11: addToList(POSTGSyn(P))
12: **end if**
13: **end for**
14: **return** combineLists(listOfLiterals, listOfTrivialFFs, OR)

Algorithm 3: **POSTGSyn**

1: **INPUT: P**, a Product-of-Sums
2: listOfLiterals = [ ], listOfTrivialFFs = [ ]
3: **for all** MSOP, S in P **do**
4: **if** S is a trivial SOP **then**
5: addToList(S).
6: **else**
7: addToList(SOPTGSyn(S))
8: **end if**
9: **end for**
10: **return** combineLists(listOfLiterals, listOfTrivialFFs, AND)

This same kind of delay sensitive combination can be done while synthesizing any two or three input gate in *combineLists*. This will ensure generation of a network, that has less depth, than a network that is got by arbitrary combination of elements in the lists ($listOfLiterals$ and $listOfTrivialFFs$).

The algorithm *generate3ITC* produces a minimal gate 3ITC that implements the given maximally factored non-extractable (MFNE) factored form. The proof of this is non-trivial and is in the Chapter Appendix. Even

Algorithm 4: **combineLists**

```
1: INPUT: listOfLiterals, listOfTrivialFFs, φ
2: while Both lists (listOfLiterals and listOfTrivialFFs) are not empty do
3:   if size(listOfLiterals) = 1 and size(listOfTrivialFFs) = 0 then
4:     return (listOfLiterals.pop())
5:   else if size(listOfLiterals) = 2 and size(listOfTrivialFFs) = 0 then
6:     return (φ(listOfLiterals.pop(), listOfLiterals.pop())
7:   else if size(listOfLiterals) = 0 then
8:     addToList(Synthesize(listOfPair.pop())
9:   else if size(listOfTrivialFFs) = 0 then
10:    Pop the top three elements of listOfLiterals
11:    addToList(Synthesize(φ (the three elements poped)))
12:  else
13:    addToList(Synthesize(φ(listOfLiterals.pop(), listOfTrivialFFs.pop()))).
14:  end if
15: end while
```

though this algorithm is gate optimal, it is of limited use, since to harness the full potential of threshold logic gates of higher fanin will have to be used. However, this algorithm shows that it is theoretically possible to achieve gate optimal threshold circuit for a given fanin. This algorithm can be further extended to generate threshold circuits using gates of higher fanin. However, unlike the 3-ITC algorithm there is no guarantee on gate optimality.

2.3 Extension of the the non-ILP based Synthesis Method for n-Input Fanin Restriction Threshold Logic Circuits

The n input threshold circuit (n-ITC) algorithm is similar to the 3-ITC algorithm but it does not restrict the fanin of gates to 3. It is based on the following properties of threshold functions:

1. Boolean *AND* and *OR* functions are threshold functions.

2. If a Boolean function $f(x_1, x_2, \cdots, x_n)$ is a threshold function, then the

function $g(x_1,x_2,\cdots,x_{n+k}) = f(x_1,x_2,\cdots,x_n)+x_{n+1}+x_{n+2}+\cdots+x_{n+k}$ is also a threshold function.

3. If a Boolean function $f(x_1,x_2,\cdots,x_n)$ is a threshold function, then the function $h(x_1,x_2,\cdots,x_{n+k}) = f(x_1,x_2,\cdots,x_n).x_{n+1}.x_{n+2}\cdots x_{n+k}$ is also a threshold function.

The only way in which the n-ITC differs from the 3-ITC is that instead of pairing the trivial POS/SOP with single literals, it groups a threshold function with any number of literals (as long as the fanin restriction is not violated). Figure 2.3 (a) shows the steps involved in the n-ITC algorithm when it synthesizes the threshold circuit for the function $G = (a+b)(c+d+e')+xy'z'$. The synthesized circuit is shown in Figure 2.3 (b).

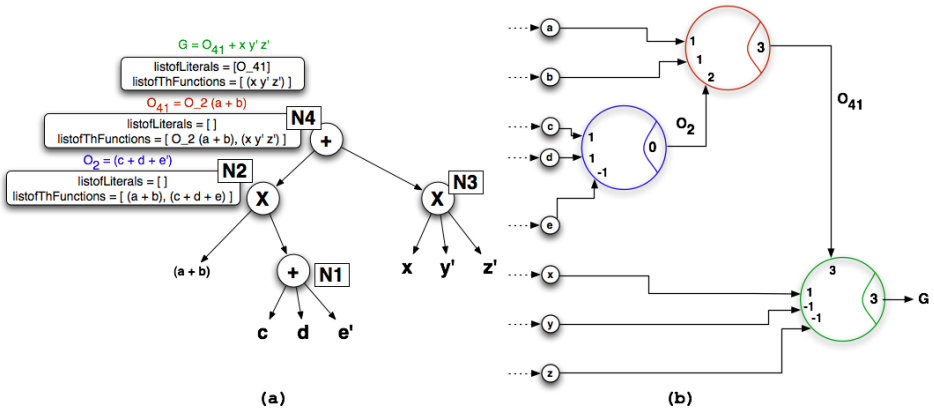


Figure 2.3: (a) Working of the n-ITC Algorithm. (b) The circuit synthesized for G, by the n-ITC algorithm.

The weight and threshold for these complex gates are assigned as follows: if n additional literals are and-ed to the threshold function F, such that

$G = x_1x_2\cdots x_nF$, then the additional literals are assigned the input weight equal to T_F (the threshold of F) and the threshold of function $T_G = (n+1)T_F$. *Example:* If $F = a+bcd \equiv [w_a = 3, w_b = 1, w_c = 1, w_d = 1; T_F = 3]$ and $G = x_1x_2F$, then $G \equiv [w_{x_1} = 3, w_{x_2} = 3, w_a = 3, w_b = 1, w_c = 1, w_d = 1; T_G = 9]$. In case n additional literals are or-ed to a threshold function F, such that $H = x_1x_2\cdots x_nF$, then the literals are assigned the input weight equal to T_F (the threshold of F) and the threshold of function $T_H = T_F$. *Example:* If $F = a+bcd \equiv [w_a = 3, w_b = 1, w_c = 1, w_d = 1; T_F = 3]$ and $H = x_1 + x_2 + F$, then $H \equiv [w_{x_1} = 3, w_{x_2} = 3, w_a = 3, w_b = 1, w_c = 1, w_d = 1; T_H = 3]$.

The depth of the circuits synthesized by the n-ITC algorithm can be improved by using a similar technique used to improve the depth of threshold circuits by the 3-ITC algorithm.

2.4 The Tree-Matching based Synthesis Technique

The drawback of the n-ITC algorithm is that it does not identify all the threshold gates. For example, the function $F = a(b+c+d)+b(c+d)+cd$ is a threshold function, but the n-ITC will generate a threshold circuit that has three gates in two levels (see Figure 2.4). Ideally, since the function F is threshold we would like to synthesize a circuit that implements the function in a single gate. It would be possible if we checked the factor tree of the function F against the factor trees of all threshold functions. This is the central idea of the tree matching approach for synthesizing threshold logic circuits propose next.

The Tree Matching Algorithm

The tree matching algorithm is formulated as a dynamic program to optimize the number of gates used to implement the threshold circuit. The algorithm

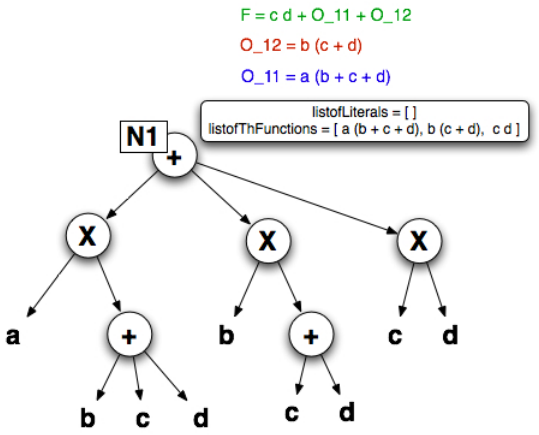


Figure 2.4: The n-ITC algorithm synthesizes a threshold circuit of three gates for a 4 input threshold function.

like the 3-ITC and the n-ITC starts off with the factor form. The tree matching algorithm then uses a *library* of factor trees of threshold gates to partition the factor tree of the function into sub-trees which are all threshold functions. For example consider the function $F = ab + cd'$. The factor tree for this function is shown in Figure 2.5 (a). If the set of factor trees of all threshold functions whose fanin is ≤ 4 is used as the library, then a valid partitioning is shown in Figure 2.5 (b). The circuit represented by this partitioning is shown in Figure 2.5 (c). Note that a unique new literal is assigned to the output of every gate synthesized. For example the output of the gate that implements the function ad' is assigned the label O_1. Similar to the earlier two algorithms negative and positive literals are not differentiated and the weight assignment for the negative literals is done by using Theorem 3.2.2 page 58 in [73]. The original set of weights are part of the library and hence weight assignment is

just a process of looking up the library and this is more efficient than using the ILP formulation that is used by earlier methods.

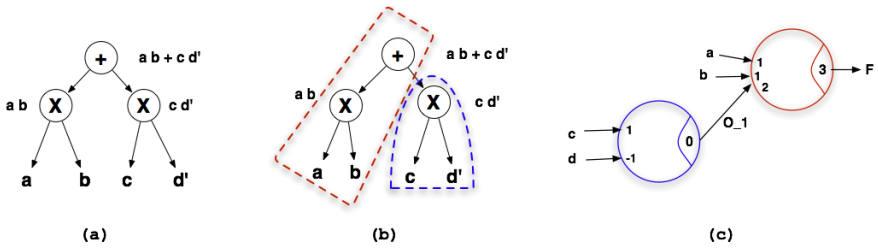


Figure 2.5: (a) The factor tree for the function $ab + cd'$. (b) A valid tree partitioning. (c) The threshold circuit represented by the tree partitioning.

The complete algorithm is listed in Algorithms 5, 6 and 7. The $treeMatching$ algorithm takes as input the factor tree of the function and generates the threshold circuit implementing the function. It uses the $getCost$ function to generate the $costMatrix$ and $Solution$ matrices that store the best solution (they are global variables). The algorithm checks all the threshold functions that would 'fit' the factor tree at the node and then choose the best (the one that will result in the least gate implementation). The $generateSolution$ function is used to reconstruct the solution from the $Solution$ matrix. In order to minimize the size of the *library* a standard factorization is chosen for constructing the factor tree and the solution. For the purposes of this work the min-max factor form is chosen as a standard [37]. This factor form has many interesting properties that are discussed in Chapter 3.

A comprehensive example is now presented to discuss the working of the algorithm. Consider the factor tree shown in Figure 2.6(a), generated by **GetFactorTree** (line 1 in Algorithm 5). The

Algorithm 5: Tree partitioning algorithm for TL Synthesis

1 ** *CostMatrix stores the min. gate count for each node.* **
2 ** *Solution stores best solution for each node.* **
3 ** *Initialize the elements in* ***CostMatrix*** *to* ∞ **

```
1: root = GetFactorTree(MFF)
2: GetCost(root)
3: GenerateSolution(root)
```

Algorithm 6: Least Gate TL Circuit Algorithm

```
1: if node is leaf then
2:    return 0
3: else
4:    for all Gate in Library do
5:       if Gate fits at node then
6:          ** L = the list of nodes in factor tree that corresponds to the leaves
            of Gate.**
7:          Cost = 1.
8:          for all inpNode in L do
9:             if CostMatrix[inpNode] < ∞ then
10:               Cost += CostMatrix[inpNode].
11:            else
12:               Cost += GetCost(inpNode).
13:            end if
14:         end for
15:         if Cost < CostMatrix[node] then
16:            CostMatrix[node] = Cost.
17:            Solution[node] = Gate
18:         end if
19:      end if
20:   end for
21:   return CostMatrix(node)
22: end if
```

Algorithm 7: Generating the solution TL circuit from the **Solution** array

```
1: Gate = Solution[node]
   Generate the weight and threshold for the corresponding function in the
   factor tree, matching Gate
   ** L = the list of nodes in the factor tree that correspond to the leaves of
   Gate.**
2: for all inpNode in L do
3:    if inpNode in Solution then
4:       GenerateSolution(inpNode)
5:    end if
6: end for
```

$Library = \{T1, T2, T3, T4, T5, T6, T7\}$, consists of factor trees $T1 = a + b$, $T2 = ab$, $T3 = a + bc$, $T4 = a(b + c)$, $T5 = abc$, $T6 = a + b + c$ and $T7 = a(b + c) + bc$. **GetCost** is invoked (line 2 in Algorithm 5) with the root of the factor tree as the argument and we get the **Solution** and $costMatrix$ shown in Figure 2.6(b). The algorithm fills up each entry in the $costMatrix$ only once. Figure 2.6(e) shows the execution of the algorithm for each node of the factor tree. **GetCost** is invoked for the leaf nodes of the matched tree when there is a match and the present cost of the node is *infinity* (lines 5 - 12 in Algorithm 6). *e.g:* for node $N2$, $T2$ and $T4$ match (Figure 2.6(d) shows how this matching is done). The best cost and solution are stored (lines 15 - 17) *e.g:* For node $N2$, $T4$ gets priority over $T2$. For the sake of simplicity only the cost is shown to be returned by the algorithm in the figure. During its execution, the algorithm fills in the **Solution** array, from which the final circuit is constructed (Figure 2.6(c)).

Optimality of the Tree Matching Algorithm

Optimality of the tree matching procedure is ensured by choosing the best circuit implementation for each node in the factor tree (lines 15 - 17 in

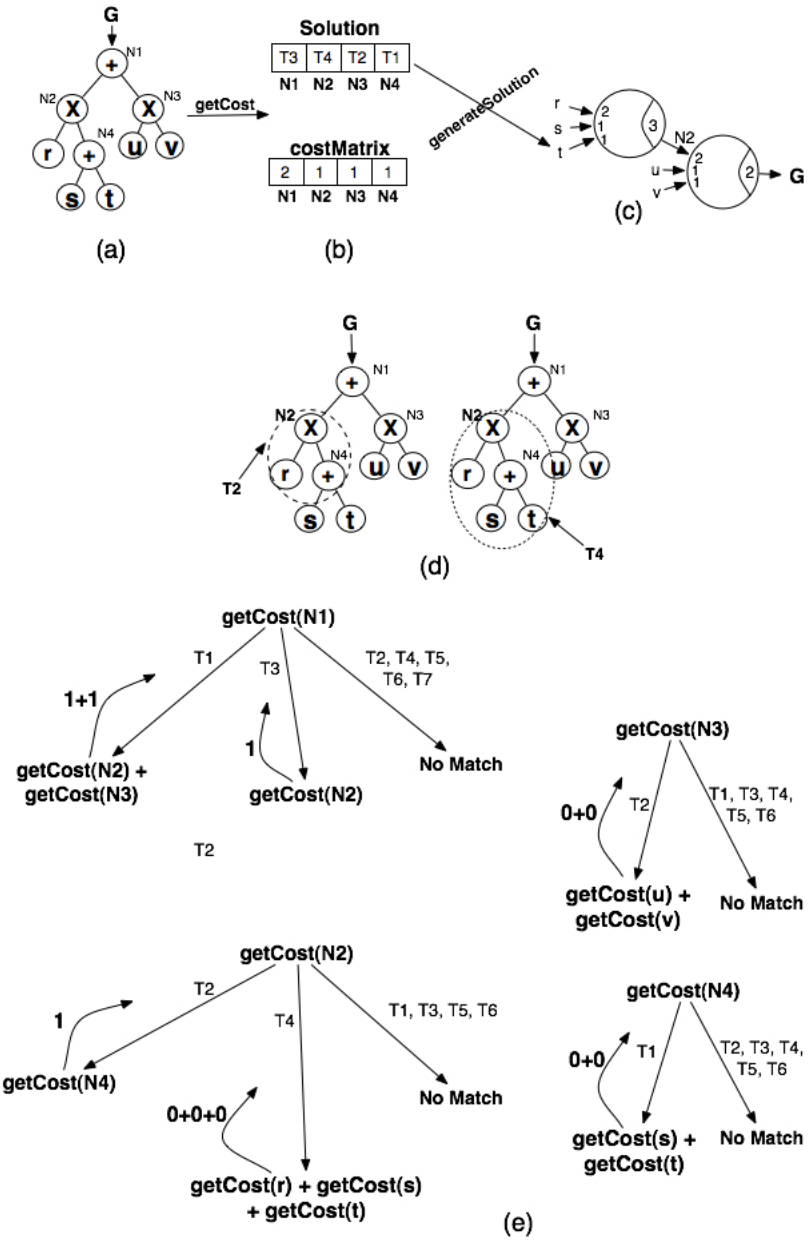


Figure 2.6: Sample Execution of The Tree Matching Algorithm

Algorithm 6). Since $Solution$ stores the gate-minimal implementation for each node, after the algorithm completes execution the gate optimal solution for the root node is present in $Solution$. Thus, the solution generated by the algorithm has the least number of gates.

The reduction in the number of gates for a threshold circuit is more complex than for a Boolean circuit. After obtaining the minimal literal factored form, obtaining the minimal gate implementation is a non-trivial problem. In this regard, the proposed algorithm guarantees a gate minimal implementation for a particular factored form. The proposed algorithm gives the optimal gate-efficient implementation for a single output function. For a multi-output function, no such claim can be made, as the quality of the result depends on the initial logic optimization and extraction.

2.5 The Synthesis Framework

The three algorithms proposed here take a factor form of a Boolean function and generate a threshold function that implements that Boolean function. But any generic circuit has multiple inputs and outputs. The different output functions share common logic. In such a situation extraction of common logic often leads to better results (fewer gates) [69]. In order to extract common logic functions the Berkeley SIS tool [28] is used. This is a reasonable thing to do, as more than two decades of work has been done to improve extraction techniques for Boolean functions. SIS was used to perform extraction on the original Boolean function. This is done by using the *simplify* command. The factor form of the extracted function is obtained by using the *print_factor* command (see Figure 2.7).

The output obtained from SIS is a circuit graph, implementing the

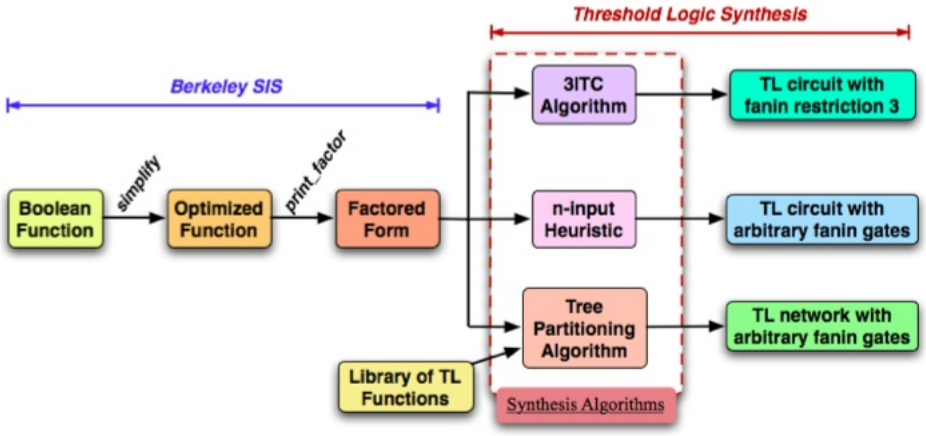


Figure 2.7: The Synthesis Flow

multi-output function. Each node in the circuit graph represents a complex Boolean function. Any of the proposed algorithms can be used to obtain the threshold circuit which implements the function of each node in the circuit graph. These threshold circuits can be put together (using to the circuit graph) to obtain a final threshold circuit that implements the required multi-output circuit.

2.6 Experimental Results

Computational Complexity

The computational complexities of both the 3-ITC and the n-ITC algorithm are analyzed first. Note that both algorithms does computation only when synthesizing a gate (when they assign weights and threshold to the new gate). So by obtaining the the number of gates synthesized, we will get the computational complexity of the two algorithms. At each node the two algorithms synthesize no more new gates than the number of children to the node. Therefore the number of gates synthesized is bound by the number of

edges in the factor tree. The work done to synthesize each gate is constant, and hence if n is the size of the input factor form (the number of literals) then the number of edges in the factor tree is bound by $O(n)$, and hence the number of gates synthesized is also bound by $O(n)$. Since the work done to assign weights for each gate is constant the complexity of the 3-ITC and the n-ITC algorithms is $O(n)$ (***linear time complexity***).

In the Tree Matching algorithm the time spent on a single call to $GetCost$ is $O(1)$ (since the size of the library is finite and fixed, it is constant). This is excluding the time spent in the recursive calls it generates. So the running time is bounded by a constant times the number of calls issued to $GetCost$. Since the algorithm gives no explicit upper bound on the number of calls, a bound is found by looking at a good measure of *progress* [57].

The most useful measure of progress is the number of entries in $CostMatrix$ (the global variable in Algorithms 5, 6 and 7) that are not *infinity*. Initially the number is zero. Each time the procedure is invoked by recursion, a new entry is filled. Since $CostMatrix$ has only $O(n)$ entries, n being the number of nodes in the input factor tree, there can be at most $O(n)$ calls to **GetCost**. Therefore the running time of $GetCost(node)$ is $O(n)$. $GenerateSolution$ and $GetFactorTree$ (lines 1 and 3 in Algorithm 5) both have $O(n)$ complexity. Hence the $treeMatching$ algorithm has $O(n)$ complexity (*linear time complexity*).

Summary of Circuit Parameters Obtained for MCNC Benchmark Suite Circuits

A histogram (see Figure 2.8) is given to compare the results got by the 3-ITC algorithm, as compared to the network generated by the method in [98]. The positive x-axis denotes improvements in gate count, and the negative x-axis represents the cases where 3ITC does worse. 3-ITC was run on 42 example

circuits in the MCNC benchmark suite. This comparison is unfair as 3-ITC uses only 3 input gates, whereas the method in [98] uses 6-input gates. However the results are still comparable.

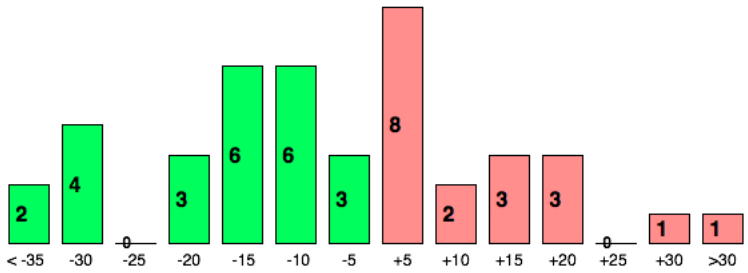


Figure 2.8: Comparison of 3ITC gate count with that of [98]

The results in [98] are compared with the n-ITC algorithm where n is set to 6. This fanin restriction is chosen for fair comparison because the circuits reported in [98] also have the same fanin limit. For the benchmark circuits a gate count improvement of 14.14%, and a level improvement of 24% was obtained. The level count can be further improved by using level sensitive n-ITC. Full results is listed in Table 2.1.

The results in [98] are compared with the circuits obtained by the tree matching algorithm. To run the tree matching algorithm first the library of all threshold gates is generated. For this the list of threshold functions provided in [73] is used. The list has all 2 to 5 input threshold networks. A subset of 6 input threshold gates were generated by using the 5 input threshold functions. If $f(x_1,\cdots,x_n)$ is a threshold function, then $y+f(x_1,\cdots,x_n)$ and $z.f(x_1,\cdots,x_n)$ are also threshold functions [73]. Therefore using five input threshold function

Table 2.1: Comparison of n-ITC with previous work

Benchmark	R Zhang et. al		6-Input Heuristic		Difference	
	Gates	Levels	Gates	Levels	Gates	Levels
b1	8	3	5	4	-3	-1
cm42a	13	3	10	1	-3	2
decod	24	3	16	1	-8	2
cm82a	12	4	12	2	0	2
majority	1	2	4	3	3	-1
parity	45	9	30	2	-15	7
x2	15	4	25	5	10	-1
cm85a	14	5	17	5	3	0
cm151a	12	5	16	5	4	0
cm162a	26	8	18	4	-8	4
cu	24	4	23	3	-1	1
cm163a	25	6	19	6	-6	0
cmb	27	6	20	9	-7	-3

a subset of six input threshold functions can be generated. Next, the factored forms and the generated factor trees for these functions are obtained. The collection of these factor trees constitutes the library, which is an input to our procedure. Compared to the method in [98] (full results is listed in Table 2.2), the proposed method generates circuits with comparable depth and 25% fewer gates on average (50% at best). Even though the algorithm is optimal for single output functions, for some (3 out of 43) circuits it does worse than the method in [98], as they are multi-output functions. The algorithm presented demonstrates a large improvement over Boolean logic implementations. It also demonstrates significant improvement over the previous TL synthesis method. The fan-in restriction assumed (six) is reasonable in light of device considerations. In the few circuits where the method does worse, results can be improved by better logic extraction for threshold logic.

Table 2.2: Comparison of Tree Matching Algorithm with Previous Work

Bench-mark	Boolean Circuit		Method in [98]		Tree Matching	
	Gates	Levels	Gates	Levels	Gates	Levels
b1	10	4	8	3	6	4
cm42a	13	3	13	3	12	3
decod	24	3	24	3	18	2
cm82a	18	5	12	4	12	6
majority	5	3	1	2	1	1
parity	45	9	45	9	30	8
z4ml	39	8	19	5	24	5
f51m	101	8	82	8	44	4
9symml	141	10	110	9	85	10
alu2	253	27	197	25	161	28
x2	20	5	15	4	20	3
cm152a	13	4	11	4	9	4
cm85a	26	5	14	5	14	6
cm151a	14	6	12	5	10	4
alu4	517	28	410	23	320	31
cm162a	39	7	26	8	18	5
cu	31	6	24	4	24	4
cm163a	40	6	25	6	15	4
cmb	33	7	27	6	21	12
pm1	25	4	23	4	24	4
tcon	32	3	32	3	16	2
pcle	42	6	35	6	26	8
sct	54	6	38	5	38	5
cc	49	6	35	6	26	3
cm150a	25	5	21	4	18	5
cordic	61	9	49	7	30	6
ttt2	127	7	100	6	89	7
pcler8	50	7	47	7	34	9
frg1	97	12	59	9	40	7
c8	109	8	85	7	62	5
comp	89	9	83	8	62	12
my_adder	160	34	96	18	65	19
term1	278	11	226	10	118	9
count	91	12	79	12	48	18
unreg	66	4	50	5	48	3
cht	119	5	82	5	73	3
apex7	171	10	118	9	106	9
x1	293	8	203	7	104	7
example2	226	9	182	8	146	8
x4	264	7	189	8	176	6
apex6	543	12	396	12	326	10
x3	660	9	441	7	352	11
pair	1199	14	907	12	611	14

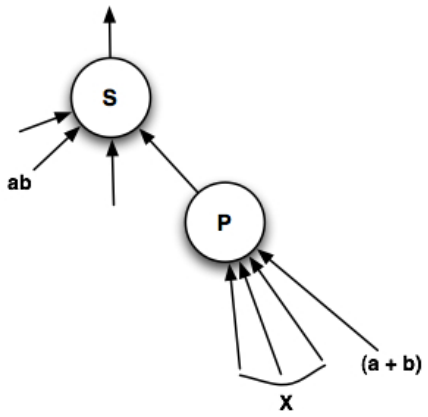


Figure 2.9: Pattern V in a factor tree.

APPENDIX: Proof of Gate Optimality of 3-ITC algorithm

The algorithm *generate3ITC* produces a minimal gate 3ITC that implements the given maximally factored non-extractable (MFNE) factored form. The algorithm first identifies all patterns of the type $X(a+b)+ab$ (call it Pattern V). In Pattern V, X is any factored form and a and b are two unique literals.) and generates 3ITCs for them. To do this it generates a 3ITC for X, and uses the output of this circuit, as an input to a three input gate. The other two inputs to this gate are a and b, and this gate implements the function $X(a+b)+ab$. We first prove that any network that implements the MFNE factored form and ignores this pattern can be transformed to a network that will synthesize a gate using this pattern without increasing the number of gates.

Consider a network that ignores at least one pattern V. Consider the

pair of nodes S and P in a factor tree T (Refer Figure 2.9), in which the pattern $X(a+b)+ab$ occurs, but is ignored. Let S and P be such that no other such pattern occurs in P. Now the node P outputs either a pair of literals or a single literal as input to S. If we choose to ignore $(a+b)$, then we can get the same output pattern at the output of P, with one less three input gate. This follows because in an optimal network, the number of two-input gates needed, depends only on the number of single literal inputs to P (Appendix I). Thus we get the same effect with one less gate. If P outputs a pair, then to reduce it to a single literal we need an extra gate. The previously saved gate will compensate for this extra gate and so far network transformation has not yielded any extra gates. Now the single literal output of P can be combined with the cube ab and $a+b$ to implement $X(a+b)+ab$ in a single gate. Now we have used an extra gate, but also eliminated the gate to which ab was input.

Example: Consider the function, $G=ab(c+d)+cd+g$. Figure 2.10 shows two implementations. The first one ignores the pattern V. It requires 3 gates. The alternate implementation maps a gate for pattern V and is still optimal (both require 3 gates)

Thus in transforming the optimal network into another one that uses the pattern V, we preserve optimality. This can be done repeatedly to each pair P and S. Hence there exists an optimal network that uses all the V, that occur in the factored form.

Our algorithm generates a network, that maps all patterns of the type V. Now it is enough to prove that for a factored form that is independent of pattern V, the 3-ITC algorithm will still find the optimal mapping.

Preliminaries:

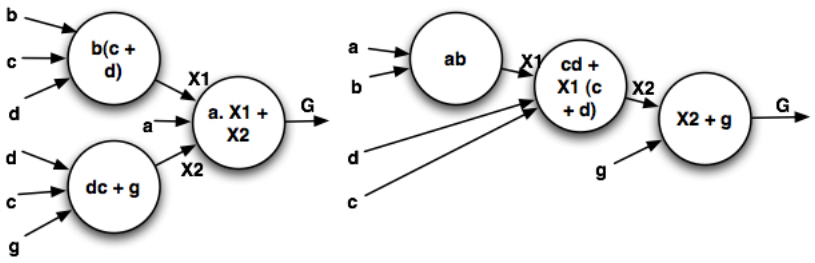


Figure 2.10: Two networks that implement $G = ab(c+d)+cd+g$

From now on consider MFNE factored forms that do not have pattern V. Each network that implements a factored form generates a *labeling* of the edges of the factor tree. The labeling is defined as follows:

If the MSOP or MPOS of the node in the factor tree is reduced to a single literal, then the output edge of the node is labeled as S (single), otherwise it is labeled as P (pair). If an edge is connected to a leaf, then the number of literals in the function of the leaf determines the label of the edge. If the leaf node is either pattern ab or $(a+b)$ then the edge label is P, else it is S.

Now consider an optimal threshold network O and the corresponding labeling L_o. Our solution (call it A), generates a labeling L_a. An edge is said *good* if its labels in L_o and L_a are the same, otherwise it is said to be *bad*.

Now a coloring of the nodes of the factor tree is defined as follows:

A node in a factor tree is colored white if its input edges are good but the output edge is bad. If the output edge and at least one of the input edges are bad, then the node is colored grey. If any of the input edges are bad, but

the output edge is good, then the node is colored black. All the nodes that are uncolored are the ones whose input and output edges are both good. Each node also has a label N_2, which is the number of two input gates used to reduce the input at the node to the output at the node.

For the nodes that are not colored it can be seen that A uses the exact same number of N_2 nodes as O (The minimum N_2 for optimal mapping is given in Appendix I). Therefore for the uncolored nodes. $N_{2O} - N_{2A} = 0$

Now uncolor all the nodes, by a series of transformations, thus transforming O to A. To show that A generates an optimal network, it is now enough to show that each of the intermediate networks generated by the transformation, is not sub-optimal when compared to O. This construction will prove that A will have no more two-input gates than O. But since O is optimal, by the corollary of Theorem 1 (which states that if a 3ITC implementing a MFNE factor form of a function, uses the least possible number of 2 input gates, then it is gate optimal among all the 3ITCs that implement the same MFNE factor form; see Appendix V), A is optimal.

The network transformation done to uncolor nodes is explained in the next section.

Network Transformations:

The colored nodes of the factor tree form a forest of trees. A typical tree is shown in Figure 2.11.

Choose from any tree a subtree S, such that its root node has more than one child, and all other non-leaf nodes have only one child. If no such sub-trees exist then all the trees in the forest are just simple paths, having the

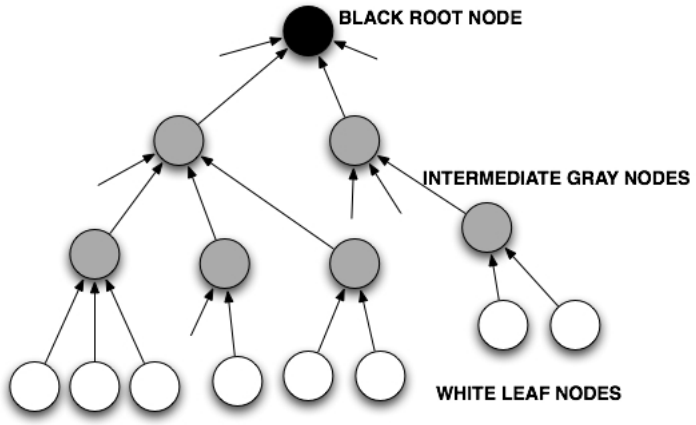


Figure 2.11: Coloring of the nodes of a factor tree.

black node at one end, zero or more gray nodes, and a single white node at the other end. Each sub-tree found will have one of the two structures shown in the Figure 2.12.

Consider a path of only grey nodes and a white node at the end (call it a grey-white path). From Appendix II, we know that for a white node the difference between the numbers of two input gates used for O, and the number of two input gates used by A is 1. i.e if O_{N_2} and A_{N_2} are the number of two input gates synthesized by O and A respectively, then, for a white node: $O_{N_2} - A_{N_2} = 1$

Similarly, it is shown in Appendix III that for a grey node: $O_{N_2} - A_{N_2} = 2$ or $O_{N_2} - A_{N_2} = 0$. Note that: $O_{N_2} - A_{N_2} \geq 0$, for a grey node. Now for a grey-white path: $O_{N_2} - A_{N_2} \geq 1$

It can be seen that the root of S has these grey-white paths as children.

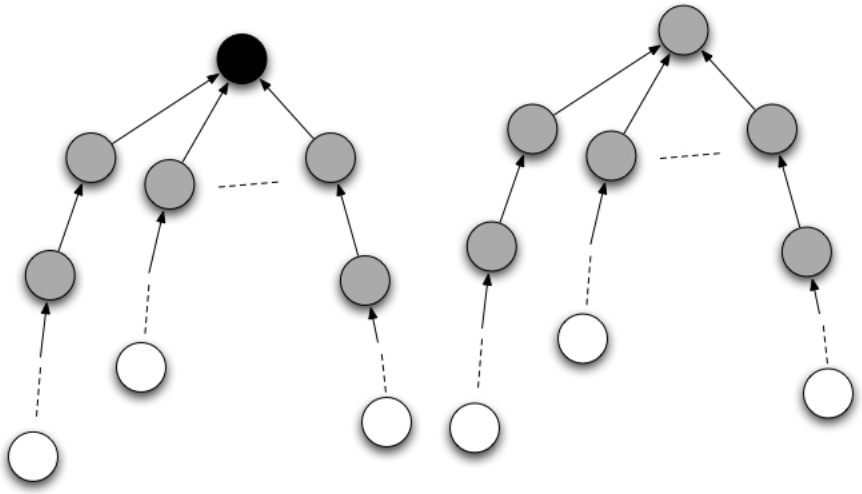


Figure 2.12: Structure of the subtrees (S)

If each of these grey-white paths is replaced by the implementation of A (i.e for each node of these paths synthesize the gates as done by A), this will lead to a savings $\geq n$ in the number of two input gates. But such a replacement will change the input edge labels of the root node of S(which is either a black or a grey node). But from Appendix I, it can be seen that from any input configuration, to get any form of output, we'll never need more that 2 two-input gates. Now replace the root of S too, by an optimal implementation, that will result in the same output form as before. This will not take more than 2 extra two-input gates.

Savings in 2-input gates by the replacement = $n - 2$. But $n \geq 2$, thus savings in two input gates ≥ 0.

This replacement will yield an implementation, that as good or better

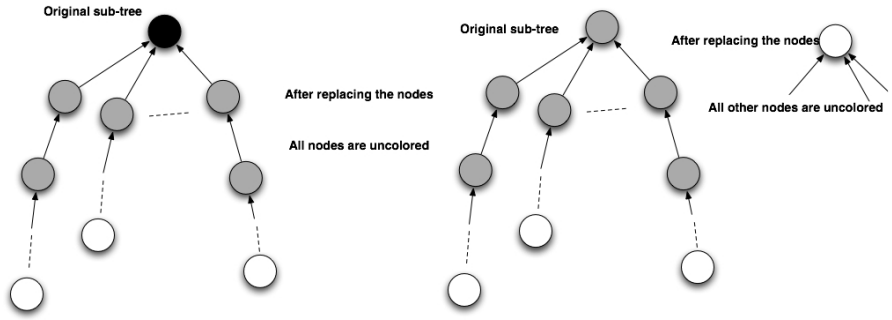


Figure 2.13: Effect of uncoloring the nodes in S.

than O.

So far we've shown how to replace the entire sub-tree S, but not change the rest of the implementation. This replacement will uncolor[2] the grey (non-root) and white nodes of S. If the root node was grey, it's turned into white, and if the root node of S was black, then it gets uncolored. This is shown in Figure 2.13.

The sub-trees of the form S can be iteratively replaced. Such replacements will yield networks that implement the same function, but are no less optimal than O. After all the sub-trees of the form S are processed and replaced we are left only with simple paths of the type shown in the figure 2.14.

These paths have a white node, a set of zero or more grey nodes and a black node at the end (call it the black-grey-white paths).

From Appendix IV, we know that the black node will not take more than one extra two-input node, if it's made to align its implementation to that of A.

[2]An uncolored node is one whose incoming and outgoing edges have the same labels for both networks O and A.

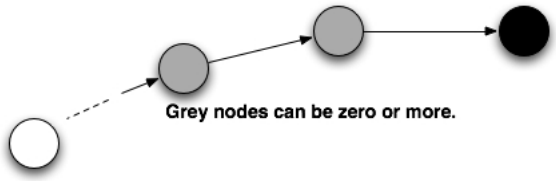


Figure 2.14: Structure of “Simple Paths"

i.e $A_{N_2} - O_{N_2} \leq -1$, or $O_{N_2} - A_{N_2} \geq -1$.

For the rest of the grey-white path, we know: $O_{N_2} - A_{N_2} \geq 1$.

Hence for the entire black-grey-white path: $O_{N_2} - A_{N_2} \geq 0$.

This implies that replacing all the nodes in the black-grey-white path, by making them the same as the corresponding ones of A, we are not sacrificing the gate optimality of the 3ITC algorithm. Such replacement can be viewed as uncoloring the nodes, of the black-grey-white path.

Appendix I

This section lists the minimum number of two input gates required to reduce different inputs that can occur at any node of the factored tree to one of the two output forms (single literal or pair).

The different configurations of input and output forms are listed in the Table 2.3. The number (minimum) of two input gates, that will be required for each case is enumerated.

This table is used repeatedly to determine $O_{N_2} - A_{N_2}$, in the next few

n[3]	Output Form	N_{2min}[4]
Odd	Single Literal	0
Odd	Literal Pair	1
Even	Single Literal	1
Even ($n \geq 2$)	Literal Pair	0
Even ($n = 0$)	Literal Pair	2

Table 2.3: Number of N_2s required for optimal implementation

Appendices.

Appendix II

This section investigates the difference in the number of two-input gates of O and A ($O_{N_2} - A_{N_2}$), for the different input combinations that can occur at a white node of a factor tree.

Let n, be the number of single literal inputs at the node, and m be the number of inputs to the node that occur as pairs.

CASE 1: $n = 0$. A will reduce the output to the form a, and will use one two input gate. O will reduce the output of the node to a pair (this is because in a white node the output format of A and O are different). For this O will use two two-input gates. Thus, $O_{N_2} - A_{N_2} = 1$.

CASE 2: $n > 0$**, n is even.** A will reduce the output to a pair of literals, and will not use any two input gates. O will reduce the output of the node to a single literal form. For this O will use one two-input gates. Thus, $O_{N_2} - A_{N_2} = 1$.

CASE 3: $n > 0$**, n is odd.** In this case, A will reduce the output to a single literal, and will not use any two input gate. O will reduce the output of the node to a pair and will use one two-input gates. Thus, $O_{N_2} - A_{N_2} = 1$.

[4]Number and nature of single literal inputs (n)
[5]Minimum number of two input gates required.

Appendix III

This section investigates the difference in the number of two-input gates of O and A ($O_{N_2} - A_{N_2}$), for the different input and output combinations that can occur at a grey node of the factored tree.

In all the following cases, the value of n specified describes the input configuration that occurred for A. In O, this value of n will change by ± 1.

CASE 1: $n = 0$. A will reduce the output to a single literal, using one two-input gate. O will get an input with $n = 1$, and will reduce it to a pair of literal output using one two input gate.

For this case: $O_{N_2} - A_{N_2} = 0$.

CASE 2: $n = 1$. A will reduce the output to a single literal, without using any two-input gate. O will: **a.** Get an input configuration, with $n = 2$, and will reduce it to a pair of literal output using zero two input gate.

For this case: $O_{N_2} - A_{N_2} = 0$.

b. Get an input configuration, with $n = 0$, and will reduce it to a pair of literal output using two two-input gate.

For this case: $O_{N_2} - A_{N_2} = 2$.

CASE 3: $n \geq 2$. **n is odd**. A will reduce the output to a single literal, without using any two-input gate. O will now get an input where n is even and will reduce the function of the node to a pair of literal form without using any two-input gate.

For this case: $O_{N_2} - A_{N_2} = 0$.

CASE 4: $n \geq 2$. **n is even**. A will reduce the output of this node to a pair of literal form, without using two-input gates. O will now get an input where n is odd and will reduce the function of the node one literal form, without using any two-input gate.

For this case: $O_{N_2} - A_{N_2} = 0$.

Appendix IV

This section finds the difference in the number of two-input gates that get synthesized by O and A ($O_{N_2} - A_{N_2}$), for the different input combinations that can occur at a black node of the factored tree.

CASE 1: Output form is single literal. Refer to Appendix I. If A gets an input with n being odd A will take no two-input gate to reduce it the output to a single literal form ($A_{N_2} = 0$). Now, O will get an input where is even, and will reduce it to a single literal output.

In this case, $O_{N_2} = 1$, and so $O_{N_2} - A_{N_2} = 1$

If A gets an input with n being even, then $O_{N_2} - A_{N_2} = -1$

CASE 2: Output form is a literal pair. If A gets an input with odd n, then A will take one two-input gate to reduce the output to a literal pair form ($A_{N_2} = 1$). Now, O will get an input where is even with either: **a.** $n = 0$ or **b.** $n > 0$

In the first case, $O_{N_2} = 2$, and so $O_{N_2} - A_{N_2} = 1$. If the reverse happens, i.e A gets an input with $n = 0$ and O gets an input, with odd n, then $O_{N_2} - A_{N_2} = -1$.

In the second case, $O_{N_2} = 0, \therefore O_{N_2} - A_{N_2} = -1$. If the reverse happens,

i.e A gets an input with even n, and $n > 0$ and O gets an input, with odd n, then $O_{N_2} - A_{N_2} = 1$.

We can now conclude, that for a black node, $O_{N_2} - A_{N_2} = \pm 1$.

Appendix V

Consider a MFNE factored form that is free of the patterns of the form $a(b+c)+bc$. The following properties hold for all the 3ITCs that implement such a factored form:

Lemma 2.6.1. *For a 3ITC, having N_2 two-input and N_3 three-input gates, implementing a MFNE factored form of ℓ literals:*

$$N_2 + 2N_3 = \ell - 1.$$

Proof: A three input gate reduces the number of literals in the factored form by 2, and a two input gate reduces the number of literals by 1. The function in it's factored form has ℓ inputs, and this is reduced to one output literal, by the network. Hence the statement of the lemma is true.

Theorem 2.6.1. *Any gate optimal 3ITC, implementing a MFNE factored form, has minimal number of two input gates, among all 3ITCs implementing the same factored form.*

Proof: Let N_2 and N_3 be the number of two input and three input gates respectively, in a gate optimal 3ITC that has K_{min} gates. Now, $N_2 + 2N_3 = \ell - 1$, where ℓ is the number of literals in the factored form [From theorem 1]. Consider another minimal gate network that has atleast one less two input

gate. Assume it has $N_2 - 2a$ two input gates[5], , where $a > 0$. Then, it should have $N_3 + a$ three input gates [Obtained by using Lemma 1].

The total number of gates in this second network is hence $N_2 - 2a + N_3 + a = N_2 + N_3 - a = K_{min} - a$. This second network has fewer gates than the optimal 3ITC, which is a contradiction.

Thus the assumption that there can be a network with fewer two input gates than the optimal gate network, is not true. Hence the gate optimal 3ITC has a minimum number of two input gates among all 3ITCs implementing the same factored form.

Corollary 1. *The threshold circuit that implements a MFNE factored form and has the least number of two-input gates is a gate optimal network.*

Proof: The proof of this corollary follows from the theorem.

[5]The difference between N_2's in any two gate optimal networks can't be odd. This claim can be proved as follows:

Let N_{2a} and N_{3a} are the number of two and three input gates for one optimal network, and N_{2b} and N_{3b} are the number of two and three input gates for another optimal network. From the first Lemma: $N_{2a} + 2N_{3a} = \ell - 1$ and $N_{2b} + 2N_{3b} = \ell - 1$.

Subtracting the two equations, and reordering, we get. $N_{2a} - N_{2b} = 2(N_{3b} - N_{3a})$ The right hand side is an even number and hence the left hand side must also be even.

Chapter 3

IDENTIFICATION AND SYNTHESIS OF THRESHOLD LOGIC FUNCTIONS USING COFACTORS

This Chapter introduces a novel approach to threshold function identification. As explained earlier, most of the existing approaches are based on solving linear programs, and sometimes exploiting specific properties of Boolean functions to quickly identify that the given function is *not* a threshold function, so as to avoid solving a linear program. In contrast, the methods presented here are based on efficient traversal of *decision diagrams* (Binary Decision Diagrams, and Max Literal Factor Trees), which allows not only to quickly identify that a function is a threshold function, but also to compute its minimal weight assignment. The basic threshold identification procedure is used to decompose a Boolean function into a network of threshold functions. It also allows for exercising limitations on certain gate parameters such as maximum fan-in, sum of weights and threshold making it more suitable for synthesis of realizable threshold circuits. Experiments done on benchmark circuits yielded circuits which on average have 23% fewer gates, and one less level when compared to the state-of-art method.

3.1 Notation and Definitions

This section contains the definition of various terms and the notation that will be used in the remainder of the Chapter.

1. ***Support Set:*** This is the set of all variables on which a given function F depends. It is denoted by S_F.

2. ***Cofactors:*** If F is a Boolean function then the positive (Shannon) cofactor of F with respect to a variable a, denoted by F_a, is the Boolean function obtained by evaluating F with $a = 1$. The negative cofactor of F, denoted by $F_{a'}$, is similarly defined. Note that S_F can also be defined as $\{a | F_a \neq F_{a'}\}$.

3. ***Don't Care variable:*** A variable d is said to be a *don't care* variable of a function F if and only if $F_{d'} = F_d$. Don't care variables do not belong to the support set of a function. With regard to threshold functions, don't care variables can be assigned a weight of 0, whereas non don't-care variables must always have a non-zero weight.

4. ***Max Literal:*** Let F be a Boolean function, given in the form of a sum of products in which no cube is contained in another (i.e. it is minimal w.r.t single cube containment). The max literal of a function F is the literal that occurs most frequently among the largest cubes in F. In the case of a tie, then the tie among those literals is broken by comparing their frequency among the next smaller size cubes.

   Example: Consider the function $F = ab + bc + cde$. The largest cubes are $\{ab, bc\}$, and since b occurs most frequently, b is the max literal of F. As another example, consider $F = abc + ad + ae + de$. The literals a, d and e each occur twice among the largest cubes $\{ad, ae, de\}$. Among (a, d, e), a occurs most often in the next largest cubes, namely, abc. Hence a is the max literal.

5. ***Function containment:*** F is contained in G, denoted by $F \subseteq G$, if and only if the onset of F is contained in the onset of G.
   $F \subseteq G \equiv F \Rightarrow G \equiv F' + G$.

6. $\succeq$***-Ordering:*** For a function F, if $F_{xy'} \supseteq F_{x'y}$, then x is *wavily greater than or equal* to y [73], and is denoted by $x \succeq y$. If $F_{xy'} \supset F_{x'y}$, then x is strictly wavily greater than y and is denoted by $x \succ y$. For a pair of variables x and y, if $x \succeq y$ and $x \preceq y$, then x is *wavily equal* to y and is denoted by $x \approx y$.

   Example: Consider the function $F = x + yuv$. Since $F_{xy'} = 1$ and $F_{x'y} = uv$, $F_{xy'} \supset F_{x'y}$, and $x \succ y$.

   An equivalent characterization of wavy ordering is that if $x \succ y$ then the number of minterms in which $x = 1$ is greater than the number of minterms in which $y = 1$ [73]. For a threshold function, $w_x > w_y$ implies $x \succeq y$, and $w_x = w_y$ implies $x \approx y$ [73]. It is also known that for a threshold function F, S_F can be totally pseudo-ordered using the $\succeq$-relation. For more details on these operators the reader is referred to [73].

3.2 Threshold Function Identification

Procedure isThreshold

In this section, we describe a procedure called **isThreshold**, that determines whether or not a given Boolean function F is a threshold function. Proofs of the properties on which the steps of the procedure are based are given in the Appendix.

The input to the algorithm is a binary decision diagram (BDD) $F = xF_x + x'F_{x'}$, where x is variable associated with node pointed to by F. If F is not a threshold function then **isThreshold** returns ϕ (nil), otherwise it will return a valid set of weights W and a threshold T. The algorithm makes use of the weights and threshold of F_x and $F_{x'}$ to determine the weights and threshold of F. This can be done because the algorithm partitions the problem of

determining the property of being a threshold function into two sub-problems of determining the same property of its cofactors.

The outline of the procedure in shown in Algorithm 8. In the algorithm, the weights are represented by a dictionary in which the variables are the keys and the values are their weights. $W[x]$ represents the weight of variable x. The algorithm uses a cache to store intermediate results.

[Terminal cases:] We first consider the terminal cases that are computed in lines 2 and 3. If $F = 1$, then its support set is empty, and by default the sum over an empty set is defined to be 0. Hence taking the threshold to be zero will result in the threshold inequality always being satisfied. Similarly if $F = 0$, then its the sum defined to be 0, it will not exceed a threshold 1.

In line 4 we check to see if the node was visited previously, and the weight and threshold pair were computed. If so, we return the *cached* values.

[Support Set and Cofactor Containment:] First, three necessary conditions for a threshold function are checked. The first two are simple. *Theorem 3.4.1* states that a necessary condition for F to be threshold is that the support set of either one of the cofactors of F is contained in the support set of the other cofactor. Line 7 checks this condition. The second necessary condition is given in *Theorem 3.4.2* which states that one of the cofactors must be contained in the other cofactor. Line 8 checks this condition.

[Check cofactors:] Another necessary condition for F to be a threshold function is that F_x and $F_{x'}$ are threshold functions (*Theorem 3.4.3*). This requires the recursive application of **isThreshold** to F_x and $F_{x'}$. This is accomplished by the code in lines 9 up to line 12. If either cofactor is not a threshold function, the procedure returns ϕ.

[Cofactors with same weights:] If control reaches line 13, then it has been determined that F_x and $F_{x'}$ are threshold functions. If the weights of F_x and $F_{x'}$ are the same, then F is a threshold function. In this situation, *Theorem 3.4.4* shows how the weights and threshold of F can be assigned. Let $F_x = [W, T_{F_x}]$, $F_{x'} = [W, T_{F_{x'}}]$. Then $F = [W \cup \{w(x) = T_{F_{x'}} - T_{F_x}\}, T_{F_{x'}}]$. This means that weight of the cofactor variable x of F is assigned the threshold of $F_{x'}$ minus the threshold of F_x, and the threshold of the F is assigned the threshold of $F_{x'}$. After computing the weight and threshold of F, it is cached and returned. The code from line 13 through line 18 accomplishes this.

[Cofactors with different weights:] We now arrive at a situation where the weights of F_x and $F_{x'}$ are not the same. It is known [73] that if F is a threshold function, then the variables in S_F must be ordered according to the wavy relation $\succeq$. This means that the variables in S_{F_x} and $S_{F_{x'}}$ must have the same wavy ordering as in F.

In [73] it is shown that for any weight assignment to a threshold function F, if $|w_x| > |w_y|$ then $x \succeq y$, and if $|w_x| = |w_y|$ then $x \approx y$. This condition is tested in line 20, and if the ordering of weights among the variables in F_x and $F_{x'}$ is different, then F is not a threshold function. Note that this is a pessimistic approach since if for some two variables a and b, if $|w_a| > |w_b|$ in F_x and $|w_a| < |w_b|$ in $F_{x'}$, then we would declare F to be non-threshold function, when in fact $a \approx b$ in both F_x and $F_{x'}$. Procedure **DiffWavvyOrdering** checks to see if the weights of cofactors have a different wavy ordering. This is on line 19 of **isThreshold**.

[Cofactors with same wavy ordering:] We know that if F is a threshold function then there exists an assignment of identical weights to F_x and $F_{x'}$.

Therefore, when the weights of F_x and $F_{x'}$ are not the same, but they both have the same $\succeq$-ordering, we attempt to *equalize* their weights. This final step is on line 20 of **isThreshold** and is performed by the procedure **tryEqualizeWeights**.

Procedure tryEqualizeWeights

Procedure **tryEqualizeWeights** attempts to equalize the weights of F_x and $F_{x'}$ in the following ways. First the support sets S_{F_x} and $S_{F_{x'}}$ of the two cofactors are checked for equivalence (line 3 of **tryEqualizeWeights**). In case they are different we attempt to *re-synthesize* the weights of one cofactor. This is performed by procedure **resynthesizeWeights** (line 4 of procedure **tryEqualizeWeights**). Note that non-equivalence means that one support set is fully contained in the other, as ensured by line 7 of **isThreshold**. If this re-synthesis step succeeds then the weights and threshold of F are returned. In case the procedure fails (i.e **resynthesizeWeights** returns ϕ), then the algorithm continues execution at line 25.

If the support sets of the two co-factors are the same, procedure **tryEqualizeWeights** checks if the weights assigned to F_x are a *valid* assignment for $F_{x'}$ and visa-versa (line 9 through line 24). The test of validity is performed by procedure **getValidThresholds**, which takes a weight vector W and a threshold function F and returns an interval of threshold values such that W and any value of a threshold in the interval serves as a valid weight-threshold assignment for F.

Suppose that W_x, which is a weight assignment to F_x, is a valid set of weights for $F_{x'}$, and the threshold interval returned by **getValidThresholds** is $[L,U]$. By Theorem 3.4.4, a valid threshold assignment to F will be the

Algorithm 8: Pseudo code of **isThreshold**(F)

```
1  isThreshold cacheLookup Cache isLeaf diffWavvyOrdering
   validWeights equalWeights tryEqualizeWeights alreadyComputed
2  (F) ;
   Input  : F is a pointer to a BDD or MLFT node.
   Output: φ if F is not a threshold function. Otherwise, if F is identified as a
           threshold function then it is a pair [W,T] where W is a dictionary
           with the care variables as keys and weights as values. T is the
           threshold. [W,T] is the weight-threshold assignment to F.

   // Terminal Cases
3  if F == 1 then return ([φ,0]) ;
4  if F == 0 then return ([φ,1]) ;

   // if result already cached, return it.
5  [W,T] = (F) ;
6  if [W,T] ≠ φ then return ([W,T]) ;
7  x = F.var ;

   // Check support set containment
8  if (S_Fx ⊈ S_Fx') or (S_Fx' ⊈ S_Fx) then return (φ) ;

   // Check cofactor containment
9  if F_x' ≰ F_x or F_x ≰ F_x' then return (φ) ;

   // Process cofactors
10 [W_x,T_x] = (F_x) ;
11 if [W_x,T_x] == φ then return (φ) ;
12 [W_x',T_x'] = (F_x') ;
13 if [W_x',T_x'] == φ then return (φ) ;

   // Both cofactors are Threshold
14 if (W_x,W_x') then
15     W = W_x ;
16     W[x] = T_x' − T_x ;
17     T = T_x' ;
18     ([W,T],F]) ;
19     return [W,T] ;
20 if (W_x,W_x') then return ((φ))
21 else return ( (F) ) ;
```

threshold of $F_{x'}$. The minimum valid threshold for $F_{x'}$ returned by **getValidThresholds** is L. Hence the threshold of F is set to L in line 15. Also,

Algorithm 9: Pseudo code of **tryEqualizeWeights**

getValidThresholds equalWeights tryEqualizeWeights cacheLookup cache resynthesizeWeights (F)
Input : F is a pointer to a BDD or MLFT node.
Output: A weight vector and a threshold or ϕ.

$x = F.var$
if $S_{F_x} \neq S_{F_{x'}}$ **then**
    $[W,T] =$ **(**F**)**
    if $[W,T] \neq \phi$ **then**
        ($([W,T]),F$**)**
        return $[W,T]$
else
    $[W_x,T_x] =$ **(**F_x**)**
    $[W_{x'},T_{x'}] =$ **(**$F_{x'}$**)**
    $[L,U] =$ **(**$W_x,F_{x'}$**)**
    if $[L,U] \neq \phi$ **then**
        $W = W_x$
        $W[x] = L - T_x$
        $T = L$
        ($([W,T]),F$**)**
        return $[W,T]$
    $[L,U] =$ **(**$W_{x'},F_x$**)**
    if $[L,U] \neq \phi$ **then**
        $W = W_{x'}$
        $W[x] = T_{x'} - U$
        $T = T_{x'}$
        ($([W,T]),F$**)**
        return $([W,T])$
    $[L_1,U_1] =$ **(**$W_{x'} + W_x,F_x$**)**
    if $[L_1,U_1] \neq \phi$ **then**
        $[L_0,U_0] =$ **(**$W_{x'} + W_x,F_{x'}$**)**
        if $[L_0,U_0] \neq \phi$ **then**
            $W = W_x + W_{x'}$
            $W[x] = L_0 - U_1$
            $T = L_0$
            ($([W,T]),F$**)** **return** $[W,T]$
return ϕ

Algorithm 10: Algorithm to determine if the given weights W is valid for the BDD/MLFT F (threshold function).

```
1  getValidThresholds
2  (W, F) ;
   Input  : F is a pointer to a BDD or MLFT node. W is a dictionary. The
            keys are variables in the support set of F and the values are the
            weights.
   Output: An interval indicating the range of valid thresholds if F is a
            threshold function, otherwise False.
3  if F == 1 then return [−∞, Σ_{x|W[x]<0} W[x]] ;
4  if F == 0 then return [Σ_{x|W[x]>0}(W[x]) + 1, ∞] ;
5  if | Σ_{x∈W∩DC(F)} W[x] |  ≥  min{| W[y] |, y ∈ S_F} then
6      return φ
7  if ∃x ∈ S_F ∋ |W[x]| ≤ 0 then
8      return φ
9  x = F.var ;
10 I_1 = (W − W[x], F_x) ;
11 I_0 = (W − W[x], F_x') ;
12 return (I_1 + W[x]) ∩ I_0
```

by Theorem 3.4.4, the weight of the variable at node F is the $T_{x'} - T_x = L - T_x$. This is performed on line 14. If W_x is not a valid set of weights for $F_{x'}$, then **getValidThresholds** will return ϕ, and **tryEqualizeWeights** checks if $W_{x'}$ is a valid set of weights for F_x. This situation is similar and is shown in lines 18 through 24.

If the cofactor weights have not been equalized thus far, then **tryEqualizeWeights** checks if the sum $W_x + W_{x'}$ is valid for both F_x and $F_{x'}$. In checking if the sum of weights is valid for both, two intervals are returned: $[L_1, U_1]$ (line 25) when checking if the sum is valid of F_x, and $[L_0, U_0]$ (line 27) when checking if the sum is valid for $F_{x'}$.

If neither is empty, then F is a threshold function and a weight and threshold can be assigned to F. The weights will simply be $W_x + W_{x'}$. To determine the threshold of F, recall from the description of weight-threshold

Algorithm 11: Algorithm to resynthesize weights to accommodate don't cares.

```
1  resynthesizeWeights cacheLookUp getValidThresholds
2  (F) ;
   Input  : F is a pointer to a BDD or MLFT node.
   Output: A weight and threshold pair [W, T].
3  x = F.var ;
4  [W_x, T_x] = (F_x) ;
5  [W_x', T_x'] = (F_x') ;
6  if S_Fx ⊃ S_Fx' then
7      W_DC = W_x \ W_x' ;
8      K = 1 + Σ_{y∈W_DC} |W_x[y]| ;
9      W_x'^new = K * W_x' ∪ W_DC ;
10     T_x'^new = K * T_x' + Σ_{y∈S_Fx' | W_x'[y]<0} W_x'^new[y] ;
11     [L, U] = (W_x'^new, F_x) ;
12     if [L, U] = φ then return φ ;
13     W = W_x'^new ;
14     W[x] = T_x'^new − U ;
15     return [W, T_x'^new] ;
16 else
17     W_DC = W_x' \ W_x ;
18     K = 1 + Σ_{y∈W_DC} |W_x'[y]| ;
19     W_x^new = K * W_x ∪ W_DC ;
20     T_x^new = K * T_x + Σ_{y∈S_Fx | W_x[y]<0} W_x^new[y] ;
21     [L, U] = (W_x^new, F_x') ;
22     if [L, U] = φ then return φ ;
23     W = W_x^new ;
24     W[x] = L − T_x^new ;
25     return [W, L]
```

assignment above, the weight that is assigned to the cofactor variable x of F, namely $W[x]$, is $T_{x'} - T_x$, and the threshold is $T = T_{x'}$. In procedure **tryEqualizeWeights**, the threshold interval for F_x is $[L_1, U_1]$, and for $F_{x'}$ it is $[L_0, U_0]$. Hence the least value of T will be L_0, the least value for $W[x]$ is $L_0 - U_1$. These computations are shown in lines 30 and 31.

Procedure getValidThresholds

In this section the procedure that checks whether or not a given set of weights is a valid assignment to a given function is described. The procedure that performs this is **getValidThresholds**, and its pseudo-code is shown in Algorithm 10. Procedure **getValidThresholds** recursively checks if a given set of weights are valid for a function. It returns an interval of valid threshold values for the given weights. An empty interval means that the weights are not valid. Note that two operations on intervals are used. Addition of a single value to an interval is defined as $[L,U]+c=[L+c,U+c]$. The intersection of two intervals is an interval representing their overlap.

Consider the terminal case of $F=1$, and w.l.o.g let $W=(w_1,w_2,\ldots,w_i,-w_{i+1},-w_{i+2},\ldots-w_n)$. Consider the inequality

$$w_1x_1+w_2x_2+\cdots-w_{i+1}x_{i+1}-w_{i+2}x_{i+2}-\cdots-w_nx_n\geq T.$$

Clearly if $F=1$, then the above inequality must evaluate true for all values of x. This will be the case if T is less than the minimum possible value of the expression on the LHS, which will be the sum of all negative weights. Hence the valid interval for T is $\left[-\infty,\sum_{x|W[x]<0}W[x]\right]$. This is shown line 2. The reasoning for the case of $F=0$ is similar and is shown on line 3.

The next test on line 4 ensures that the sum of the absolute weights of *don't care* variables is less than the minimum of the absolute weights of all the *care* variables (Lemma 3.4.2). The second test on line 6 ensures that the absolute weight of every *care* variable is strictly non-zero (Lemma 3.4.3).

After having dispensed with the terminal cases, **getvalidThresholds** is recursively applied to each cofactor. The result will be two intervals I_1 and I_0,

corresponding to the application of **getValidThresholds** to F_x and $F_{x'}$. Since we are checking for the validity of a given set of weights, the weight of the cofactor variable x of F, denoted by $W[x]$ is already given. Recall that the weight assignment in **isThreshold** computes this value as $W[x] = T_{x'} - T_x$. Therefore, $T_{x'} = W[x] + T_x$. Interval I_1 represents the values for T_x, and I_0, the values for $T_{x'}$. Since both must satisfied, the interval of thresholds associated with F must be $(I_1 + W[x]) \cap I_0$. This is the interval returned at line 11 by **getValidThresholds**.

Procedure resynthesizeWeights

What remains is an explanation of the procedure **resynthesizeWeights**. This relies on the fact that the weights of a threshold function depend on the set of don't care variables. For instance, consider the function $F = ab \vee c$. Then $S_F = \{a, b, c\}$. If no don't care variables are specified, then one valid weight assignment for F would be $W = [w_a = 1, w_b = 1, w_c = 2]$ and $T = 2$. This corresponds to the inequality $F \equiv a + b + 2c \geq 2$. Now consider the same function, but with two variables d and e specified as don't cares with weights $w_d = -5$ and $w_e = 3$. In this case, one set of valid weights would be $W = [w_a = 9, w_b = 9, w_c = 18, w_d = -5, w_e = 3]$ and $T = 13$, which corresponds to the inequality $F \equiv 9a + 9b + 18c - 5d + 3e \geq 13$. Thus don't care variables affect the weight-threshold assignment.

As stated before, procedure **resynthesizeWeights** is called when $S_{F_x} \neq S_{F_{x'}}$. For definiteness, suppose $S_{F_x} \supset S_{F_{x'}}$ (line 5 of **resynthesizeWeights**). In this case, some variables in F_x are don't care variables in $F_{x'}$. In such a situation, the care variables in $F_{x'}$ can be assigned weights conditional on the weights of its don't care variables that were

assigned values in F_x. Lemma 3.4.2 states that the minimum weight of a care variable must be at least one more than the sum of the magnitudes of the weights of the don't care variables. This quantity is used to scale the weights of the care variables in $F_{x'}$ (line 8). This scaling is done by multiplying each weight and the threshold by a constant. The scaled weights and the corresponding threshold (line 9) will always be valid for $F_{x'}$. Next, a test is made to check if these new weights $W_{x'}^{new}$ are valid for F_x using **getValidThresholds** (line 10). If they are, then both cofactors now have the same set of weights, and as before, the new weights and threshold of F are calculated and returned.

Examples of BDD based Threshold Identification

In this section we present three examples that exercise different parts of the procedures. We first consider a simple (threshold) function $F = ab + bc + ca$. This function exercises the core steps of the procedure **isThreshold**, and serves as a starting point for the discussion of more complex examples.

The variable ordering in the BDD representing F, shown in Figure 3.1, is $a < b < c$. The initial dictionary representing variables and weights would be $F.W = [a:?, b:?, c:?, T:?]$. Note that the tests for support set containment in **isThreshold** (line 7) and cofactor containment (line 8) always succeed for this example. The following is the sequence of computations that take place when **isThreshold** is applied to F.

1. **isThreshold** recursively descends down the BDD to the leaf nodes 1 and 0. The weights and threshold returned by 1 and 0 are $[\phi, 0]$ and $[\phi, 1]$, respectively (Figure 3.1(a)).

2. At node c, both of its cofactors are threshold functions. Their weight vectors (e.g. ϕ) are the same. Hence control reaches line 14. The result is $w_c = 1$, $T_c = 1$. This stage is shown in Figure 3.1(b).

3. At node b (which is a's left child $b + c$), both of its cofactors are threshold functions. However $W_b = \phi$ (b's 1-cofactor) and $W_{b'} = [w_c = 1]$. The test on line 13 fails, and control reaches line 19, where the test fails by default. That is, the two cofactors have the same wavy ordering. Hence an attempt is made to equalize the weights of b's cofactors (line 20).

4. In procedure **tryEqualizeWeights**, $F = b + c$, $x = b$, $S_{F_b} = \phi$, and $S_{F_{b'}} = w_c$. On line 3, we check if the support sets S_{F_b} and $S_{F_{b'}}$ are equal. Since they are not the same, procedure **resynthesizeWeights** is called with $F = b + c$ as the argument.

5. In procedure **resynthesizeWeights** $x = b$, $W_b = \phi$, $T_b = 0$, $S_{F_{b'}} = \{c\}$, and $S_{F_b} = \phi$. Since $S_{F_{b'}} \supset S_{F_b}$, control reaches line 16. Executing lines 16 through 19, results in

   $W_{DC} = [w_c = 1], K = 2, W_b^{new} = 2 * \phi + [w_c = 1] = [w_c = 1] \Rightarrow T_b^{new} = 0.$

   Note that the leaf node $F_b = 1$ now has a weight-threshold pair equal to $[W_b^{new}, T_b^{new}] = [w_c = 1, 0]$, because c which is a don't care variable for the leaf node, is now added to its leaf node. This represents the inequality $c \geq 0$, which is always satisfied, and hence still represents the function 1.

   In line 20, we check if the weight $[w_c = 1]$ is valid for $F_{b'} = c$. It obviously is, and the interval of thresholds returned is $[L, U] = [1, 1]$. Hence the weight and threshold pair for node b is $[w_b = 1, w_c = 1, T = 1]$, which represents the function $b + c$. This stage is shown in Figure 3.1(c).

6. Returning to procedure **tryEqualizeWeights**, control reaches line 5, where the weights $[w_b = 1, w_c = 1; T_b = 1]$ are returned back to **isThreshold** (see Figure 3.1(c)).

7. The negative cofactor of a, function bc, is processed by **isThreshold** similarly and the weight-threshold of $[w_b = 1, w_c = 1; T_b = 2]$ is assigned when control returns to line 11 of **isThreshold**.

8. After having processed a's cofactors, both of which are now threshold functions, control returns to line 13 of **isThreshold**. Here $x = a$, and the weight and threshold of a's positive cofactor are $W_a = [w_b = 1, w_c = 1]$ and $T_a = 1$, and that of its negative cofactor are $W_{a'} = [w_b = 1, w_c = 1]$ and $T_{a'} = 2$. Since $W_a = W_{a'}$, the test on line 13 succeeds. In lines 14-16, w_a and T_a are assigned the values 1 and 2 respectively. In line 18 the weights and threshold of $F \equiv [w_a = 1, w_b = 1, w_c = 1; T_a = 2]$, is returned (Figure 3.1(d)).

To further illustrate the **tryEqualizeWeights** procedure two more examples are presented. Consider the function $F = ab + acd + ace + bcd + bce$. The BDD of this function is shown in Figure 3.2(a). During the execution of **isThreshold**, nodes F_a and $F_{a'}$ have different weights: $W_a = [w_b = 3, w_c = 2, w_d = 1, w_d = 1] \neq W_{a'} = [w_b = 2, w_c = 2, w_d = 1, w_e = 1]$. The control reaches line 20 of the **isThreshold** procedure and **tryEqualizeWeights** is called. In the function **tryEqualizeWeights**, since the support sets of F_a and $F_{a'}$ are equal, control passes to line 9. The call to **getValidThresholds** on line 11 checks if weights W_a are valid for $F_{a'}$. The edges of Figure 3.2(a) are annotated with the intervals returned by recursive calls to **getValidThresholds**. The final result returned by

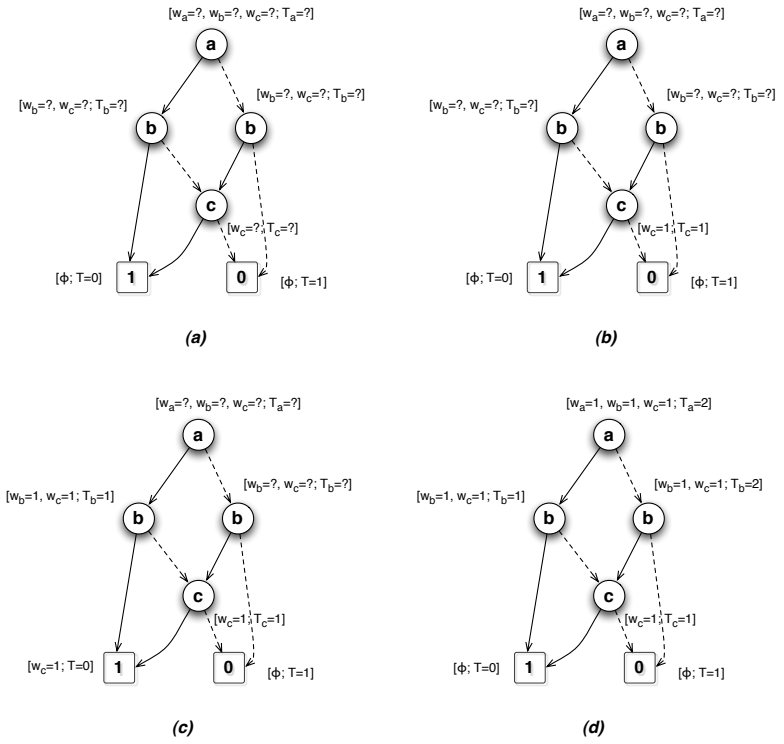


Figure 3.1: Execution of procedure isThreshold on BDD of $F = ab + bc + ca$. Solid edges are 1-cofactors, dashed edges are 0-cofactors.

getValidThresholds is $([3,3]+3) \cap [5,\infty] = [6,6]$. Since $[6,6] \neq \phi$, weights of function F is assigned to be $[w_a = 3, w_b = 3, w_c = 2, w_d = 1, w_d = 1]$ and the threshold T is assigned to be 6. This tuple is returned by **tryEqualizeWeights** which in turn is the return value of **isThreshold**.

The last example we consider is the function $F = abc + abd + abe + acd + ace + ade + bcd + bce + bde$. The BDD of this function is shown in Figure 3.2(b). During the execution of **isThreshold** procedure nodes F_a and $F_{a'}$ have different weights $(W_a = [w_b = 1, w_c = 1, w_d = 1, w_d = 1] \neq W_{a'} = [w_b = 2, w_c = 1, w_d = 1, w_e = 1]$. On line 20 of **isThreshold**, **tryEqualizeWeights** is called. Since the support-sets of both co-factors are identical, control reaches line 9 of

tryEqualizeWeights. The tests to check if either cofactor's weights is valid for the other fails (lines 11 and 18). The final test is to check if the sum of the cofactor weights is valid for both cofactors. This test succeeds and the final weight-threshold for function F $[w_a = 3, w_b = 3, w_c = 2, w_d = 2, w_e = 2; T = 7]$ is returned.

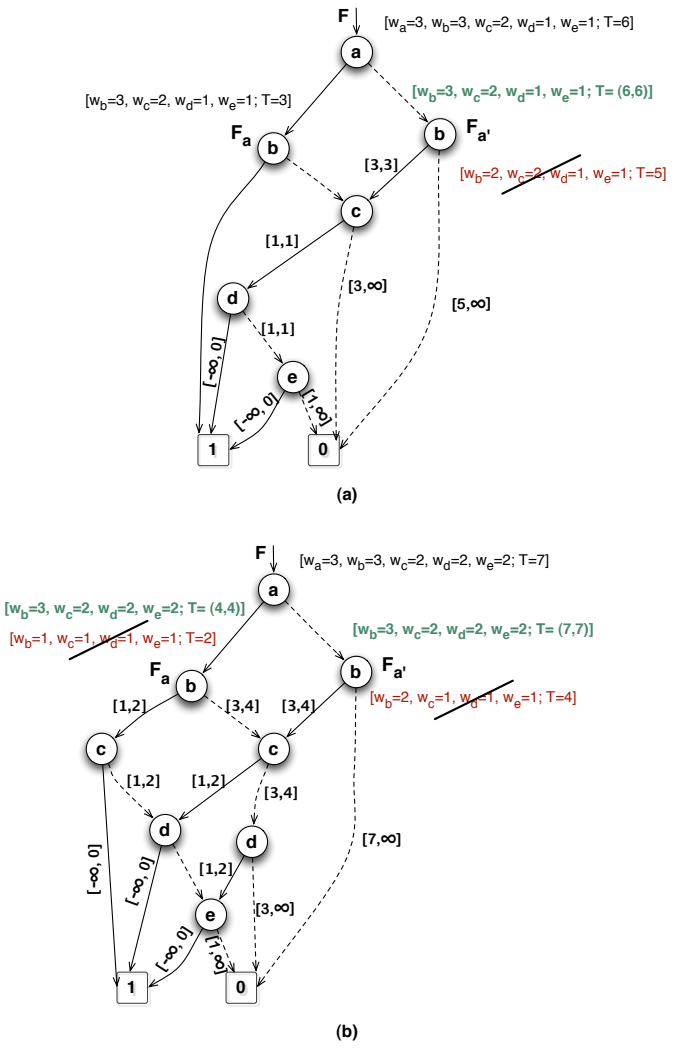


Figure 3.2: Execution of procedure isThreshold on BDD of (a) $F = ab + acd + ace + bcd + bce$, (b) $F = abc + abd + abe + acd + ace + ade + bcd + bce + bde$.

Effect of Variable Ordering of a BDD

The ordering of variables in the BDD can affect the outcome of the threshold identification procedure. Figure 3.3 shows the result of applying **isThreshold** to the function $G = abc + abd + abe + acd + bcde$ with the ordering $a < b < c < d < e$. The procedure **isThreshold** correctly identifies the function as a threshold function with a weight-threshold assignment $[w_a = 4, w_b = 3, w_c = 2, w_d = 2, w_e = 1; T = 8]$. Figure 3.4 shows the same function with the ordering $b < a < c < d < e$. This ordering requires **isThreshold** to repeatedly attempt to equalize the weights, which eventually fails for the cofactors of the root node b. The original weight-threshold assignment of b's cofactors are $W_b = [w_a = 2, w_c = 1, w_d = 1, w_e = 1; T = 3]$ and $W_{b'} = [w_a = 1, w_c = 1, w_d = 1, w_e = 0; T = 2]$. The support sets aren't the same (e is a don't care variable in $G_{b'}$). Hence procedure **resynthesizeWeights** is called. In **resynthesizeWeights**, the weight-threshold vector computed for $G_{b'}$ is $[w_a = 1, w_c = 1, w_d = 1, w_e = 0; T = 2]$. However this weight vector does not have a valid threshold for G_b. Therefore procedure **resynthesizeWeights** fails and the control passes to line 25 of **tryEqualizeWeights**. Finally, the test of whether $W_a + W_{a'}$ is valid for both cofactors, also fails. Consequently, procedure **isThreshold** returns ϕ, declaring that G is not a threshold function.

Max Literal Factor Tree: An Alternate Structure

We now describe an alternate data structure, which is referred to as a *Max Literal Factor Tree* (MLFT). An MLFT will be seen to be more efficient and accurate for identifying threshold functions. An MLFT is a decision diagram similar to a BDD, and the algorithms presented in the previous section remain

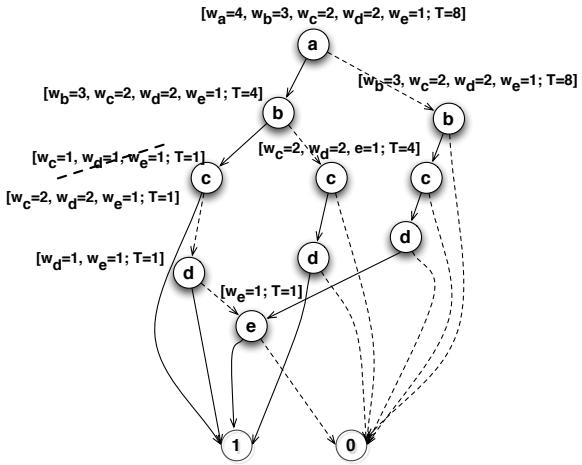


Figure 3.3: Result of *isThreshold* on $G = abc + abd + abe + acd + bcde$ with ordering $a < b < c < d < e$.

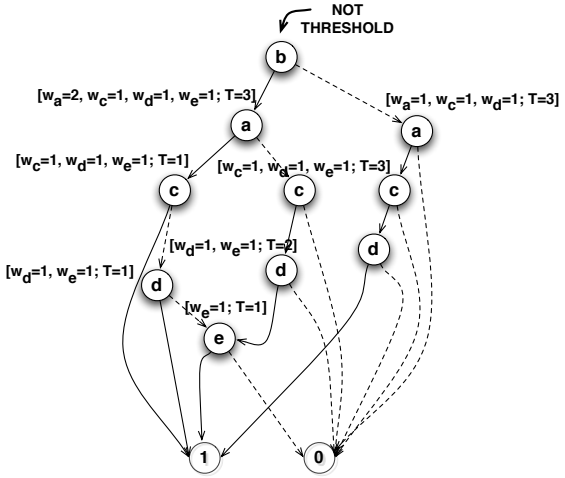


Figure 3.4: Result of *isThreshold* on $G = abc + abd + abe + acd + bcde$ with ordering $b < a < c < d < e$.

exactly the same when operating on a MLFT instead of a BDD. There are several important characteristics of an MLFT that should be noted.

1. An MLFT is a factor tree that can be constructed from a sum of products (SOP) representation that is minimal with respect to single cube containment.

2. Unlike a BDD, no particular ordering of variables can be associated with an MLFT of an arbitrary function.

3. However, a well defined ordering of variables, which we refer to as a *max literal* (ML) ordering, can be associated with an MLFT of a threshold function F. The BDD of a threshold function with its variables ordered according to the ML ordering is the same as an MLFT.

The construction of an MLFT is as follows. First, without loss of generality, we assume that F is a positive unate function and F is given in the form of a SOP that is minimal with respect to single cube containment. The result of dividing F by a literal x is a pair of functions (Q,R) such the $F = xQ+R$. Here, x is the divisor, Q is the quotient and R is the remainder. The MLFT of F is obtained by repeated *algebraic* division of F using max literals as divisors at each step of the division. In an MLFT, the factored form $F = xQ+R$ is represented by binary tree, where the root node is labeled by x, and its left and right children are the MLFT of Q and R respectively. The process stops when both quotient and remainder are the constant functions 1 or 0. **Note:** Unlike a BDD, the ML ordering of Q and R might be different, and hence no particular *ordering* can be associated with an MLFT of a non-threshold function.

Consider $F = ab+c$. The ML ordering is $c < a < b$. The MLFT of F is obtained by the following factorization: $F = c(1)+[a(b)+0]$. Figure 3.5 shows the MLFT of F. This is the same as the BDD for the ordering $c < a < b$ because F is a threshold function.

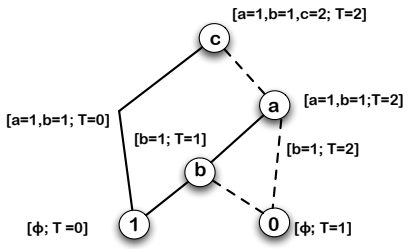


Figure 3.5: MLFT of $f = ab + c$ with ordering $c < a < b$. Same as a BDD.

Now consider the function $G = ab + ade + bde + efg$, which is not a threshold function. Its MLFT is obtained by the following factorization.

$$\begin{aligned} G &= ab + ade + bde + efg \\ &= a[b(1) + de] + bde + efg \\ &= a[b(1) + d[e(1) + 0]] + [e[bd + fg] + 0] \end{aligned}$$

Figure 3.6 shows the MLFT of G, which is not a BDD.

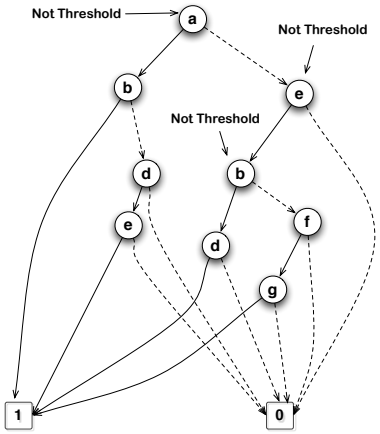


Figure 3.6: MLFT of a non-threshold function $G = ab + ade + bde + efg$ is not the same as a BDD.

The example threshold function $G = abc + abd + abe + acd + bcde$ given in Section 3.2 was correctly identified by **isThreshold** when the BDD was ordered according to an ML ordering. The ML ordering is of particular

significance because for a threshold function, the ML ordering[1] is consistent with a wavvy ordering.

We noticed that a threshold function in the form of an MLFT (which is the same as a BDD with an ML ordering) was always correctly identified as such by procedure **isThreshold**. This claim was tested by examining all permutations of all threshold functions of five variables (i.e. 120 permutations for each threshold function). There are 92 representative threshold functions of five variables listed in [73] (all others are can be mapped to one of the 92 functions). Table 3.1 shows the results of applying **isThreshold** to all permutations of each of the 92, five variable functions. It indicates the number of functions and the fraction of their 120 permutations that were correctly identified as threshold functions. For instance, for 75 of the 92 functions, all 120 permutations resulted in a correct identification. On the other hand, the ML ordering resulted in correctly identifying all 92 functions as threshold functions.

	% of orderings that resulted in correct identification			
	100%	80%	60%	40%
# of 5 variable functions out of 92	75	5	6	6

Table 3.1: Results of applying isThreshold on all 120 orderings of all 92, 5 input functions

The fact that an MLFT of a threshold function results in a ML ordering of its support set is one of the main reasons that an MLFT is this successful in identifying threshold functions. Even if a function F is not a threshold function, some cofactor of F, say G, might be a threshold function. In such a case, the repeated factoring of F by the max literals will result in the MLFT of G being

[1] ML ordering of a threshold function is the same as ordering by Chow parameters [73]

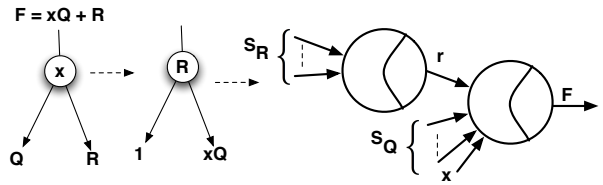
quickly identified as a threshold function since the ordering of the variables in G's support set would be consistent with a wavvy ordering. In a BDD, procedure **isThreshold** may not identify the sub-tree representing G as a threshold function.

3.3 Synthesis of Threshold Networks

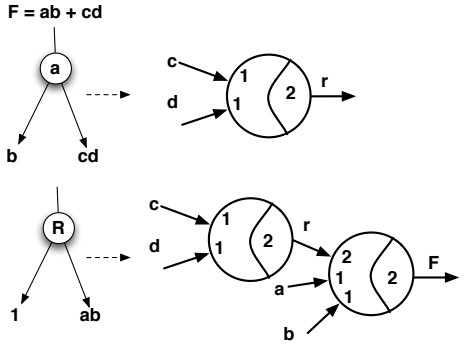
The first time that a node F is determined to be a non-threshold function, it will be with both of its cofactors being threshold functions. This is because, the algorithm proceeds bottom-up and leaf node is always a threshold function. At this point, one or both of the cofactors can be *substituted* by a literal which will represent the output of a threshold gate. The rules for substitution vary for an MLFT and a BDD.

Consider an MLFT node $F = xQ + R$, where Q and R are threshold functions and F is not. Since xQ is also a threshold function, we can substitute either xQ or R. Suppose R is substituted by a literal r. This means that the function R is implemented as a threshold gate and the literal r denotes its output. Therefore the new MLFT would be $F = r + xQ$, which is a threshold function whose support set is $\{r,x\} \cup S_Q$. After this transformation, the threshold identification procedure can proceed toward the root. Figure 3.7 shows the transformation, and its application to the function $F = ab + cd$. $R = cd = [w_c = 1, w_d = 1; T = 2]$, $F = r + ab = [w_r = 2, w_a = 1, w_b = 1; T = 2]$. Similarly, if xQ is implemented by a threshold gate, whose output is denoted by the literal h, then $F = h + R$ is a threshold function.

Now consider the function $G = e(ab + cd)$. In this case, we let $x = ab + cd$ and $G = ex$. From Figure 3.7 we see that $(ab + cd)$ requires two threshold gates. Figure 3.8 shows two possible implementations for G. In



(a) MLFT substitution rule: selecting the 0-zero cofactor

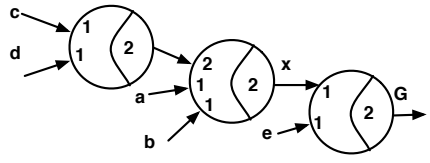


(b) MLFT substitution example

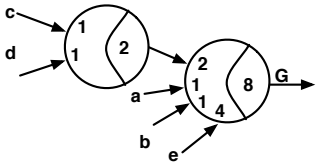
Figure 3.7: Decomposition of a non-threshold MLFT. (a) General rule, (b) Example $F = ab + cd$

.

Figure 3.8(a) x and e are fed as inputs to a third threshold gate, and in Figure 3.8(b) e it is merged into the threshold gate that produced x. As is apparent, these implementations differ in the number of gates and levels, the maximum fanin, and maximum weight and threshold.



(a)



(b)

Figure 3.8: Possible implementations of $G = e(ab + cd)$.

The substitution rules in the case of BDDs are similar. Consider the BDD node $F = xF_x + x'Fx'$, and assume that F_x and $F_{x'}$ are threshold functions, but F is not a threshold function. The substitution can be done in one of three ways: (1) implement $y = F_x$ and $z = x'F_{x'}$ with two separate threshold gates with node $F = xy + z$ becoming a new threshold function $F = [w_x = 1, w_y = 1, w_z = 2; T = 2]$, or (2) implement $y = xF_x$ and $z = F_{x'}$ with two separate threshold gates with node $F = x'z + y$ becoming a new threshold function $F = [w_{x'} = 1, w_y = 2, w_z = 1; T = 2]$, or (3) implement $y = xF_x$ and $z = x'F_{x'}$ by two threshold gates with $F = y + z$ becoming a new threshold function $F = [w_y = 1, w_z = 1; T = 1]$. As with MLFTs, these choices lead to differences in gates, levels, weights and thresholds.

The choice of the substitution will impact the overall area, power and performance of the network. In fact, with certain circuit architectures of threshold gates such as those that use differential logic, the maximum weight and threshold values have a strong impact on the reliability of the gate.

Due to the absence of a physical realization of a threshold gate, or models of area, delay and power of threshold gates, our choice for the substitution is based on reducing the number of gates.

Although in this work synthesis has been discussed in detail, the core contribution of the work can be used to enhance other methods used in designing threshold circuits. Detection of threshold function is an often recurring problem in threshold CAD and this technique can be plugged with existing approaches for detecting a threshold function. For example threshold circuits have been used to improve the characteristics of asynchronous circuits [50, 51]. The approach in [51] uses merging of adjacent cells in an

unoptimized netlist, as an optimization technique. Since the functions considered are threshold functions the method proposed in this Chapter can be used along with the techniques proposed in [51] to detect which mergers yield valid threshold functions.

3.4 Experimental Results

Synthesis of Threshold Circuits

The algorithms described in this Chapter were tested on the MCNC suite of benchmark circuits [25]. We present results based on BDDs and MLFTs. Figure 3.9 shows the steps that were followed in synthesizing a threshold network. Starting from a circuit specification in the form a BLIF [28] or Verilog netlist, a multi-level logic synthesis tool, such as SIS [28], is used to generate a Boolean network. This is a directed acyclic graph with each node representing a multi-input, single output Boolean function. Before carrying out the threshold logic synthesis method described, a BDD or MLFT of each node in the Boolean network was constructed with the local inputs as its support set. Hence the original network structure is preserved, with the network's nodes being replaced with a threshold network.

When constructing a BDD of a node in the Boolean network, no particular ordering is specified. We let the decision diagram package [65] select the order to achieve compact representations. The MLFTs of the node functions were computed by first flattening (i.e. multiplying out the factored form to produce sum of products (SOP)) the factored form obtained from the multi-level synthesis step shown in Figure 3.9, and then making the SOP minimal with respect to single cube containment. The resulting SOP was repeatedly factored using the max literal as the divisor.

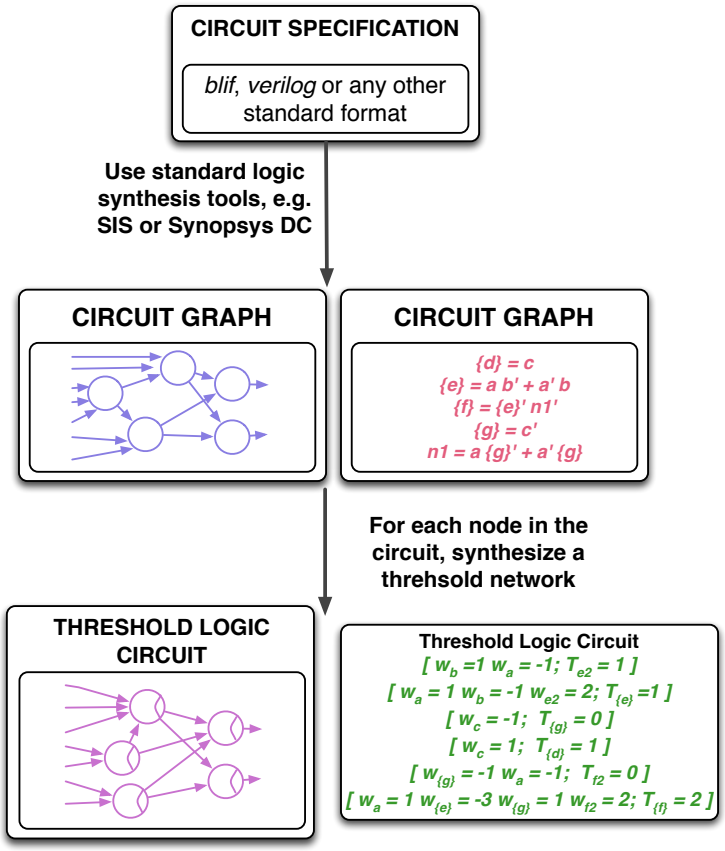


Figure 3.9: (a) Generic Synthesis Flow (b) Example of synthesis of $b1$ benchmark circuit.

We compare the method presented in this Chapter with results reported in [98]. The approach described in [98] also starts from a gate-level netlist, and selects sub-circuits whose functions are unate and checks whether or not the function is a threshold function by attempting to satisfy a set of linear inequalities derived from the truth table.

A comparison of the results of applying the method described using BDDs with the results in [98] is shown in Figure 3.10. Due to the large number of circuits and many differences in their characteristics, the results were grouped into four categories based on the number of gates. The bins were determined so as to have approximately an equal number of circuits.

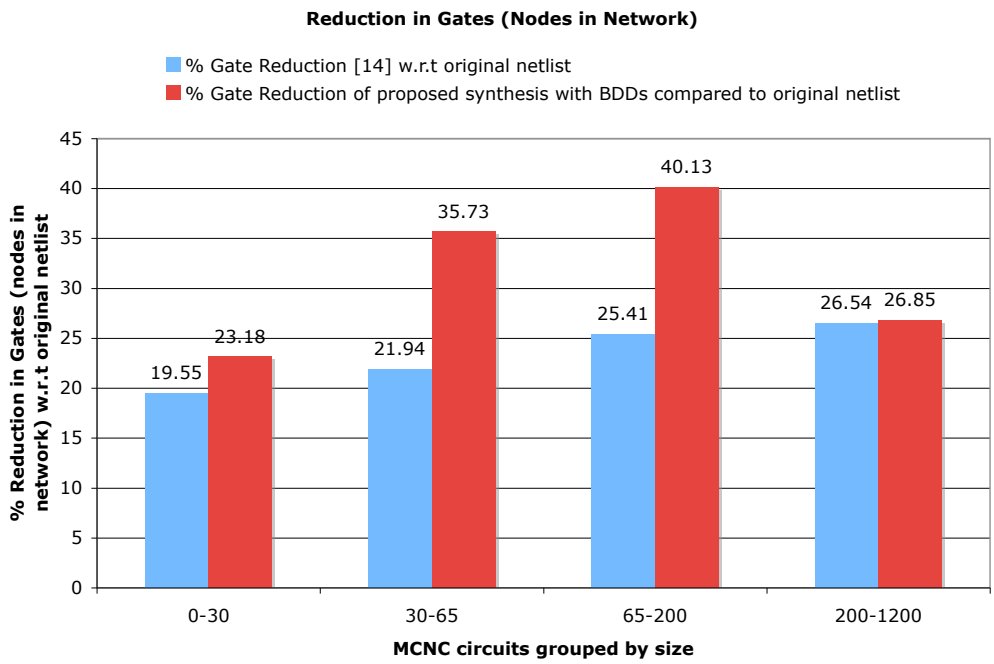


Figure 3.10: Percentage gate reduction obtained by [98] versus proposed method using BDDs.

As expected, both the proposed method and the one in [98] resulted in fewer gates than the original logic network. Over the set of all circuits, [98] showed an average of 23% reduction in the total number of gates and the reduction of approximately 1.1 levels on the average, whereas the proposed method using BDDs resulted in an average of 31% reduction in the number gates.

Figure 3.11 shows the results when MLFTs were used. The results are much better when compared to those based on BDDs. Using MLFTs, the proposed method resulted in an overall average of 36% reduction in the number of gates and approximately 2.5 fewer levels.

The threshold network synthesis procedure using MLFT was implemented in Python, and executed on a laptop computer with a single core.

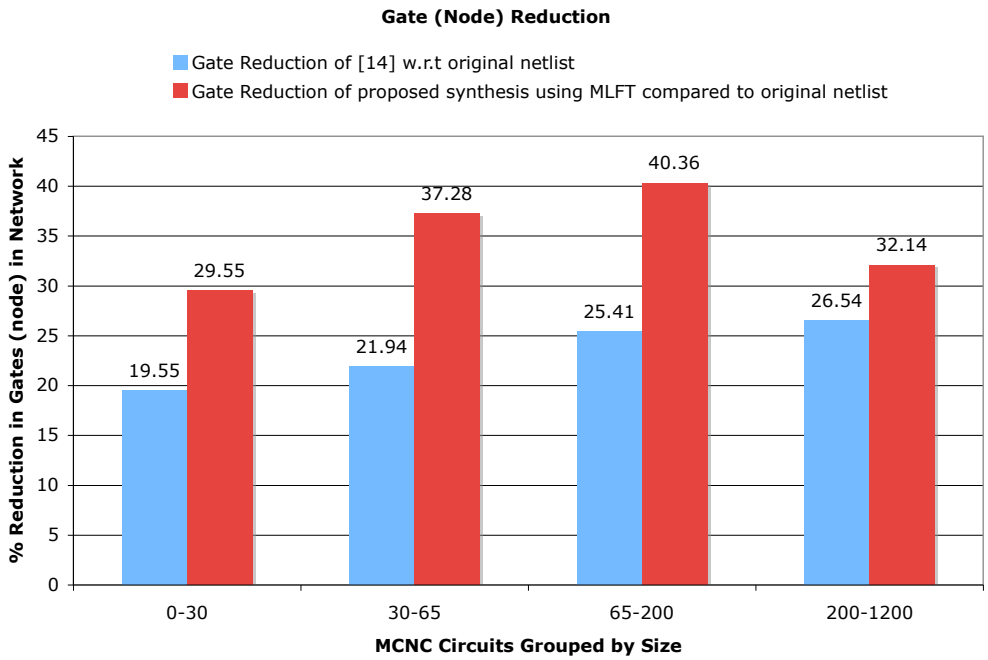


Figure 3.11: % Gate reduction obtained by [98] versus proposed method using MLFT.

The execution time ranged from 2.5 seconds to 24 seconds. The Python implementation would easily be two to three orders of magnitude slower than a optimized C implementation. The BDD based procedure was implemented in C++ using an existing BDD package [65]. In this case, the execution times were less than a second for each circuit.

A comparison of the proposed approach with those in [3, 88] is not meaningful for the following reasons. The method described in [88] is essentially a two level synthesis procedure. Multi-level circuits are obtained only when the fanin exceeds a certain limit or if the computation time becomes excessive (in which case they use SIS for preprocessing). The motivation for the work in [88] is the study of neural network architectures (since the threshold element and the Perceptron are functionally identical), and hence gates with of fanins as large 86 are produced. This may not be an issue for

neural network implementations, however for most logic circuits such high fanins are not feasible. Since the vast majority of circuits generated are two level implementations, no meaningful comparison can be made with a multi-level synthesis approach. The results reported in [3] do not include MCNC circuits. Moreover, their work and the one described herein are qualitatively different both in terms of approach and computational efficiency. In [3] an extensive search for feed forward implementation is done which results in one gate per level. This method takes minutes to synthesize even simple circuits of few inputs and outputs. Moreover the number of levels is equal to the number of gates.

Appendix

Lemma 3.4.1. *If $F(X)$ is a threshold function and has a weight threshold assignment $[\{w_1, \cdots w_{a-1}, w_a, w_{a+1}, \cdots, w_n\}; T_F]$, then for $F(X; a \rightarrow a')$, $[\{w_1, \cdots w_{a-1}, -w_a, w_{a+1}, \cdots, w_n\}; T_F - w_a]$ is a feasible weight threshold assignment.*

Proof. See citation [73] *pg 58*. □

Definition 1. *The positive form of weight assignment of a threshold function is the weight assignment obtained by the repeated application of Lemma 3.4.1, such that all the input weights are positive.*

Lemma 3.4.2. *Let F be a threshold function. Then for any weight-threshold assignment $[W, T]$ to F,*

$$\sum_{x \notin Supp(F)} |w_x| < \min_{y \in Supp(F)} \{|w_y|\}. \tag{3.1}$$

Proof. (By contradiction) Let **D** denote the set of don't care variables of F. Assume the contrary that

$$\sum_{x \in \mathbf{D}} |w_x| \geq |w_y|, \text{ for some } y \in Supp(F). \tag{3.2}$$

Let P_y be a prime implicant of F that contains y. Therefore

$$\sum_{v \in P_y} w_v \geq T \equiv \sum_{\substack{v \in P_y \\ v \neq y}} w_v + w_y \geq T. \tag{3.3}$$

Combining Equations (3.2) and (3.3), we obtain

$$\begin{aligned} \sum_{v \in P_y} w_v + \sum_{x \in \mathbf{D}} |w_x| &\geq \sum_{v \in P_y} w_v + |w_y| \text{ by (3.2)} \\ &\geq \sum_{v \in P_y} w_v + w_y \geq T \text{ by (3.3).} \end{aligned} \tag{3.4}$$

Let $P_y^* = P_y \setminus \{y\}] \cup \mathbf{D}$. P_y^* is the cube obtained from P_y after removing y and including all the variables in **D** in positive form. Equation (3.4) implies that the cube P_y^*, in which all the don't care variables are set to 1, is an implicant of F. By definition of don't cares, this implies that the cube $P_y \setminus \{y\}$, in which all the don't care variables are set to 0, is also an implicant of F. This contradicts the fact that P_y is a prime implicant of F. □

Lemma 3.4.3. *Let F be a threshold function. Then for any weight-threshold assignment $[W,T]$ to F, $\forall a \in Supp(F)$, $|w_a| > 0$.*

Proof. (by contradiction) If $|w_a| \not> 0$, then $w_a = 0$. In this case w_a does not affect the weighted sum whether $a = 1$ or $a = 0$. Thus the output of F is not affected when $a = 1$ or $a = 0$. Thus $a \notin Supp(F)$ (by definition of a don't-care variable). This is a contradiction and hence the lemma is proved. □

Lemma 3.4.4. *If $F = xF_x + x'F_{x'}$ is a threshold function, then there exists a weight-threshold assignment for F_x and $F_{x'}$ such that they have the same variable weights.*

Proof. If $F = [W; T_F]$, then (by the definition of cofactors) $F_x \equiv [W \setminus \{w_x\}; T_F - w_x]$ and $F_{x'} \equiv [W \setminus \{w_x\}; T_F]$. Note that variable weights for both F_x and $F_{x'}$ are the same input weights ($W \setminus \{w_x\}$). □

Theorem 3.4.1. *If $F = x.F_x + x'.F_{x'}$ is a Boolean function and if $Supp(F_x) \setminus Supp(F_{x'}) \neq \phi$ and $Supp(F_{x'}) \setminus Supp(F_x) \neq \phi$, then F is not a threshold function.*

Proof. Suppose F is a threshold function. Since $Supp(F_x) \setminus Supp(F_{x'}) \neq \phi, \exists a \in Supp(F_x) \setminus Supp(F_{x'})$. Similarly $\exists b \in Supp(F_{x'}) \setminus Supp(F_x)$. The variable a is a don't care for $F_{x'}$, and b is a don't care variable for F_x. By Lemma 3.4.4, there exists a weight-threshold assignment for F in which a has the same weight in F_x and $F_{x'}$, as does b. By Lemma 3.4.2, $|w_b| < |w_a|$ in F_x because b is a don't care variable in F_x. Similarly, since a is a don't care variable in $F_{x'}$, $|w_a| < |w_b|$. These contradict Lemma 3.4.4, because it holds true for all weight-threshold assignments. Hence the statement of the theorem is true. □

Theorem 3.4.2. *If $F = xF_x + x'F_{x'}$ is a threshold function, then either $F_x \subseteq F_{x'}$ or $F_{x'} \subseteq F_x$.*

Proof. By Lemma 3.4.4, there exists a set of weights W such that $F_x \equiv [w_1, \cdots, w_n; T_{F_x}]$ and $F_{x'} \equiv [w_1, \cdots, w_n; T_{F_{x'}}]$. If $T_{F_x} > T_{F_{x'}}$, then every one-point of F_x is a one-point in $F_{x'}$ (by definition of a threshold function). Therefore $F_x \subseteq F_{x'}$. Similarly if $T_{F_{x'}} > T_{F_x}$, then $F_{x'} \subseteq F_x$. □

Theorem 3.4.3. *If $F = x.F_x + x'.F_{x'}$ is a threshold function then F_x and $F_{x'}$ are also threshold functions.*

Proof. If F is a threshold function then $\exists[W;T_F]$, that implements F (by definition of threshold logic). Now $F_x \equiv [W \setminus \{w_x\}; T_F - w_x]$ and $F_{x'} \equiv [W \setminus \{w_x\}; T_F]$ (by definition of cofactors). Since there exists a weight threshold assignment that implements F_x and $F_{x'}$, they are both threshold. □

Theorem 3.4.4. *Let $F = xF_x + x'F_{x'}$ be a Boolean function, and suppose that F_x and $F_{x'}$ are threshold functions with weight-threshold assignments $[W, T_{F_x}]$ and $[W, T_{F_{x'}}]$ (identical weights), respectively. Then F is a threshold function with $w_x = T_{F_{x'}} - T_{F_x}$, and $F = [W \cup w_x, T_{F_{x'}}]$.*

Proof. Let $X = (x_1, x_2, \ldots, x_n)$. We need to show that $F(X) = 1 \Rightarrow \sum_{x_i \in X} w_i x_i \geq T_F$, and $F(X) = 0 \Rightarrow \sum_{x_i \in X} w_i x_i < T_F$. Applying the Shannon decomposition w.r.t x_i, $F = x_i F_{x_i} + x_i' F_{x_i'}$.

Case 1: $F(X) = 1$. If $x_i = 1$ then

$$F(X) = 1 \Rightarrow F_x(X \setminus x_i) = 1 \Rightarrow \sum_{x_j \in (X \setminus x_i)} w_j x_j \geq T_{F_{x_i}}$$

$$\Rightarrow \sum_{x_j \in (X \setminus x_i)} w_j x_j + (T_{F_{x_i'}} - T_{F_{x_i}}) x_i \geq T_{F_{x_i'}}$$

$$\Rightarrow \sum_{x_j \in X} w_j x_j \geq T_F \text{ since } w_i = (T_{F_{x_i'}} - T_{F_{x_i}}) \text{ and } T_F = T_{F_{x_i'}}.$$

If $x_i = 0$ then

$$F(X) = 1 \Rightarrow F_{x'}(X \setminus x_i) = 1 \Rightarrow \sum_{x_j \in (X \setminus x_i)} w_j x_j \geq T_{F_{x'_i}}$$

$$\Rightarrow \sum_{x_j \in (X \setminus x_i)} w_j x_j + (T_{F_{x'_i}} - T_{F_{x_i}}) x_i \geq T_{F_{x'_i}}$$

$$\Rightarrow \sum_{x_j \in X} w_j x_j \geq T_F.$$

Case 2: $F(X) = 0$. If $x_i = 1$ then

$$F(X) = 0 \Rightarrow F_x(X \setminus x_i) = 0 \Rightarrow \sum_{x_j \in (X \setminus x_i)} w_j x_j < T_{F_{x_i}}$$

$$\Rightarrow \sum_{x_j \in (X \setminus x_i)} w_j x_j + (T_{F_{x'_i}} - T_{F_{x_i}}) x_i < T_{F_{x'_i}}$$

$$\Rightarrow \sum_{x_j \in X} w_j x_j < T_F.$$

If $x_i = 0$ then

$$F(X) = 0 \Rightarrow F_{x'}(X \setminus x_i) = 0 \Rightarrow \sum_{x_j \in (X \setminus x_i)} w_j x_j < T_{F_{x'_i}}$$

$$\Rightarrow \sum_{x_j \in (X \setminus x_i)} w_j x_j + (T_{F_{x'_i}} - T_{F_{x_i}}) x_i < T_{F_{x'_i}}$$

$$\Rightarrow \sum_{x_j \in X} w_j x_j < T_F.$$

□

Chapter 4

EQUIVALENCE CHECKING OF THRESHOLD LOGIC CIRCUITS

In this Chapter the first efficient procedure in literature to determine the logic function of a threshold gate is introduced (called $TG2MFF$). The procedure is provably correct and has polynomial total complexity. It generates a maximally factored form representation of the logic function, which is very compact. This procedure can be used to perform equivalence checking using the Boolean Equation Diagrams (BEDs). The effectiveness of $TG2MFF$ results in the significant speed-up($1.25X$ to $16X$) when equivalence checking is done using BEDs [46]. Moreover the maximally factored form generated is that of a minimal sum-of-products (which is the complete sum for a unate function [40]). This results in much smaller representation of logic functions using BEDs.

4.1 Definitions

The algorithm to determine the logic function realized by a threshold gate does so by generating a maximally factored form. Below is a list of some basic definitions related to factored forms. For further details, the reader is referred to Hachtel et al. [40].

Maximally Factored Form: A factored form is maximally factored, if

1. for every sum of products, there are no two syntactically equivalent factors in the product,
2. for every product of sums, there are no two syntactically equivalent factors in the sums.

Complete Sum: An SOP formula is a complete sum (a sum of all prime implicants and only prime implicants) *iff*:

1. no term includes any other term,
2. the consensus of any two terms of the formula either does not exist or is contained in some other term.

The complete sum of function F is denoted by $CS(F)$. For example, the complete sum of $ab' + ab + c$ is $a + c$.

Exact Factored Form: An exact factored form of an SOP is a factored form which when expanded by repeated algebraic multiplication only (without absorbing terms), will result in the original SOP. For example, consider the SOP form $F = ab + bc + ca$. The factored form $a(b + c + bc) + bc$ is not an exact factored form even though $F \equiv a(b + c + bc) + bc$. The factored form $a(b + c) + bc$ is an exact factored form of F.

Iterated Consensus: Iterated consensus is a method, based on the *consensus theorem* [40], and generates the complete sum of a function using any SOP. This method adds to the SOP, all the consensus terms of all pairs of cubes in the SOP. It then removes the terms that are present in other terms. This procedure is repeated until no further consensus is possible. *E.g.:* Consider the SOP of F, $x_1x_2 + x_2'x_3 + x_2x_3x_4$. *Iterated Consensus(F)* $= x_1x_2 + x_2'x_3 + x_1x_3 + x_3x_4$, the $CS(F)$.

The problem of demonstrating equivalence of two Boolean functions f and g, or their combinational circuit representations, has been extensively studied [59–61]. The "straightforward approach" is to construct an Ordered Boolean Decision Diagram (OBDD) [14] of $f \equiv g$, which reduces to **1** if they

are equivalent. The drawback of this approach is that construction of the OBDDs of f or g may not be efficient because their size may be exponential in the number of variables, regardless of the variable ordering (e.g. multiplier).

An alternative approach is to use to intermediate representations such as the AND/INVERTER graph [59] or Boolean Expression Diagram (BED) [46], that allow us to exploit the structural similarities that exist between the functions being compared. Since the implementation discussed here is based on BEDs [46], the discussion is limited only to them.

Boolean Expression Diagram

BED is a data structure obtained by extending the OBDD representation with operator vertices. A BED is similar to a logic graph representation of a Boolean circuit, with each gate replaced by an equivalent operator node and each input replaced by the corresponding variable node. All variable nodes are connected to the two terminal nodes (**0** and **1**). Figure 4.1 gives the example of a BED for the miter of the two circuits that are being compared.

A BED representation is not canonical but is polynomial in size of the original circuit. Equivalence checking of two functions f and g is done by constructing the BED of their *miter* [11] (Figure 4.2). Hulgaard et al. [46] present efficient transformations to reduce the size of the miter. The (significantly) reduced miter, can then be efficiently transformed into a OBDD [46], resulting in the equivalence check. The advantage of using BEDs is two fold. First, they provide for efficient hashing to simplify and speedup identification of structurally isomorphic parts of the two circuits. Second, it avoids creating the individual OBDDs for f and g, and constructs the OBDD of the reduced BED of the miter directly. This leads to a significant improvement

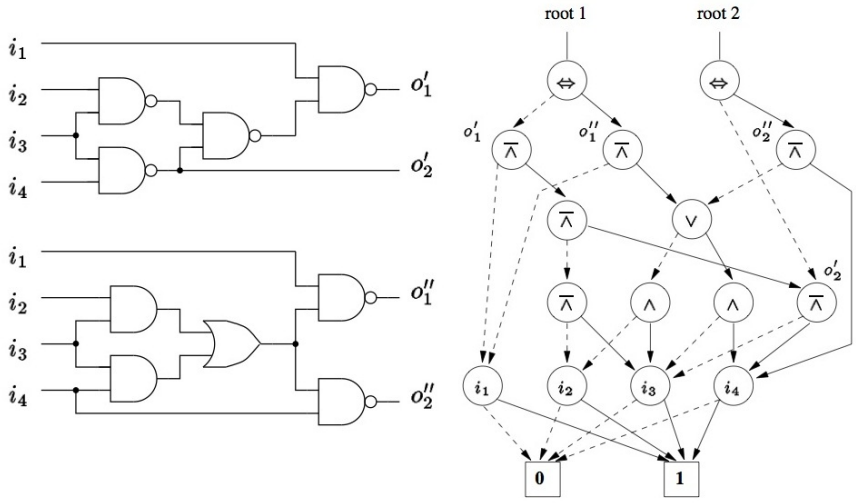


Figure 4.1: Boolean expression diagram example (taken from [46])

in performance over the OBDD based approach. In fact, it often allows equivalence checking of circuits that have exponential size OBDDs.

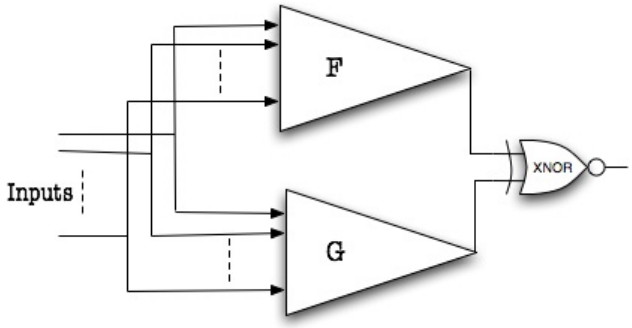


Figure 4.2: The *miter* of circuits F and G.

4.2 Problem Statement and Approach

The problem addressed in this chapter is the determination of equivalence of two threshold networks f and g. At least one of f or g is given in the form of a threshold network. The other may be logic network, a threshold network, or a

functional specification of the circuit. It is assumed that f and g have the same set of inputs and outputs, i.e., the mapping between the inputs and outputs of the two circuits is known *a priori*.

A key step in verifying equivalence of threshold networks is the determination of the logic function realized by a threshold gate. Once this is done, then a logic network with the threshold gate replaced by its logic function can be constructed. Using this the miter of the BED can be constructed to determine equivalence of the two networks.

The naive way to determine the logic function of a threshold gate is to try all 2^n input combinations and determine the *on-set* of the function, and generate a SOP representation. One of the features of threshold gates is that they permit efficient realization (both in area and delay) of gates with large fan-in. Hence the naive approach will not be practical. For instance, consider an n-input majority function which can be implemented as a single threshold gate. For $n = 16$, using the naive approach, takes over six minutes to generate the logic function and about eight seconds to verify equivalence (see Section 5). For $n = 24$, the naive approach takes more than a day and does not complete execution. In contrast, the method to be described takes about nine minutes.

Since the subsequent step of equivalence checking relies on the use of BEDs, it is important to generate a compact logic network representation of a threshold gate, as this will reduce the size of the BED. Hence, instead of generating an SOP form of a threshold gate, it is most important to generate a *maximally* factored form. It would be best if the maximal factored form of the minimal SOP (which is the complete sum for a unate/threshold

function [40, 58]) can be generated. The algorithm described herein does exactly that – it generates a maximally factored form of a minimal SOP for a threshold gate directly, without explicitly enumerating all the minterms or generating the complete sum.

4.3 The Algorithm TG2MFF

The algorithm to determine a maximally factored form of a threshold gate is referred to as **TG2MFF**. It takes an n-input threshold function $F = [W;T]$, where $W = (w_1, w_2, \ldots, w_n)$, and the support set is $X = (x_1, x_2, \ldots, x_n)$. Let $W \setminus w_k = (w_1, w_2, \ldots w_{k-1}, w_{k+1}, \ldots, w_n)$. **TG2MFF** recursively decomposes F using cofactors. Its pseudo code is given in Algorithm 12.

Statements 2 through 9 constitute the terminal cases and are easily verified. The other two terminal cases (statements 11 to 14) can be verified using the fact that all minterms are in the *on-set* of **1** and no minterm is in the *on-set* of **0**. Selecting a variable whose absolute weight is maximum is **necessary** in order to obtain a maximally factored form.

Example: Consider $F(a,b,c) \equiv [2,1,-1;2]$, with $w_a = 2, w_b = 1, w_c = -1$ and $T = 2$. Applying **TG2MFF** we get:

$$\begin{aligned} F &= [2,1,-1;2] = a \cdot [1,-1;0] + [1,-1;2] \\ &= a\{b[-1;-1] + [-1;0]\} + 0 = a\{b(1) + c'\} \\ &= a(b + c') \end{aligned}$$

It can be seen that $[2,1,-1;2]$ is a feasible assignment for the function $a(b+c')$.

Algorithm 12: Pseudo code of **TG2MFF**

```
** W = [w_1, ..., w_n], X = [x_1, ..., x_n] ** T is the threshold **
if n = 1 then
    if w_1 ≥ T and T ≤ 0 then
        return 1;
    end if
    if w_1 ≥ T and T > 0 then
        return x_1;
    end if
    if w_1 < T and T ≤ 0 then
        return x_1';
    end if
    if w_1 < T and T > 0 then
        return 0;
    end if
else
    if ∑_{∀w_j<0} w_j ≥ T then
        return 1;
    end if
    if ∑_{∀w_j>0} w_j < T then
        return 0;
    end if
    ** w_k is the largest absolute weight **;
    F_1 = [W \ w_k, T − w_k];
    F_2 = [W \ w_k, T];
    if w_k > 0 then
        return x_k · TG2MFF(F_1) + TG2MFF(F_2);
    else
        return TG2MFF(F_1) + x_k' · TG2MFF(F_2);
    end if
end if
```

Proof of Correctness

As can be seen, **TG2MFF** is very simple. However, the proof of correctness, which is essential, is not obvious. First a useful property of the co-factors of a threshold function is stated as the following Lemma.

Lemma 4.3.1. *Let* $\mid w_k \mid \geq \mid w_i \mid, \forall i$*. Suppose that* F *is positive unate in* x_k*. Let*

$CS(F) = Ax_k + B$. *Then* $A + B = A$.

Proof. Refer to Theorem 5.1.7 (pg. 121) in [73], from which the proof follows. □

Lemma 4.3.1 is also true if F is negative unate in the variable with the maximum weight. The proof is similar.

Lemma 4.3.2. *Let* $F \equiv [W;T]$. *Algorithm* ***TG2MFF***(F) *generates an exact factored form of* $CS(F)$.

Proof. We first show, by induction, that the factored form generated by **TG2MFF** evaluates to the Boolean function represented by $[W;T]$. It is trivial to verify that for the terminal cases ($n = 1$), **TG2MFF** produces a factored form that evaluates to same function as $[w,T]$.

Let w_k be the weight largest in magnitude, and assume $w_k > 0$. The proof for $w_k < 0$ is similar. From Equation 1.1 we see that setting $x_k = 1$ and $x_k = 0$ yields $[W \setminus w_k; T - w_k]$ and $[W \setminus w_k; T]$, respectively. Assume that **TG2MFF**, when supplied with $[W \setminus w_k; T - w_k]$ and $[W \setminus w_k; T]$, produces the factored forms for F_{x_k} and $F_{x_k'}$, which are the positive and negative cofactors of F. Examining the pseudo code, **TG2MFF** when supplied with $[W;T]$ produces the factored form $x_k F_{x_k} + F_{x_k'}$. We want to show that this evaluates to the function denoted by $[W;T]$.

Let F be the function that $[W;T]$ represents. By Shannon decomposition, $F = x_k F_{x_k} + x_k' F_{x_k'}$. Since F is positive unate in x_k, $CS(F) = Ax_k + B$. Computing the cofactors of F using $CS(F)$ results in $F_{x_k} = A + B$ and $F_{x_k'} = B$. By Lemma 4.3.1, $A + B = A$. Therefore, $F_{x_k} = A$. Hence

$CS(F) = x_k F_{x_k} + F_{x_{k'}}$. We have shown that what **TG2MFF** computes, evaluates to $CS(F)$, which is a representation of F.

It is now shown that **TG2MFF** produces an exact factored form. Since $CS(F) = Ax_k + B$, A and B must each be complete sums. By induction, **TG2MFF** produces exact factored forms for $[W \setminus w_k; T - w_k]$ and $[W \setminus w_k; T]$. These, if multiplied out would be the complete sums A and B, respectively. Therefore, $x_k[W \setminus w_k; T - w_k] + [W \setminus w_k; T]$ is an exact factored form of $CS(F)$. □

Theorem 4.3.1. ***TG2MFF*** *generates a maximally factored form of the complete sum of the given threshold function.*

Proof. The factorization $F = Q \cdot D + R$, obtained by dividing F by D (to get quotient Q and remainder R. Q, D and R are repeatedly factored), will result in a maximally factored form, if the following two conditions hold [40]:

1. If Q is a single cube then no literal in Q occurs in any cubes of R, and
2. If Q has more than one cube, then there is no factor of Q that is also a factor of R.

We prove that, the two conditions sufficient for maximal factorization are satisfied by the **TG2MFF** algorithm. Let $F \equiv [W; T]$, and w_k be a largest magnitude weight. As before, we assume $w_k > 0$. The proof of $w_k < 0$ is the same.

By factoring out x_k in $CS(F)$ we get $CS(F) = Ax_k + B$. Note that $B \leq A$ since $A + B = A$ by Lemma 4.3.1.

Case 1: Suppose A is a single cube. Since $B \leq A$, $B = AC$, where $C = C_1 + C_2 + \cdots + C_n$. Therefore $F = A(x_i + C_1 + C_2 + \cdots + C_n)$. Since A is a

single cube, it must have at least one literal, say y. $A = Qy$. Hence $F = Qy(x_k + C_1 + C_2 + \cdots + C_n)$.

A one-point of F is $Q = 1, y = 1, x_k = 0, C_i = 1, C_j = 0, i \neq j$, for some i, j. Therefore

$$\sum_{\ell \in Q} w_\ell + w_y + \sum_{\ell \in C_i} w_\ell \geq T$$

Since $w_k \geq w_y$,

$$\sum_{\ell \in Q_i} w_\ell + w_k + \sum_{\ell \in C_i} w_\ell \geq T$$

This implies that that $y = 0$ is in the onset of F, which is not possible. Therefore A cannot be a single cube. Hence the first condition required for a maximal factorization is satisfied.

Case 2: Now suppose A has at least two cubes and that A and B have a common factor. Therefore, let $A = (X_1 + X_2 + \cdots + X_a)(Y_1 + Y_2 + \ldots Y_b)$ and $B = (X_1 + X_2 + \cdots + X_a)(Z_1 + Z_2 + \ldots Z_c)$.

Note because the factorization is *algebraic*, none of the X_i and Y_i have a common literal and none of the X_i and Z_i have no common literal. Rewriting F, we have

$$F = (X_1 + \ldots + X_a)[(Y_1 + \ldots + Y_b)x_k + (Z_1 + \ldots + Z_c)]$$

A one-point of F is $Z_i = 1, X_j = 1, Z_p = 0, X_q = 0, x_k = 0$, for some i, j, and $\forall p \neq i, \forall q \neq j$. Hence,

$$\sum_{\ell \in Z_i} w_\ell + \sum_{\ell \in X_j} w_\ell \geq T.$$

$$\sum_{\ell \in Z_i} w_\ell + w_{X_{j_1}} + \ldots + w_{X_{j_d}} + \cdots + w_{X_{j_r}} \geq T.$$

Since $w_k \geq w_\ell, \forall \ell \ \varepsilon X_j$.

Replacing $w_{X_{j_d}}$, by w_k, we obtain

$$\sum_{\ell \varepsilon Z_k} w_\ell + w_{X_{j_1}} + w_{X_{j_2}} + \ldots w_k + \cdots + w_{X_{j_r}} \geq T.$$

This implies that $X_j = 0, Z_i = 1, x_k = 1, Z_p = 0$, for some i, and $\forall j$, every $p \neq i$ belongs to the onset of F, which is not possible. Therefore A cannot have a factor that is a factor of B, when A has more than one cube. Hence **TM2MFF** produces a maximally factored form of the complete sum of F, using the feasible weight assignment of F. □

The Verification Procedure

The equivalence checking procedure starts with two threshold networks. The maximally factored form of each threshold element in the network is obtained by the algorithm **TG2MFF**. These factored forms are used to construct the BED for each output of the two functions. As mentioned earlier the correspondence between the outputs of the two functions is known. This information is used to construct the BED of the *miter* for each output pair. The ROBDD of the *miter* is then obtained by using the BED package [93]. The BED package has efficient algorithms to convert a BED into an equivalent ROBDD. The outputs are equivalent if the ROBDD of the *miter* is the constant **1**. If all outputs of the two circuits are verified to be equivalent then the entire circuit is equivalent. To verify two circuits when one of them is Boolean and the other is threshold, a similar approach is followed.

An Example: Consider the threshold circuit shown in Figure 4.3. Assume that a synthesis tool generated this circuit when it was given the following specification:

$f = d(ab' + ac' + b'c)$; $e = (a+b)(a'+b')$.

To verify the two circuits by the method proposed, first the factored form of each node in the threshold circuit is obtained. Using $TG2MFF$ algorithm, we get the following factored forms, for each node: $X2 = d(b'+c')$;
$f = X2(c+a)$; $X1 = a'b'$; $X0 = X1 + ab$; $e = X0'$.

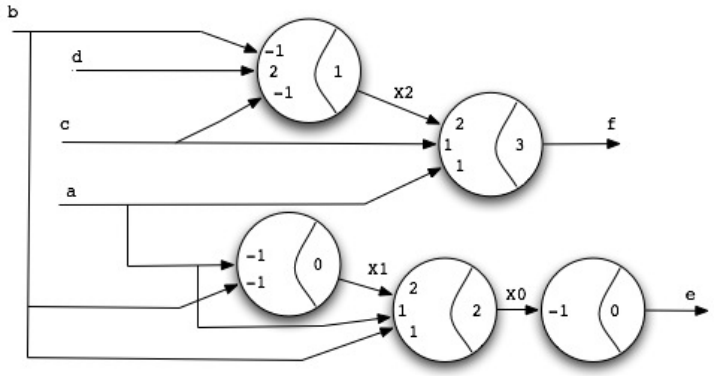


Figure 4.3: A generated threshold circuit

Using these factored forms and the circuit specification, BEDs of the miters are constructed. Since the circuits being compared here have two outputs we get two miters (*root 1* and *root 2* in Figure 4.4). The ROBDD of these two miters are constructed using the BED package. In our case the ROBDD of root 1 and root 2, turn out to be the constant **1**. Thus we can conclude that the threshold circuit synthesized is according to the specification. The verification of two threshold circuits is done in a similar way.

Complexity Analysis

TG2MFF generates a maximally factored from given a single threshold element. Hence its time complexity depends only on the size of the input and output, the size of the output being much larger than that of its input.

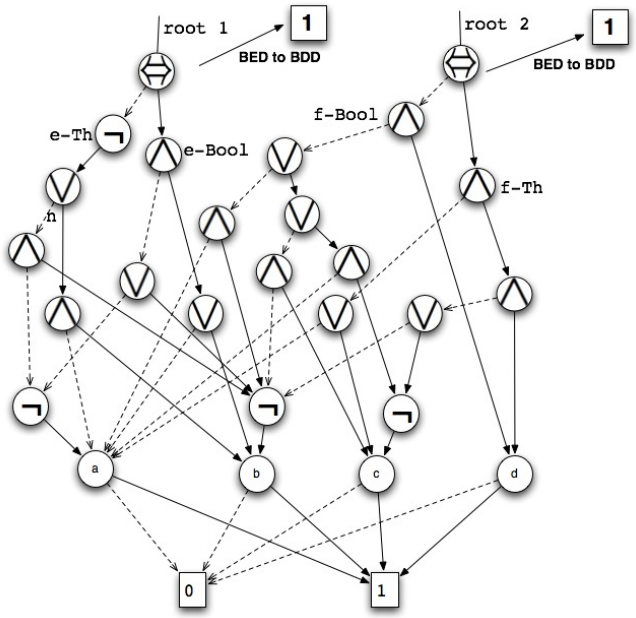


Figure 4.4: Example of threshold circuit verification

Consequently, the time complexity is expressed in terms of the size of both inputs and outputs. This is typically done for algorithms, whose output size is much larger than the input size [57].

Let n and N be the number of literals in the input and the output respectively. When a terminal case (statements $2-9$ and $11-14$) is encountered **TG2MFF** halts the recursion and in the non-terminal case (statements $15-21$), it continues the recursion. At each stage (whether terminal or non-terminal) **TG2MFF** spends $O(n)$ time. This includes the checking for terminal cases and the time taken to invoke the next stage. At each non-terminal stage a new literal is added to the generated factored form. Thus the number of non-terminal stages is $O(N)$.

Each non-terminal stage can generate at most two terminal stages. Since the number of non-terminal stages is bounded by N, the number of

terminal stages is also $O(N)$. Thus the number of invocations of the algorithm (sum of terminal and non-terminal stages) is $O(N)$. As said earlier since the algorithm spends $O(n)$ in each stage, the total complexity of **TG2MFF** is $O(nN)$. Hence the total complexity of **TG2MFF** is polynomial in the combined size of input and output. After the Boolean factor form is generated the BED generation can be done in linear time, since BED is just another representation of the factor form [46].

4.4 Experimental Results

Table 4.1: Runtime Comparison

Benchmark Circuits	Inputs/ Outputs	Avg. Fanin	Max. Fanin	**A** : TG2MFF (sec)	**B** : BED (tg2mff) (sec)	**C** : Naive (sec)	**D** : BED (naive) (sec)	C / A	D / B
f51m-t	14 / 11	5	10	0.093	0.040	0.137	0.050	1.47	1.25
z4ml-t	19 / 9	6	10	0.099	0.040	0.181	0.060	1.83	1.5
cmb-t	19 / 15	3	6	0.116	0.040	0.218	0.080	1.88	2
cu-t	24 / 21	3	6	0.210	0.060	1.075	0.200	5.12	3.33
pcle-t	16 / 4	3	6	0.090	0.040	0.123	0.050	1.37	1.25
sct-t	47 / 36	3	6	0.126	0.040	0.239	0.070	1.90	1.75
majority-8	8 / 1	8	8	0.342	0.120	7.957	1.140	23.27	9.5
cht-t	9 / 1	4	6	0.459	0.130	22.149	1.760	48.25	13.54
cm152a-t	8 / 8	11	11	0.140	0.050	1.078	0.290	7.70	5.8
ttt2-t	7 / 4	3	9	0.110	0.040	0.417	0.140	3.79	3.5
x2-t	10 / 7	5	12	0.084	0.030	0.154	0.050	1.83	1.67
9symml-t	11 / 1	5	13	0.146	0.060	1.109	0.120	7.60	2
majority-16	16 / 1	16	16	1.975	0.470	374.168	7.790	189.45	16.57
majority-24	24 / 1	24	24	413.994	92.540	>1 day	–	–	–
Average Improvement								**22.73X**	**4.9X**

The few synthesis methods that have appeared in the literature recently [3, 96] generate circuits with high fan-in gates. However, in order to best demonstart the effectiveness of $TG2MFF$, new benchmark circuits using the existing MCNC circuits were generated. Since the bottle-neck of the naive verification procedure (e.g. exhaustive enumeration of minterms) when applied to a threshold network is the fanin of gates and **not** the number of gates, threshold networks with large fanin threshold elements were generated. The directed acyclic graph representation of each MCNC benchmark circuit was taken and each node was replaced with a threshold element. This

provided a complex threshold network. The weights and threshold of each threshold element were generated randomly.

Once the threshold networks were constructed, an equivalence check of each circuit with itself was done, to examine the running time of two procedures (**TG2MFF** vs exhaustive enumeration). After deriving the logic function, the BED tool was used to check the equivalence. The experiments were run on a Sun Fire V880 machine with 16GB RAM.

Table 4.1 lists the running time required for verifying the circuits by the proposed and naive method. There are two columns corresponding to each method. The first is the run-time to generate the function and the second is the time taken for the BED based verification. As seen from the Table, **TG2MFF** is more than an order of magnitude faster than the naive method. **TG2MFF** verified the 24 input majority gate in nine minutes, whereas the naive approach could not complete execution even after twenty four hours. **TG2MFF** takes much longer to generate the factor form of majority-24, when compared to the time taken to generate the factor form for majority-16, even though the input to the algorithm in the former case is only 1.5 times that of the latter case. This is because of the large differences in their factor forms (the output of **TG2MFF**).

TG2MFF also reduces the time required for the BED based verification. It runs **22X** faster on average and speeds up the BED based verification by **5X** on average (for circuits that could be verified by both methods). The first speed up is because of the polynomial total complexity of **TG2MFF**. The second speed up is due to the compact function representation produced by the algorithm. Note that the factored form produced by **TG2MFF** is compact in two ways. First because it is the factored form of the minimal SOP (the

complete sum for a unate function). Secondly, because of the maximal factorization, the generated BED is compact. It can be observed that the time required for verification by the naive approach, depends on the fan-in of the individual gates and not necessarily on the number of gates. *Example:* Even though f51m-t has 8 gates it can be verified within a second, whereas majority-24, which has only one gate could not be verified in a day. This is because of the huge fan-in of the one gate in the majority-24 circuit.

Chapter 5

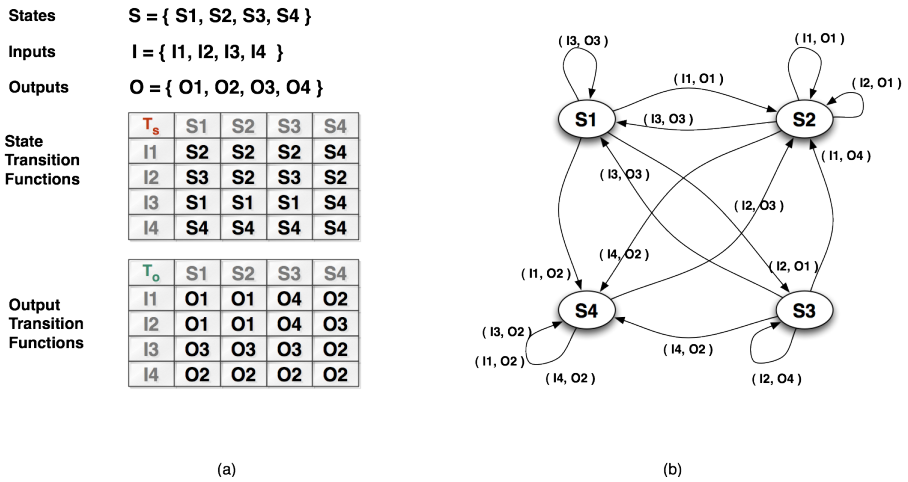
RECURSIVE BRANCH AND SEARCH ALGORITHM FOR STATE SPACE ENCODING TARGETING SINGLE-LAYER THRESHOLD CIRCUITS

State machines are common components of most practical circuits. When implementing state machines in hardware the objective is to design a circuit that has desirable features like less area, delay, power etc. Many novel circuit design techniques have been explored with the intent of improving existing circuits. Even though CMOS continues to be the most dominant design style, other alternatives to CMOS are being actively researched. This has been discussed in detail in earlier chapters. Newer techniques of synthesis, decomposition and equivalence checking have been proposed in the preceding chapters. Encoding of state machines for efficient CMOS circuit implementation is a very well understood problem. An evolutionary algorithm has been proposed recently, which searches for an encoding that yields the best threshold logic implementation. This approach uses an existing threshold synthesis tool to synthesize state machine circuit implementations.

In this Chapter we consider a more fundamental question, which is : *is there an encoding of states that will make the state machine implementable as a single layer threshold circuit?*. The aim of this work is to propose a technique inspired by the approach taken to solve the Boolean Satisfiability problem. The proposed approach aims to identify an encoding of states that will yield a single layer threshold circuit implementation (if one exists) or will declare that no such encoding exists for the choosen encoding length.

5.1 Problem Formulation

A state machine is a 5-tuple consisting of a set of states (S), inputs (I), outputs(O), state transition functions (T_S) and output transition functions (T_O). The set of outputs and output transition functions are optional, but the other three components are necessary to completely define a state machine. A state machine thus defined can be represented by a state transition diagram. Consider the state machine shown in Figure 5.1. It is a simple state machine consisting of 4 states, 4 inputs and 4 outputs. It can be completely described by the transition functions in Figure 5.1 (a), or by the state transition diagram in Figure 5.1 (b). The edges in the state transition diagram represent the transition of states when on a particular input (the first element of the tuple labeling the transition edge). The second element of the edge label represents the output value resulting from the state transition.



T_S	S1	S2	S3	S4
I1	S2	S2	S2	S4
I2	S3	S2	S3	S2
I3	S1	S1	S1	S4
I4	S4	S4	S4	S4

T_O	S1	S2	S3	S4
I1	O1	O1	O4	O2
I2	O1	O1	O4	O3
I3	O3	O3	O3	O2
I4	O2	O2	O2	O2

Figure 5.1: An example state machine (a) 5-tuple description, (b) State transition diagram representation.

The pertinent question now is how can a state machine that is described as a 5-tuple, be implemented as a circuit?

A state machine can be *encoded* by assigning a unique binary vector to each state, input and output. This results in an binary input-output relation between the input state encoding variables, (present) state encoding variables and the (next) state encoding variables and the output variables. Consider an encoding as shown in Figure 5.2 (a) for the state machine in Figure 5.1. Minimum encoding sizes has been chosen, and the encoding length of states, inputs and outputs are each 4. Let the state encoding variables be $\{s1, s2, s3, s4\}$, the input encoding variable be $\{i1, i2, i3, i4\}$ and the output encoding variables be $\{o1, o2, o3, o4\}$.

A Boolean function mapping the current state and input encoding variables to the next state and output encoding variables is just another representation of the state machine. This alternate representation of a state machine is convenient when implementing a circuit. Figure 5.2 (b) shows the truth table resulting from the chosen encoding.

In this work, the focus is on *determining an encoding of states, inputs and outputs of a given state machine such that the resulting circuit of Boolean encoding variables can be implemented with a single layer of threshold gates*. To do this the approach taken is of exploring all encodings and checking if there exists an encoding that satisfies the required condition. Since the number of encodings are large we develop heuristic techniques to prune the search space. Many years of research into pruning search spaces of assignments in a SAT solver gives us a useful framework to emulate.

To the best of the Author's knowledge, the only other similar work in literature is a method of determining if an encoding results in 2-assummable functions proposed in Coates et al. [41]. 2-assummability is a necessary but

State Encoding	Input Encoding	Output Encoding
E(S1) = 00	*E(I1) = 00*	*E(O1) = 00*
E(S2) = 01	*E(I2) = 01*	*E(O2) = 01*
E (S3) = 10	*E (I3) = 10*	*E (O3) = 10*
E (S4) = 11	*E (I4) = 11*	*E (O4) = 11*

(a)

s1(t)	s2(t)	i1(t)	i2(t)	s1(t+1)	s2(t+1)	o1(t+1)	o2(t+1)
0	0	0	0	1	1	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	1	0
0	0	1	1	1	1	0	1
0	1	0	0	0	1	0	0
0	1	0	0	0	1	0	0
0	1	1	0	0	0	1	0
0	1	1	0	1	1	0	1
1	0	0	0	0	1	1	1
1	0	0	1	1	0	1	1
1	0	1	0	0	0	1	0
1	0	1	1	1	1	0	1
1	1	0	0	1	1	0	1
1	1	0	1	0	1	1	0
1	1	1	0	1	1	0	1
1	1	1	1	1	1	0	1

(b)

Figure 5.2: An example state machine (a) An encoding, (b) Boolean function of encoding variables.

insufficient condition for a function to be threshold. Even though this work provides a detailed analysis of 2-assumability, it is not practical and is incomplete.

To summarize, from the discussion so far it can be observed that given a state machine description, determining the encoding results in a circuit specification that implements the state machine (Figure 5.3). Different encodings result in different Boolean function mapping of encoding variables and hence different circuits. This work is interested in determining the encoding that results in a ***single layer*** *threshold circuit implementation*, if it exists, or declare that no such encoding exists for the given encoding length.

Figure 5.3: Components involved in determining a circuit implementation of a state machine.

5.2 Background

In this section an overview of SAT solvers and threshold logic properties are presented. These will be made use of later in the paper for designing the proposed algorithm.

Boolean Satisfiability Solvers

Boolean Satisfiability is one of the original NP-Complete problems [20]. Despite extensive search for a polynomial algorithm to solve the SAT problem, such an algorithm has not been found. The most general version of the SAT problem has a Boolean Formula expressed in Conjunctive Normal Form (CNF). The formula is a logical and of individual clauses. Each clause is a logical or of at most three literals (for 3-SAT, which is the most common form of SAT). A literal is a variable or its negation.

Since there is no polynomial time algorithm, different heuristic techniques have been proposed over the last few decades to solve the SAT problem. But in spite of its simplicity the DPLL algorithm proposed in 1960 remains the best framework used by the most efficient solvers of today. Virtually all improvements in SAT solvers have been achieved by fine-tuning the different components of the SAT solvers.

The basic structure of a SAT solver is as shown in Algorithm 13. The

pseudo-code shows the high level operation of the SAT solver. This algorithm recursively assigns values to each literal of the given Boolean formula and checks if the assignment makes the formula satisfiable. The main focus in designing a DPLL based algorithm is to fine-tune the different components that make up this algorithm. This fine-tuning is done with the aim of minimizing the size of the explored search space of assignments.

Algorithm 13: Pseudo code of **DPLL** ()

```
1  DPLL(formula, assignment)
2  necessary = deduction(formula, assignment) ;
3  newAssignment = union(necessary, assignment) ;
4  if isSatisfiable(formula, newAssignment) then
5      return SATISFIABLE ;
6  if ifConflict(formula, newAssignment) then
7      return CONFLICT ;
8  currentVar = chooseVariable(formula, newAssignment) ;
9  currentAsgn = union(newAssignment, assign (currentVar, 1)) ;
10 if DPLL(formula, newAssignment) == SATISFIABLE then
11     return SATISFIABLE ;
12 else
13     currentAsgn = union(newAssignment, assign (currentVar, 0)) ;
14     return DPLL(formula, newAssignment) ;
```

The three major components of a DPLL SAT Solver are:

1. Deduction Algorithm
2. Branching Heuristics
3. Assignment Heuristics

The Deduction Algorithm (called on line 2 or Algorithm 13) takes stock of the implications of the last assignment. Many different techniques of

deduction have been proposed, but here we look at one popular deduction mechanism – *Boolean constraint propagation*.

During the functioning of a SAT solver, if there is any clause left with only one literal which does not have an assignment and the other two literals in the clause are assigned 0, then the only way that clause could be true is if the remaining literal is assigned to 1. Thus the search space is automatically reduced. This assignment of 1 to such clauses (popularly known as *unit clauses*) is called Boolean constraint propagation. Even though this is a simple deduction rule it has been found to be very effective in practice.

Branching heuristic refers to choosing a variable among all unassigned variables for the purpose of assigning an assignment. Different branching heuristics result in different size search trees. The popular branching heuristics count the number of occurrences of the literals in unresolved clauses. For example the Maximum Occurrences on Minimum sized clauses (MOM) heuristic chooses the literal that occurs most in the most number of unresolved clauses. This approach aims to result in a branch that result in the largest number of implications and has been found to work well in practice.

The SAT solver is a very simple case of the general constraint satisfaction problem (CSP). SAT is one of the most researched CSP. The choices made by the selection heuristic of a SAT solver is restricted to 1 and 0 for the variable chosen by the branching heuristic. For the problem we consider there can be more than two choices. SAT solvers exploit the limited size of the selection space and the unique properties of the problem (such a fixed clause size of 3, for the standard 3-SAT problem) to arrive at efficient selection heuristics. The problem of state encoding considered here does not

share these properties with the SAT solver. However, the properties of threshold functions can be exploited to design a different selection heuristics. For this purpose an overview threshold function properties is presented next.

Properties of Threshold Functions

Even though threshold gates are more expressive than the generic Boolean gates (that are implemented by pull-up / pull-down transistors) they cannot implement all Boolean functions. Although all threshold functions are known to be unate [58], not all unate functions are threshold (threshold functions are those that can be implemented by a threshold element). The function $ab + cd$ cannot be implemented by a threshold gate even though it is unate. The relation between Boolean, unate and threshold functions is shown in Figure 5.4. Theorem 5.2.1 proves the threshold functions are all unate.

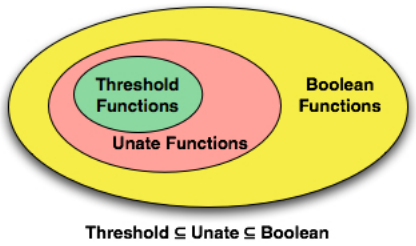


Figure 5.4: Relationship between Boolean, unate and threshold functions

Theorem 5.2.1. *If $F = xF_x + x'F_{x'}$ is a threshold function, then F is either positive unate or negative unate in x.*

Proof. By definition of unateness in order to prove the theorem all we need to show is that either $F_{x'} \subseteq F_x$ (F is positive unate in x) or $F_x \subseteq F_{x'}$ (F is negative unate in x).

By definition of a threshold function, there exists a set of weights W_F of function F, $F \equiv [w_1, w_2, \cdots, w_n; T_F]$. By the definition of cofactors: $F_x \equiv [w_2, \cdots, w_n; T_F]$ and $F_{x'} \equiv [w_2, \cdots, w_n; T_F - w_1]$. Note that $T_{F_x} T_F$ and $T_{F_{x'}} T_F - w_1$.

If $T_{F_x} > T_{F_{x'}}$, then every one-point of F_x is a one-point in $F_{x'}$ (by definition of a threshold function). Therefore $F_x \subseteq F_{x'}$. Similarly if $T_{F_{x'}} > T_{F_x}$, then $F_{x'} \subseteq F_x$.

Therefore the theorem is true.

□

Design of a DPLL Inspired Branch and Search Algorithm for State Space Encoding

The problem of interest is assigning an encoding to each state, input and output of a state-machine, such that the resulting circuit can be implemented as a single layer of threshold gates. A trivial approach would be to choose an encoding length and for every possible valid encoding (i.e every state has a unique assignment and no two states have the same encoding), check if a single layer threshold circuit implementation exists. The main issue with this approach is that the search space considered is exponential. This is identical to the issue of exponential search space encountered in the SAT solvers. Decades of advancements in SAT solvers have focussed their efforts on the three components – deduction, selection and branching heuristics to reduce the size of the explored search space. To solve the problem of state space

encoding, the generic outline of the SAT solver is used. However, specific deduction, selection and branching heuristics are designed.

5.3 The Algorithm

Overview

We will continue to use the state machine defined earlier (Figure 5.1) throughout this section to demonstrate the steps of the proposed algorithm. The overall approach involves assigning an encoding to all state machine parameters and to check if such an encoding results in a threshold logic function for each encoding variable. We will use the standard ILP test to check for thresholdness. Just like in the case of satisfiability, when given an encoding it is straightforward to check if the encoding results in a single layer threshold circuit. However, even if only the minimum encoding length is considered the number of encodings to be checked are exponential. Therefore the deduction, selection and assignment heuristics play an important role in reducing the size of the search space of encodings. The ILP method of detecting thresholdness is discussed later in the section.

The objective of this work is to design heuristics using the properties of threshold logic such that the number of encodings checked are minimal. We will design ways of checking if certain branches of the search tree are worth pursuing or not. The discussion of the structure of the SAT solver will come handy as this structure is found to be ideal in all modern SAT solvers. By incorporating the general structure of the SAT solver, we can focus on designing and fine tuning the three components – the deduction algorithm, branching and assignment heuristics.

The principle algorithm involved is listed in Algorithm 15. Before this

algorithm is invoked certain global variables have to be initialized (in Algorithm 14). The role of these global variables will be made clear by the following discussion on the book-keeping data structures used.

The initial call to **encodeStateSpace** is made by setting the *currentTruthTable* to empty. The *currentTruthTable* maintains the truth table of the resulting Boolean function of encoding variables as a result of the encodings assigned thus far. The *available assignment* table maintains all the assignments that could possibly be assigned to each state. The *currentAssignment* maintains a list of encodings assigned (empty at first). *ASGVAR* is a variable that was assigned the encoding *ENCODING*. Both are initially set to ϕ. These variables initializations are done by Algorithm 14. The encoding length is also chosen at this time. The minimum encoding length for a state machine of n states should be $\geq \lceil log(n) \rceil$.

Algorithm 14: Pseudo code of **initializeGlobals** ()

```
1 initializeGlobals();
2    Choose encoding length ;
3    Generate empty currentTruthTable ;
4    Generate empty available assignment table ;
5    Generate empty currentAssignment list ;
6    Generate empty unateTable ;
7    ASGVAR = φ ;
8    ENCODING = φ
9 return ;
```

The high-level objective of **encodeStateSpace** is to assign a unique encoding to each state, input and output and then check if the resulting circuit results in a single layer threshold circuit (see Figure 5.5). This naive approach has some issues. For a state machine of n inputs, m states and p outputs the minimum encoding lengths are $\lceil log(n) \rceil$, $\lceil log(m) \rceil$, and $\lceil log(p) \rceil$ respectively.

Algorithm 15: Pseudo code of **encodeStateSpace** (*CurrentTruthTable*, *CurrentAssignment*)

```
1  encodeStateSpace(CurrentTruthTable, CurrentAssignment);
   Input  : CurrentTruthTable is a partial multiple output truth table
            representing the next state of the state encoding variables.
            CurrentAssignment is a dictionary of states, Inputs and Outputs
            with their assigned encodings.
   Output: φ if no encoding exists. Otherwise, if an encoding is returned
            that can be implemented as a single layer threshold circuit.

   // Terminal Case
2  if isEncodingComplete (currentAssignment) then
3      if isThreshold (currentAssignment) then
4          return (currentAssignment);
5      return (φ);

6  necessaryTruthTable = deductionAlgorithm (currentTruthTable);
   // If a conflict is found
7  if necessaryTruthTable == CONFLICT then
8      return (φ);

9  unassignedVariables = getAllUnassignedVariables
   (CurrentAssignment);

   // Choose an variable to encode
10 for ASGVAR in selectionHeuristic (unassignedVariables) do
       // Decide encoding for ASGVAR.
11     ENCODING = assignmentHeuristic (ASGVAR);
12     unassignedVariables.remove(ASGVAR);
       // Update book-keeping values.
13     currentAssignment = updateAssignment (currentAssignment,
       ASGVAR, ENCODING);
14     updatedTruthTable = updateTruthTable (necessaryTruthTable,
       ASGVAR, ENCODING);
15     solution = encodeStateSpace (updatedTruthTable,
       currentAssignment);
16     if solution ≠ φ then
17         return (solution);
18 end
19 return (φ);
```

Thus the number of encodings to consider is $O(n!m!p!)$. This explosion of search space is identical to the search space size explosion encountered by

the SAT solver. We will now design deduction, assignment and branching heuristics to reduce this search space.

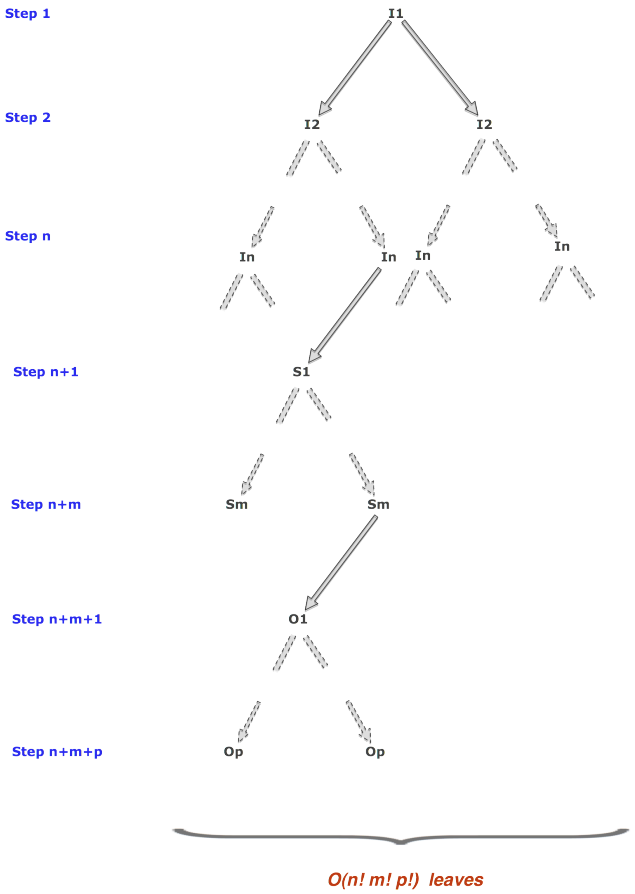


Figure 5.5: Search tree resulting from a naive branch and search encoding.

Deduction Algorithm

It may not be necessary to explore all branches of the search space of state encodings. The encodings assigned so far may dictate the choice of encoding available for other unencoded states. One simple case is that since a valid state encoding assigns unique encoding to each state, an encoding that is already assigned cannot be assigned to another state. This reduces the search tree from size $O(n^n m^m p^p)$ to a more manageable $O(n!m!p!)$, but it still is

computationally expensive. We now make use of another property of threshold logic to reduce available encodings to unencoded states even further. The encoding available to each unassigned state is maintained by the $availableAssignment$ table.

We know that threshold functions are unate (from Theorem 5.2.1). Using this information the deduction algorithm inspects the existing truth table and looks for inconsistencies. If no such inconsistencies are found the entries to the existing truth table that are implied by the current assignment are filled in – this restricts the branches that can be taken. In order to better understand the functioning of the algorithm, the following example encodings are considered for the state machine in Figure 5.1.

Algorithm 16: Pseudo code of **updateTruthTable** ()

```
1 updateTruthTable(currentTruthTable, ASGVAR, ENCODING) ;
2 for row_i in currentTruthTable do
3     if E'(currentState(row_i)) ∈ currentAssignment and
      E'(input(row_i)) ∈ currentAssignment then
4         var theNextState = nextState(E'(input(row_i)), currentState(row_i)) ;
5         if theNextState ∈ currentAssignment then
6             output(row_i) = E(theNextState);
7 end
8 return ;
```

Motivational Examples

Two examples are considered here that encode the inputs and states of the state machine in Figure 5.1. The first encoding does not lead to a single layer threshold logic implementation. In order to simplify the example we only consider encoding of the next state functions.

Example 1: Encoding $E(I1) = 00$, $E(I2) = 01$, $E(I3) = 10$, $E(I4) = 11$,

Algorithm 17: Pseudo code of **deductionAlgorithm** ()

```
1  deductionAlgorithm(currentTruthTable);
2  if ASGVAR == φ then
3      return ;
4  makeCopy (currentTruthTable, necessaryTruthTable) ;
5  for row_i in necessaryTruthTable do
6      if row_i does NOT represent a state transition rule involving ASGVAR
       then
7          continue ;
8      for j in {input encoding variables, state encoding variables} do
9          for k in {state encoding variables, output encoding variables} do
10             if getUnateness (j, k, row_i) == UNDEFINED then
11                 continue ;
12             if unateTable[j,k] == UNDEFINED then
13                 unateTable[j,k] = getUnateness (j, k, row_i) ;
14                 continue ;
15             if unateTable[j,k] ≠ getUnateness (j, k, row_i) then
16                 return CONFLICT ;
17         end
18     end
19 end
```

Algorithm 18: Pseudo code of **getUnateness** ()

```
1  getUnateness(j, k, row_i)
2  Let row_i' be a row in necessaryTruthTable, in which all inputs are assigned
   the same values as in row_i, but the value of input j is the negation of its
   value in row_i.
3  Let k' be the output value of row_i'.
4  if k == UNDEFINED or k' == UNDEFINED or k == k' then
5      return UNDEFINED ;
6  if (k > k' and j > j') or (k < k' and j < j') then
7      return Positive ;
8  return Negative ;
```

$E(S1) = 00$, $E(S2) = 01$, $E(S3) = 10$, $E(S4) = 11$.

Figure 5.6 shows the partial search tree leading to the assigned

encoding. In the first step the encoding length is chosen and the empty $currentTruthTable$ is initialized. This truth table has all the inputs rows filled. $input(row_i)$ represents the binary vector that represents the input encoding bits in row_i of $currentTruthTable$, and $currentState(row_i)$ denote the state encoding bits in row_i of $currentTruthTable$. As the encoding is assigned to all states and inputs each $output(row_i)$, which is the output vector of the function of truth table in Figure 5.6 ($currentTruthTable$) is assigned. Let us now consider the changes to the global data structures that take place as each parameter of the state machine is assigned an encoding.

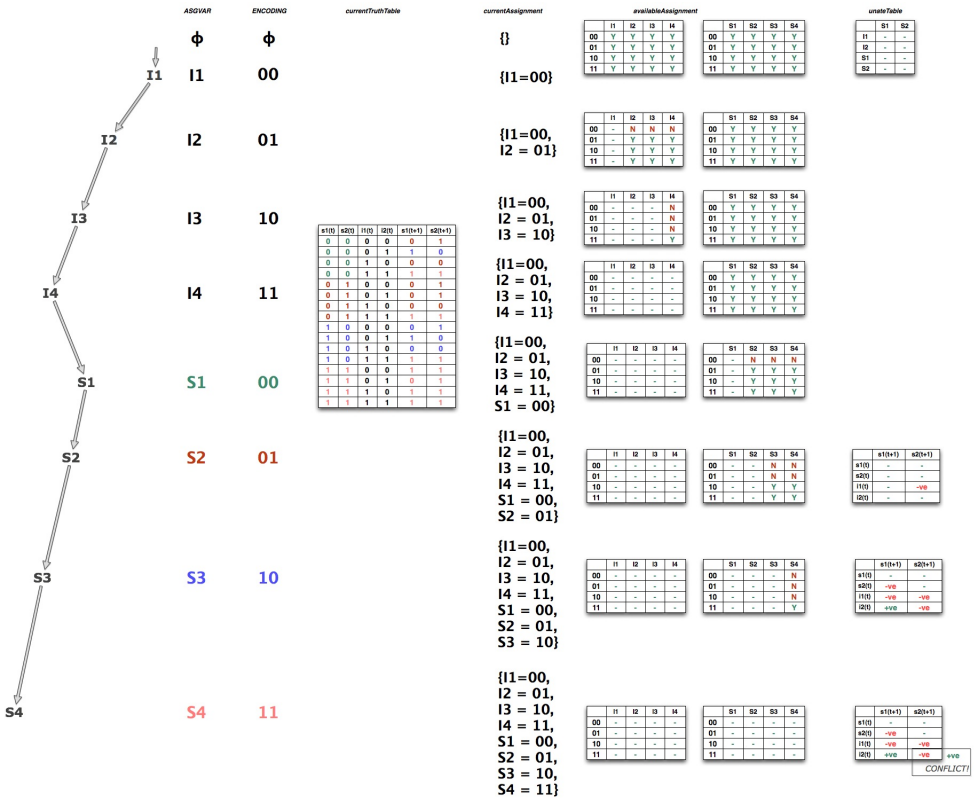


Figure 5.6: Applying $encodeStateSpace$ when the encoding is $E(I1) = 00$, $E(I2) = 01$, $E(I3) = 10$, $E(I4) = 11$, $E(S1) = 00$, $E(S2) = 01$, $E(S3) = 10$, $E(S4) = 11$.

When the inputs $I1$, $I2$, $I3$, $I4$ and $S1$ are assigned encoding the output

of $currentTruthTable$ remains unchanged, but $currentAssignment$ is updated to reflect the changes (line 12 of Algorithm 15). When $S2$ is assigned an encoding, the outputs on rows $1,3,5,6$ and 7 are populated to reflect the behavior of the state machine (line 13 of Algorithm 15). These entries are shown in red and are decided according to Algorithm 16. Note that $E(S)$, represents the binary vector of encoding variables for state S in $currentAssignment$, and $E'(\{s_1,\cdots,s_n\})$ represents the state that has the encoding $\{s_1,\cdots,s_n\}$ in $currentAssignment$.

After this $encodeStateSpace$ is invoked again with updated $currentTruthTable$ and $currentAssignment$ values (line 14 of Algorithm 15). In line 2 of Algorithm 15, it is checked if the encoding is complete. The encoding is complete if $currentTruthTable$ is completely defined. Since $currentTruthTable$ is not yet complete, the control now passes to line 4, where $deductionAlgorithm$ is invoked.

The $deductionAlgorithm$ inspects the newly filled rows in $necessaryTruthTable$ and to determine if there is any contradiction to the *unateness property* (Theorem **??**) of threshold functions. If a conflict if found the current search path will be abandoned (line 16 of Algorithm **??**). If no conflict is found then the unateness imposed by $necessaryTruthTable$ is made note of in $unateTable$ (line 13 of Algorithm **??**).

For the current encoding there is a conflict in the unateness of $s2(t)$ with respect to $s2(t+1)$. The $currentTruthTable$ after all state machine parameters ($\{I1\cdots I4\}$, $\{S1\cdots S4\}$)are assigned encodings, implies that output $s2(t+1)$ is both positive unate and negative unate with respect to input $s2(t)$. This gives rise to a conflict, and hence the current assignment will not result in a single

layer threshold circuit implementation.

Example 2: Encoding $E(I1) = 00$, $E(I2) = 01$, $E(I3) = 11$, $E(I4) = 10$, $E(S1) = 00$, $E(S2) = 11$, $E(S3) = 01$, $E(S4) = 10$.

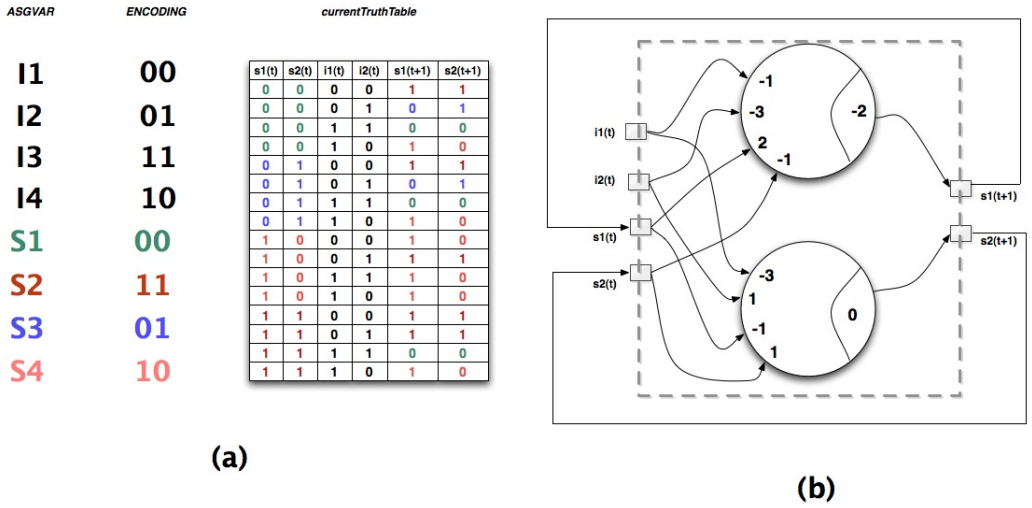


Figure 5.7: (a) $currentTruthTable$ when the encoding is $E(I1) = 00$, $E(I2) = 01$, $E(I3) = 11$, $E(I4) = 10$, $E(S1) = 00$, $E(S2) = 11$, $E(S3) = 01$, $E(S4) = 10$. (b) Single level threshold circuit implementation of the state machine.

The second encoding when run through the steps of $encodeStateSpace$, does not give rise to any conflict and is hence a candidate for checking if a single layer threshold circuit can be obtained. Using an ILP solver for the truth table in Figure 5.7 (a) results in the threshold circuit shown in Figure 5.7 (b).

Detecting if a Function is Threshold

On line 3 of Algorithm 15, the function $isThreshold$ is used to detect if a function is threshold or not. There are many different ways in which this can be done. The traditional approach is to use the Integer Linear Programming (ILP) to both detect and assign weights to threshold functions. Recently

certain efficient but incomplete approaches have been proposed. For more details on this refer to [36], [38].

This work uses the Integer Linear Programming (ILP) formulation to test if a function is threshold. In the following discussion examples are used to show how a Boolean function can be checked if it is threshold or not using the ILP formulation.

Consider the function $F = a + bc$. The truth table of this function is shown in Figure 5.8. To determine if this function is threshold, an Integer Linear Program (ILP) is generated as follows:

First an assumption is made that the given function (say F) *is* threshold and there exists a threshold element with input weights w_a, w_b, w_c and threshold T that implements the function. The constraints of the ILP are generated from the rows of the truth table. One constraint is obtained from each row (except for the row in which all inputs are 0) of the truth table. For example, consider row 4, b and c are the only inputs that is 1 and the output is 1. Therefore $w_b + w_c \geq T$ (from the definition of a threshold function). For the ILP of function F the set of all constraints are: $w_c < T$ (row 2), $w_b < T$ (row 3), $w_b + w_c \geq T$ (row 4), $w_a \geq T$ (row 5), $w_a + w_c \geq T$ (row 6), $w_a + w_b \geq T$ (row 7) and $w_a + w_b + w_c \geq T$ (row 8). The objective function is set to: Minimize $w_a + w_b + w_c + T$. This is a reasonable aim since in most threshold logic circuit implementations reduction in weights and threshold result in a reduction in area/power of the circuit. In any case, we are mostly interested in identifying the feasibility of the ILP (which implies that the function F is threshold – the original assumption). In case the ILP is not feasible it implies that the original assumption was wrong and the function F is not threshold. This function is

threshold and an integer solution to the ILP is: $[w_a = 2, w_b = 1, w_c = 1; T = 2]$.

Row #	a	b	c	F
1	0	0	0	0
2	0	0	1	0
3	0	1	0	0
4	0	1	1	1
5	1	0	0	1
6	1	0	1	1
7	1	1	0	1
8	1	1	1	1

Figure 5.8: Truth Table for the function $F = a + bc$.

Now consider an example of a function which is not threshold: $G = ab + cd$. This function is not threshold because among the constraint equations we have $w_a + w_b \geq T$, $w_c + w_d \geq T$, $w_a + w_c < T$, $w_b + w_d < T$. Adding the first two and the last two constraints we get $w_a + w_b + w_c + w_d \geq 2T$ and $w_a + w_b + w_c + w_d < 2T$, which are clearly incompatible. The ILP hence has no solution which implies the function G is not threshold.

Selection and Assignment Heuristics

In the motivational examples discussed in the previous sections the selection of variable to be assigned encoding, as well as its encoding were decided before hand. It is clear from the examples about that the order in which the state machine parameters are assigned encodings and the encodings assigned determine if the encoding is desirable (meaning that it leads to a state machine circuit, which can be implemented using a single layer of threshold gates). In this section we introduce different heuristics for

determining the next parameter to be assigned encoding and the choice of encoding assigned in order to arrive at a desirable encoding with the minimal exploration of search space.

To achieve this goal the selection and assignment heuristics can be designed to do either of the following:

1. To choose the parameter to encode, and to choose an encoding for it that has the best chance of leading to a desirable encoding.
2. To choose the parameter to encode, and to choose an encoding for it leads to an conflicting encoding sooner.

The first kind of heuristic is designed to arrive at a desirable encoding eventually. However, the second kind of heuristic is to arrive at a bad encoding sooner so that the total size of the search space explored is as small as possible. The first type of encoding is much suited for a state machine for which a desirable encoding exists, and the second type is more suited for a state machine which has no desirable encoding. The approach taken in this work is to arrive at a constraint sooner. The work chooses the best solution (which is the solution that has the most number of state encoding functions that are threshold) in case there does not exist a perfect solution.

Note that selection and assignment heuristics for a 3-SAT solver are much simpler as the structure of the problem restricts the selection and assignment heuristics (for e.g. there are only two choices for variable assignment 1 and 0, but this is not the case for our problem).

The following are the heuristics evaluated in this work: *Select the most constrained variable; assign the most constraining value*

Consider the state machine and assignment shown in Figure 5.9 (a). After $S2$ has been assigned encoding, values of $currentTruthTable$, $currentAssignment$, $availableAssignment$ and $unateTable$ is as shown in Figure 5.9 (b, c). Since $s2(t+1)$ is negative unate with respect to $i1(t)$, it can be inferred that $s2(t+1)$ cannot be zero, as the output of row 4, is 01 and row 2 is identical to row 4, except for $i1(t)$. Thus $S3$ cannot be assigned encoding 10 as it would make $i1(t) = 0$, because of how the state machine transitions are defined. This is reflected in the last $availableAssignment$ table in Figure 5.9 (c). This makes $S3$ more constrained than $S4$ since they have 1 and 2 encoding choices available respectively.

As there is only one encoding choice for $S3$, the issue of choosing the assignment does not arise in this example.

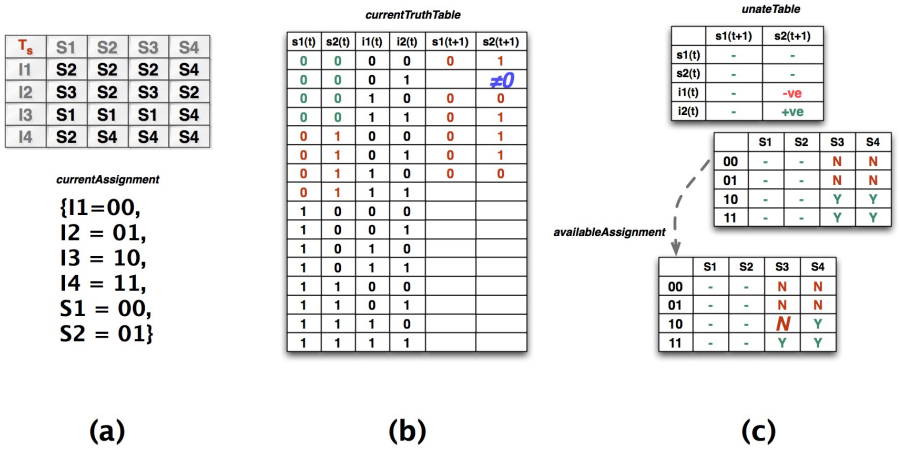


Figure 5.9: (a) A state machine. (b) $currentTruthTable$. (c) Book-keeping data-structures

The heuristics presented here were evaluated on MCNC benchmarks [25]. Other heuristics and speed-up methods can be integrated to improve the proposed procedure. We discuss one such technique called

forward checking now.

Forward Checking

In Line 11 of Algorithm 14, $assignmentHeuristic$ returns an ordered list of available assignments. As described in the previous section, different heuristics can be used to determine this list. In the normal execution of the Algorithm, this list is not empty. Forward Checking is to look one or more steps ahead to see if the domain of available assignments is not empty.

The objective of this look-ahead is to reduce the amount of backtracking. It is clear that a complete look-ahead (to the leaf of the search tree) eliminates *all* backtracking. The issue with this is that the computation time saved by reduced backtracking is small compared to the time invested in forward checking. The relationship between forward checking and backtracking time is shown in Figure 5.10. It has been found that for most practical Constraint Satisfiability Problems (CSPs) forward checking to one level (look-ahead of two, called arc consistency) or two levels (look-ahead of three, called path consistency) results in the best computational efficiency [23].

Completeness and Exactness of Proposed Approach

The proposed procedure is both complete and exact. This means that the proposed procedure finds a solution if one exists, and if not, reports that no solution is possible.

The procedure searches all the possible encodings and is hence complete. The only branches pruned in the branch and search procedure are done so because they violate a *necessary* conditions of thresholdness (such as unateness and not having a ILP solution). The procedure returns an

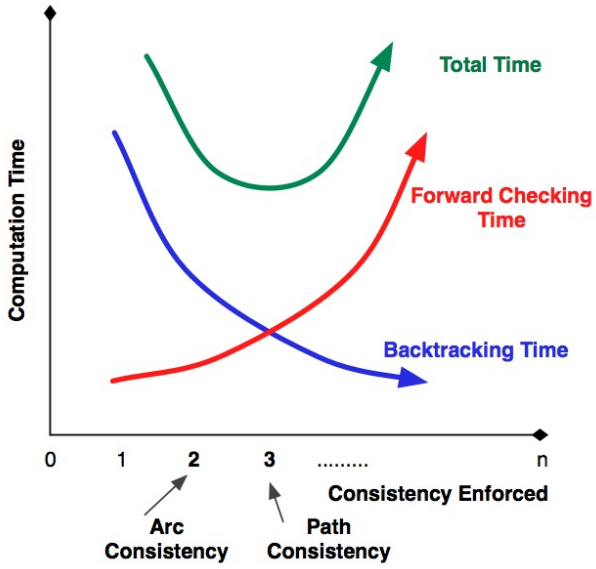


Figure 5.10: Variation of forward checking time and computation time as the amount of look ahead is varied.

encoding that results in a single layer threshold circuit, or returns *nil* (ϕ), if no solution exists. Again, since all possible encodings of a finite length are inspected the procedure is exact.

5.4 Experimental Results

The selection and assignment heuristics along with the deduction algorithm and forward checking added to the naive approach. The proposed approach was tested on state machines provided by the MCNC benchmark suite [25]. The benchmark circuits have inputs and outputs encoded, but the state encodings are unspecified. The experiments were run to determine the encoding of states that result in a single layer threshold circuit. However, since not all circuits result in a single layer implementation. In fact only five circuits (out of about 50) have single layer implementations. These circuits are listed in Table 5.1.

Table 5.1: Computation time required for different approaches and heuristics.

BENCHMARK	# STATES	# Input Enc. Vars	# Input Enc. Vars	# TH.FUNCs.	% OF TH.FUNCs.
bbtas	6	2	3	3	100
dk15	4	3	2	2	100
dk27	7	1	3	3	100
shiftreg	8	1	3	3	100
tav	4	4	2	2	100

The experiments were run on a Linux Desktop with 4 GB RAM. Most of the other circuits had very low number of threshold functions and have not been listed in the Table. The reason for this is that as the input size of functions increase the percentage of threshold functions decreases dramatically. The percentage of Boolean functions with 8 or more inputs that are threshold is less than $10^{-6}\%$. This explains why the circuits whose encoding variable functions had large support sets do not have *any* functions that can be implemented in a single layer threshold implementation. This work however, can be a starting poit of more complex decomposition algorithms that implement the state machine circuit as a *hybrid* circuit consisting of both Boolean and threshold elements.

Chapter 6

THRESHOLD LOGIC GENE MODEL OF EMBRYO DEVELOPMENT IN DROSOPHILA MELANOGASTER

In this chapter a detailed rational for using threshold logic is provided by analyzing its biological relevance. Threshold logic models for the anterior posterior patterning and the dorsal ventral germ layer formation are developed and both are evaluated by extensive simulation. With respect to the dorsal ventral model developed a novel spatial representation of the underlying system that helps study the distribution of germ layers within the dorsal ventral axis of the *Drosophila* embryo is introduced. It is demonstrated that threshold logic can generate accurate models that can be then used to predict the behavior of the underlying system. A discussion on how the models generated mimic the properties of actual gene systems like homeostasis, switch like behavior and stability is also provided.

The first step in the generation of gene models is to determine the regulatory interaction between pairs of genes (activation and inhibition). This pairwise interaction can be represented by a directed graph called the *gene interaction graph*. This graph is used to generate the *gene interaction model*. There are two types of gene regulatory models. One assumes the expression of the gene products to be continuous functions which interact with each other in continuous time (*e.g:* [39], [76]). Another group of models assume that gene expression takes place in discrete levels and that gene interaction takes place in discrete time (*e.g:* [84], [1]). The challenge is to come up with a model that captures the temporal and spatial characteristics of gene action which is easy to construct and simulate. It is generally agreed that merely specifying the type

of interaction between genes is not enough to characterize gene regulation [1]. In this work it is shown that for at least two gene systems knowing the kind of interaction between genes will suffice to construct an expressive model which can mimic the biological process accurately. For the embryo patterning problem that is considered in this chapter, a fixed number of discrete levels of expression is assumed for gene products. This approach is well suited for this purpose as once the gene is either expressed or not expressed in a cell, it remains that way to enable the cell to develop towards its determined *fate*.

The importance of developing accurate gene regulatory models can be understood by the following two examples. An important application of gene models is in cancer research [18]. It is now understood that there is a genetic basis for most cancers [16]. The normal tissue and the cancerous tissue differ in the *activity* of genes. If an accurate gene model is developed for a particular cancer it might be possible to prevent cancer [83]. A predictive gene model can also provide insights for developing an effective cure for cancer. Another domain where the interaction of genes is involved in creating complex behavior is developmental biology. The activity of genes in a cell determine the nature of the cell. In a developing embryo the undifferentiated cells differentiate into different cells and eventually into different organs by the action of genes involved in embryo development [94]. Modeling these genes is useful understand how simple interaction between genes can create the complex diversity of cells in an adult body. This will eventually provide insights into understanding organ formation and the manifestation of genetic abnormalities.

Most of the advances in developmental biology have come from the work done on fruit flies – the *model organism* in developmental biology [77]. Different groups of genes are responsible for the changes that takes place in

the fruit fly embryo. Like all animals with bilateral symmetry, *Drosophila* is patterned along two distinct, independent axes: the anterior-posterior axis and the dorsal-ventral axis [94]. Along the anterior-posterior axis the larva appears regularly segmented. During early embryo development, the dorsal-ventral axis is divided into four distinct regions [94]. Organization along these two axes occurs more-or-less simultaneously and is regulated by different sets of genes.

In [1], it is shown that by using two discrete states of gene expression and Boolean logic for gene regulation rules it is possible to model the action of segment polarity genes in the *Drosophila* embryo. While it has been demonstrated that Boolean logic rules are enough to describe gene interaction, there is little understanding on how to construct Boolean rules for a generic gene regulatory system. In this work it is proposed that the Boolean rules used in gene regulation belong to a special class of functions called threshold functions [24, 58]. These functions are a subset of Boolean functions (the percentage of Boolean functions that are threshold decreases exponentially with the number of inputs [47]). Threshold logic was first proposed as a alternative to Boolean logic for modeling gene regulation in [35]. In that work threshold logic was used to model the dorsal-ventral germ layer in *Drosophila*. It focussed mainly on speeding up the simulation of gene model using specialized hardware.

6.1 Approach

Choice of Threshold Logic for Modeling

Threshold logic as mentioned before has discrete inputs and output. There are several reasons why a discrete model was chosen to model the effect of

genes in the embryogenesis of *Drosophila*. Principal among them is that much of the experimental data available is coarse grained. In this situation, a discrete model is more preferable than a continuous model [48].

The use of discrete model is also well suited for the gene system of interest (embryo development). Morphogens are chemicals that are responsible for cell differentiation and trigger switch like behavior in cells [94]. Cells are not able to sense the absolute concentration of morphogens. However, they can recognize if the concentration of the a particular morphogen is above a certain level (threshold). This is described as the *French flag* model of cell pattern formation [94]. Figure 6.1 shows how the ability of cells to perceive different discrete chemical concentrations can lead to the formation of different kinds of cells. If the cells are sensitive to three discrete concentrations, three kinds of cells are formed along a continuous morphogen gradient. Similarly, if the cells are sensitive to two levels, two kinds of cells are formed.

Now a note on how the threshold logic gene model improves on one of the most popular discrete gene model – the Boolean logic gene model [1, 56, 84]. A gene can affect another gene by either activating or inhibiting it. This can be modeled intuitively in the threshold gene model using positive and negative weights. A positive weight corresponds to an activator gene and a negative weight corresponds to an inhibitor gene. *Example:* Consider the threshold element shown in Figure 6.2. Gene g_y is an activator of gene g_x and is hence assigned a positive weight. Similarly g_z inhibits g_x and is therefore assigned a negative weight. The threshold model can model the difference in the level of influence different genes have. For example, consider the TE shown in Figure 6.3. Gene g_b has more influence on gene g_a than

either g_c or g_d. This can be incorporated within the threshold model by making sure that $w_{g_b} > w_{g_c}$ and $w_{g_b} > w_{g_d}$. These two features can be trivially mapped onto the threshold logic model, but this mapping is non-evident in the Boolean logic model.

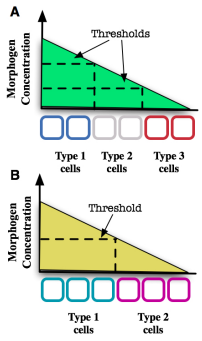
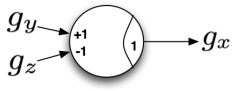


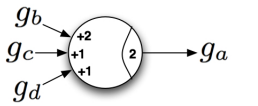
Figure 6.1: The *"French flag" patterning*. Interpretation of morphogen concentration leads to different kinds of cells.



$w_{g_y} = +1, w_{g_z} = -1; T = 1$

g_y	g_z	Weighted Sum	g_x
0	0	0	0
0	1	-1	0
1	0	1	1
1	1	0	0

Figure 6.2: The TE for the regulation of gene g_x.



$w_{g_b} = +2, w_{g_c} = +1, w_{g_d} = +1; T = 2$

g_b	g_c	g_d	Weighted Sum	g_a
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	2	1
1	0	0	1	0
1	0	1	3	1
1	1	0	3	1
1	1	1	4	1

Figure 6.3: The TE for the regulation of gene g_a.

The Procedure

The threshold model consists of a threshold element for each gene product (mRNA, protein and protein complexes), i.e there is one TE for each node in the gene interaction graph. The inputs to a TE are the incoming edges to the corresponding node in the gene interaction graph. The model is simulated by updating the threshold rules at regular time steps. This updating is done synchronously – i.e all the genes are updated at the same time. The state of the gene system at any given time is the set of all gene values. The initial state is set before the model simulation begins. In general the initial state is set according to a biologically relevant gene expression that is studied. The model enters a steady state or a steady cycle after the simulation has run for a

finite time. This steady state (if the model is accurate) corresponds to the steady state observed in biological systems.

The steps involved in generating the gene regulatory model are shown in Figure 6.4. The procedure to generate the model starts with the gene interaction graph (Step 1). Since the model has to mimic the biological process of gene regulation the model is modified iteratively until an accurate model is obtained [22] (Step 3).

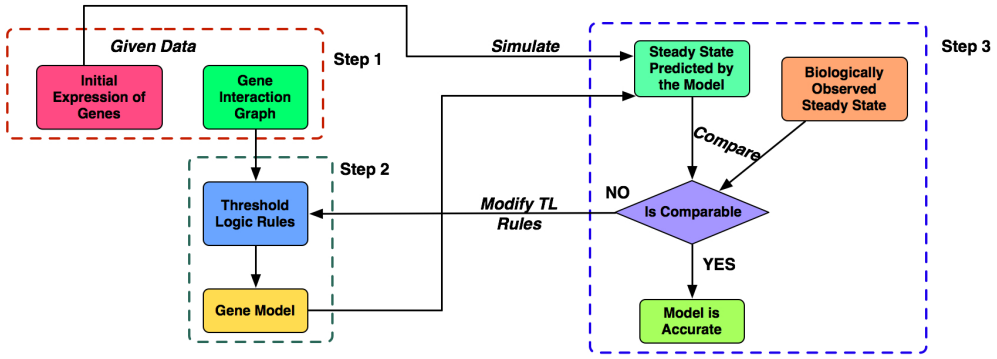


Figure 6.4: Steps involved in the generation of a threshold gene circuit

Beginning with the gene interaction graph the threshold logic rules are generated (Step 2) using the following rules (based on the insights discussed in previous subsection):

1. The weights and threshold are restricted to integers to simplify rule generation. However, this does not limit the expressiveness of threshold logic [73].

2. Genes and gene products that act as activators are assigned positive weights. Inhibitors are assigned negative weights.

3. Input gene products are selected from the gene interaction graph.

4. Since in general inhibition is stronger than activation, inhibitors have a higher absolute weight than activators.

6.2 Methods

Dorsal Ventral (DV) Modeling

Embryonic cell differentiation along the dorsal-ventral axis in *Drosophila* is essentially two dimensional. Along the dorsal-ventral axis the cells are differentiated into four different types during the blastoderm stage of the embryo [94]. Four different cell types make up the four germ layers – amnioserosa, dorsal ectoderm, ventral ectoderm and mesoderm (from the dorsal region to the ventral region as shown in Figure 6.5). The cells in the blastoderm stage embryo are concentrated in the periphery leaving a hollow in the center [94]. Therefore for modeling purposes the dorsal-ventral axis is abstracted as a two-dimentional ring made of twelve segments as shown in Figure 6.5.

The germ layer formation in the blastoderm stage is caused by the action of following genes: *twist (twi)*, *snail (sna)*, *rhomboid (rho)*, *tolloid (tld)*, *decapentaplegic (dpp)*, *zerknüllt (zen)* [94]. These genes influence each other by pairwise activation and inhibition. This influence is exerted via the proteins they synthesize. The different germ layers are determined by the activation of a subset of these genes. For example the mesoderm is formed by the cells in which *twi* and *sna* are activated [94]. These genes are also influenced by two other proteins – short gastrulation (*sog)* and dorsal (*dl*). The expression of these proteins remains unchanged during the process of germ layer formation [94].

The interaction between genes involved in the DV patterning is depicted

in the gene interaction graph as shown in Figure 6.6. The gray boxes represent one of the 12 segments of our model. The gene products between neighboring segments interact through diffusion. In each segment every gene product is expressed at a particular level and this determines the effect it has on other gene products within the same segment and the neighboring segments. The arrows (→) represent positive influence (activation) and the t-connectors (⊣) represent negative influence (inhibition). The oval nodes represents mRNA and the rectangular nodes represent proteins. The convention adopted uses lower case for mRNA and upper case for proteins.

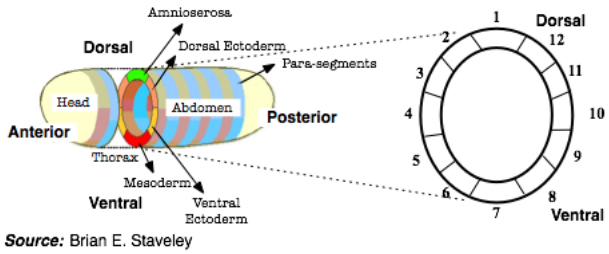


Figure 6.5: The dorsal ventral axis of the embryo is abstracted as a ring of 12 segments.

Using the gene interaction graph the threshold logic rules for each gene product is derived by the procedure outlined in the previous section. For more details on the threshold rule generation, refer to [35].

Anterior Posterior (AP) Modeling

The anterior-posterior patterning in *Drosophila* is due to the action of four sets of genes – maternal genes, gap genes, pair rule genes and segment polarity genes [39, 94] . Segmentation in *Drosophila* is due to the action of the segment polarity genes [1]. These genes are activated by the pair rule genes

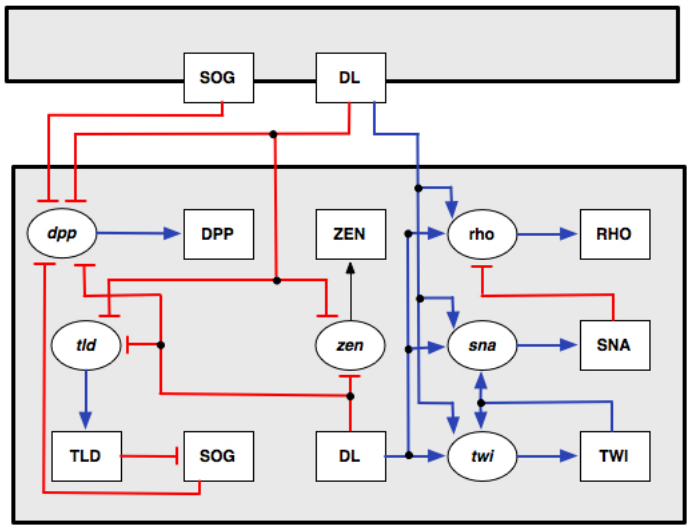


Figure 6.6: The gene interaction graph for the dorsal-ventral patterning genes.

in the cellular blastoderm phase [1]. Segment polarity genes help maintain the segment boundaries along the anterior-posterior axis in stages 8-11 of *Drosophila* embryogenesis [1]. The interest of this work is in the action of the segment polarity genes in these stages, as the goal is to explain the action of these genes in creating segments in the embryo. The interaction between different segment polarity gene products is shown in Figure 6.7. This gene interaction graph notation is similar to the dorsal-ventral gene interaction graph. The only change is that the protein complex (PHO) is represented by an octagonal node.

Since it is difficult to model a large number of cells, for modeling purposes a reasonable abstraction of the embryonic tissue is chosen as was done for the dorsal-ventral model. The expression of the segment polarity genes occur in stripes that encircle the embryo [1]. This segmentation repeats regularly along the axis and hence it is enough to model 4 segments only. The

section of 4 segments is treated as a one dimensional array of 12 cells (Figure 6.8). The segments at the ends have 2 cells each and the middle two segments have 4 cells each. The boundary of the segments is assumed to be one cell thick. This modeling abstraction is similar to the one used in [1]. The threshold logic rules is derived for each gene product as was done for the dorsal ventral case and thus the model for the genes responsible for *Drosophila* segmentation is generated. In the next section the models generated are provided and their predictive utility is discussed.

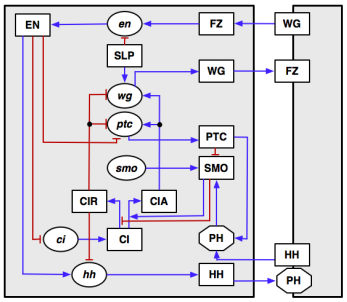


Figure 6.7: Gene interaction graph of the segment polarity genes.

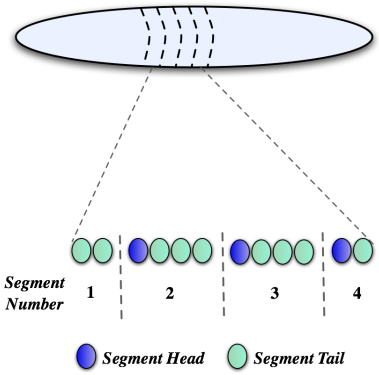


Figure 6.8: Modeling of 4 *Drosophila* segments. Each segment is assumed to be 4 cells wide. The segments in the periphery have 2 cells each.

6.3 Results

The Dorsal Ventral Model

Using the proposed procedure threshold logic rules for each node in the gene interaction graph are obtained. These rules together form the threshold logic model, as shown below:

- ***rho:*** $rho_i{}^{t+1} = [DL_{i-1}{}^t = +1, DL_i{}^t = +1, DL_{i+1}{}^t = +1, SNA_i{}^t = -3; T = 1]$.
- ***twi:*** $twi_i{}^{t+1} = [TWI_i{}^t = +2, DL_{i-1}{}^t = +1, DL_i{}^t = +1, DL_{i+1}{}^t = +1; T = 3]$.
- ***sna:*** $sna_i{}^{t+1} = [TWI_i{}^t = +2, DL_{i-1}{}^t = +1, DL_i{}^t = +1, DL_{i+1}{}^t = +1; T = 3]$.
- ***dpp:*** $dpp_i{}^{t+1} = [SOG_{i-1}{}^t = -1, SOG_i{}^t = -1, SOG_{i+1}{}^t = -1, DL_{i-1}{}^t = -1, DL_i{}^t = -1, DL_{i+1}{}^t = -1; T = -1]$.
- ***tld:*** $tld_i{}^{t+1} = [DL_{i-1}{}^t = -1, DL_i{}^t = -1, DL_{i+1}{}^t = -1; T = 0]$.
- ***zen:*** $zen_i{}^{t+1} = [DL_{i-1}{}^t = -1, DL_i{}^t = -2, DL_{i+1}{}^t = -1; T = 2]$.
- ***RH:*** $RH^{t+1} = [rh = +1; T = 1]$
- ***TWI:*** $TWI^{t+1} = [twi = +1; T = 1]$
- ***SNA:*** $SNA^{t+1} = [sna = +1; T = 1]$
- ***ZEN:*** $ZEN^{t+1} = [zen = +1; T = 1]$
- ***DPP:*** $DPP^{t+1} = [dpp = +1; T = 1]$
- ***TLD:*** $TLD^{t+1} = [tld = +1; T = 1]$

The Anterior Posterior Model

Using the gene interaction graph in Figure 6.7 and the threshold inference rules the following rules are derived:

- ***SLP:*** $SLP_i^{t+1} \equiv [SLP_i^t = 1; T = 1]]$.
- ***wg:*** $wg_i{}^{t+1} \equiv [CIR_i{}^t = -2, wg_i{}^t = 1, CIA_i{}^t = 1, SLP_i^t = 1; T = 2]$
- ***WG:*** $WG_i^{t+1} \equiv [wg_i{}^t = 1; T = 1]$
- ***en:*** $en_i{}^{t+1} \equiv [WG_{i-1}{}^t = 1, WG_{i+1}{}^t = 1, SLP_i^t = -2; T = 1]$
- ***EN:*** $EN_i^{t+1} \equiv [en_i{}^t = 1; T = 1]$
- ***hh:*** $hh_i{}^{t+1} \equiv [EN_i^t = 1, CIR_i^t = -1; T = 1]$
- ***HH:*** $HH_i^{t+1} \equiv [hh_i{}^t = 1; T = 1]$
- ***ptc:*** $ptc_i{}^{t+1} \equiv [CIA_i{}^t = 1, EN_i^t = -1, CIR_i^t = -1; T = 1]$
- ***PTC:*** $PTC_i^{t+1} \equiv [ptc_i{}^t = 3, PTC_i^t = 1, HH_{i-1}{}^t = -1, HH_{i+1}{}^t = -1; T = 1]$
- ***PH:*** $PH_i^t \equiv [PTC_i^t = 2, HH_{i-1}{}^t = 1, HH_{i+1}{}^t = 1; T = 3]$
- ***SMO:*** $SMO_i^t \equiv [PTC_i^t = -1, HH_{i-1}{}^t = 1, HH_{i+1}{}^t = 1; T = 0]$
- ***ci:*** $ci_i{}^{t+1} \equiv [EN_i^t = -1; T = 0]$
- ***CI:*** $CI_i^{t+1} \equiv [ci_i{}^t = 1; T = 1]$
- ***CIA:*** $CIA_i{}^{t+1} \equiv [CI_i^t = 3, SMO_i^t = 1, hh_{i-1}{}^t = 1, hh_{i+1}{}^t = 1; T = 4]$
- ***CIR:*** $CIR_i^{t+1} \equiv [CI_i^t = 1, SMO_i^t = -1, hh_{i-1}{}^t = -1, hh_{i+1}{}^t = -1; T = 1]$

Note: A node that activates only one other node is considered *transparent*. *e.g:* WG of neighboring cells activate FZ, which in-turn activates en. Thus en is assumed to be activated by WG of neighboring cells. Similarly HH of adjacent cells are assumed to activate SMO (via PH). The inputs to each TE is identical to the inputs used for the Boolean rules in [1] (for detailed rationale refer to [1]).

6.4 Discussion

Now the accuracy and validity of the model is examined. Steady states in a cellular system (also known as homeostatic state) results in a particular cell type [45]. It is tested if the model can predict the same homeostatic state observed in wild-type embryos. The model is simulated starting from an initial state and is updated at discrete time steps. A steady state is reached when the value of gene products do not change from one time step to the next.

The steady states observed in wild-type embryos when all genes are functioning correctly is used to build the model; so it is not surprising that the model predicted that correctly. However, the model was also able to generate other significant predictions that were not used for model building. Some of these predictions could be verified by existing biological literature. Others provide specific phenotypic effects of the malfunctioning of genes. These are useful to biologists studying fruit flies, and they can now verify if the predictions made here are biologically accurate. Apart from the predictive capability of the model, it also exhibits many properties that are inherent to gene systems. Important among them – stability, homeostasis, switch like behavior, and disproportionate influence of genes are discussed later.

Prediction of Normal Steady States

For dorsal ventral patterning the initial states of DL and SOG is known [94]. All other genes are assumed to be unexpressed in all segments. The initial state of the simulation (at $t = 0$) is shown in Figure 6.9.

In the model simulation starting with the above described initial state, after time $t = 3$ the expression of all variables in the model remain unchanged.

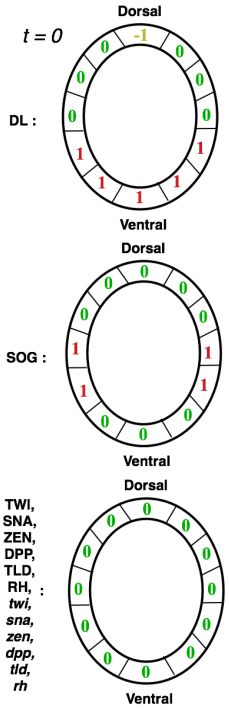


Figure 6.9: Gene expression along the DV axis at time $t = 0$.

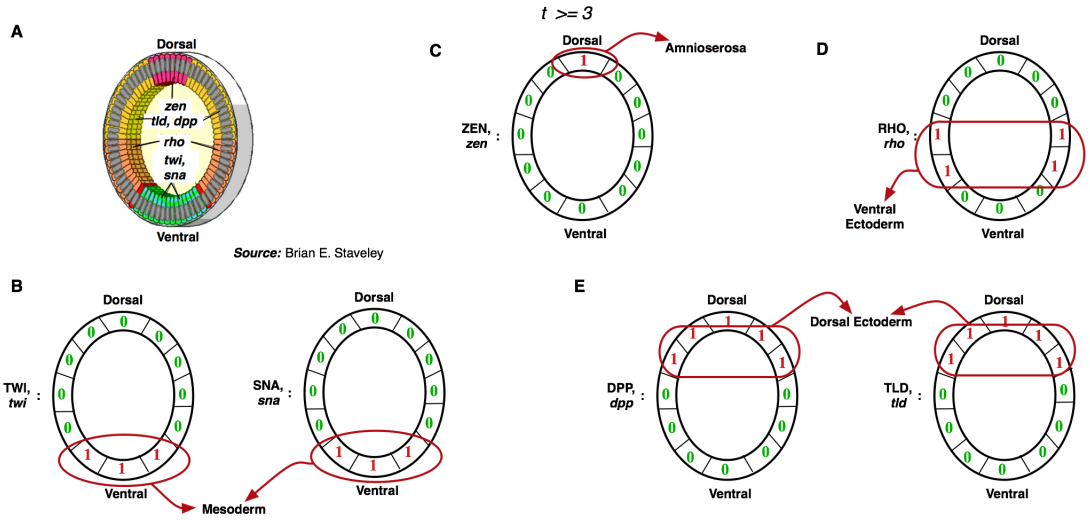


Figure 6.10: A: Biologically observed gene expression in the *Drosophila* blastoderm. B-E Steady state expression obtained from simulation of the model.

This steady state is then compared with the biological steady state observed in wild-type embryos. As shown in Figure 6.10, *zen* mRNA and protein are expressed only in the region where amnioserosa is formed in the wild-type embryo. *zen* is known to be associated with the formation of amnioserosa [94]. Similarly *twi* and *sna* are associated with the formation of mesoderm, *dpp* and *tld* are associated with the formation of dorsal ectoderm and *rho* is associated with the formation of ventral ectoderm [94]. The model predicts the region of formation of all the germ layers accurately as shown in Figure 6.10.

For the anterior posterior model the expression pattern in phase 8 wild type embryos is used as the initial state for the simulation (this is similar to the initial state used in [1]). This initial state is shown in Figure 6.11 (A). Each oval represents one of the 12 segments in the model. A red colored segment indicates the expression of the corresponding gene product in that segment (a blue segment indicated that the corresponding gene product is not expressed in that segment). The threshold gene model rules is simulated until a steady state is attained. The steady state gene expression obtained (which is a point attractor) from the model simulation is shown in Figure 6.11 (B). This is compared to the wild type expression of phase 11 embryos.

Our model could predict the following gene expression patterns experimentally observed in wild-type phase $9-11$ embryos [1]: **1.** wg and WG are expressed in the most posterior cell of each segment. **2.** en, EN, hh and HH are expressed in the most anterior cell of each segment. **3.** ci is expressed everywhere, with the exception of the cells expressing en. **4.** ptc is expressed in cells on each side of the en-expressing cells. **5.** SMO is present in broad stripes whose anterior border coincides with the anterior border of the wg stripe and whose posterior border extends about one cell. **6.** CIA is

expressed in the neighbors of the HH-expressing cells. **7.** CIR is expressed far from the HH-expressing cells.

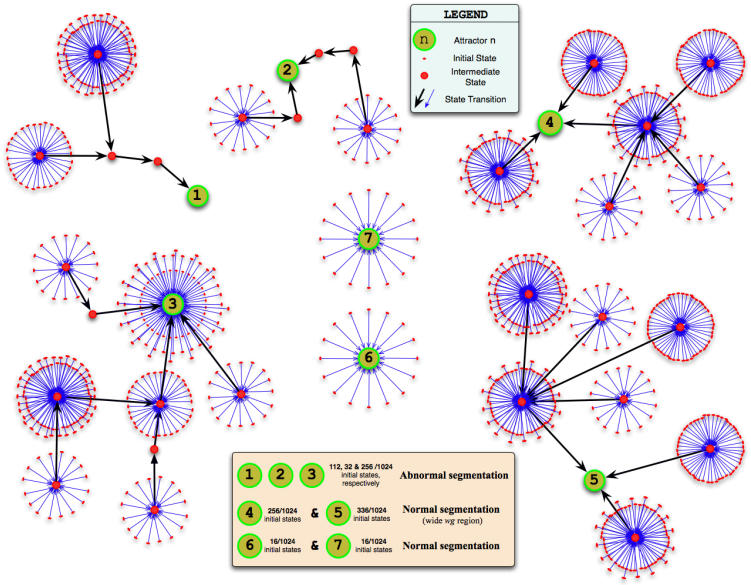


Figure 6.11: A: The initial state at $t = 0$. B: Steady state obtained by simulation of the AP model.

In-Silico Simulation of Gene Malfunctions

To further test the predictive capability of the model, more simulations were done with the dorsal-ventral model to predict the effects of abnormal gene expression. The predictions that follow are not part of the data used in generating the model.

In order to check the effect of spatially misplaced expression of *dl* gene, the following simulation was done using the model: In the model space the *dl* protein at $t = 0$ (in each germ layer) was alternately expressed and un-expressed and the simulation was executed. Since there are 4 germ layers and *dl* can take the value of 1 or 0 in each of them, there are $2^4 = 16$ unique initial states. For each of the these initial states the simulation yielded point

attractors. The results showed that the *dl* gene plays a very important part in determining the spatial location of the germ layers. The detailed results of this simulation is given in Appendix I.

It is known that when *dl* protein is excluded uniformly, *dpp* is expressed everywhere and the *twi* and *sna* are not expressed anywhere. This is called dorsalization of the embryo [94]. In the extensive simulation it was found that $DL = 0$ in all segments at $t = 0$ resulted in the exact same condition as observed in experiments with dorsalized embryos (Figure 6.12 (A)). If the *dl* protein is expressed throughout the dorsal-ventral section, *twi* and *sna* are expressed throughout and *dpp* is not expressed at all in the steady state. This is called ventralization of the embryo [94]. Our simulation was also able to predict the fate of ventralized embryos (Figure 6.12).

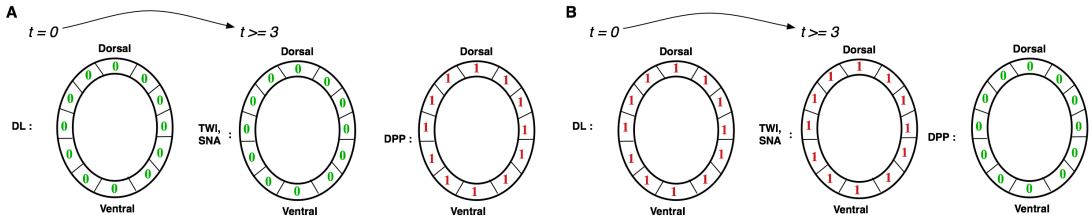


Figure 6.12: A: Gene expression predicted by the model when DL is uniformly unexpressed. B: Gene expression predicted by the model when DL is uniformly expressed.

Next, more extensive simulations were done in order to generate more predictions. Starting from 2^{24} initial states (two binary values for *rho, zen, twi, sna, dpp* and *tld* in four germ layers = $2^{(6X4)}$ initial states) the steady state corresponding to each initial state was obtained. The different initial states generated only two distinct steady states. One of them was the steady state shown in Figure 6.10. The other is shown in Figure 6.13. In this steady state *rho* is not expressed in any segment and *twi* and *sna* are expressed in the

segments where mesoderm and ventral ectoderm are formed in a wild-type embryo. The expression of all other genes is identical to the expression pattern depicted in Figure 6.10. The biological interpretation of this steady state is the absence of ventral ectoderm and the extension of the mesoderm into the regions where the ventral ectoderm should have been present. The initial states that caused this corresponded to the *twist* gene being expressed in the region corresponding to the ventral ectoderm. This result is not obvious from the gene interaction graph. This result has been verified by a biological experiment [30]. The importance of this simulation experiment is that it demonstrates that results that are not intuitive to infer from the gene interaction graph can be obtained from modeling of gene systems.

Another observation from the extensive simulation is that, only the malfunctioning of *twist* and *dorsal* genes result in abnormal states. From the gene interaction graph (Figure 6.6) we can see that these are the only 2 genes that have an out degree of ≥ 2. Moreover *twist* m-RNA and protein have a feedback loop between them.

A series of extensive simulations were then done using the AP gene model. The five genes - *wg*, *en*, *hh*, *ptc* and *ci* are assigned different values in each initial state. In each segment (4 cells), the left-most cell is called the head of the segment and the remaining 3 cells the tail of the segment. For each gene the head of the segment and the tail of the segment are assigned value of either 1 or 0. The different initial states considered here cover all the different ways each of these 5 genes are expressed at the segment boundaries. The total number of initial states thus considered is $2^{5x2} = 2^10$. Starting from each of these initial states the steady states (which all happen to be point attractors) were obtained. Our simulation showed that there exist only

7 distinct point attractors into which each of these initial states converge (in no more that 5 simulation steps). The details of this simulation is given in Appendix II.

The seven distinct attractor states can be grouped into four different phenotypic interpretations – normal segmentation, normal segmentation with extended *wg* regions, and two distict non-segmentation conditions. A majority of initial states lead to the first two types of attractors (624/1024 which is $> 60\%$). This fits well with the observed behavior of actual biological systems. Even though gene systems are complex many conditions of malfunctioning genes are self-correcting. Another feature of the simulation result – a small number of attractors also concurs with the behavior of real gene systems. In Appendix II the list of all prime implicants (the complete sum [40]) of the initial state conditions required to lead to each of these steady states is provided.

The expression of en and wg is crucial for the formation of segments [67]. To confirm this, a *mutation* experiment was performed using the anterior-posterior gene model. In this experiment en gene was inhibited. The steady state resulting from the simulation is shown in Figure 6.14. All gene products are unexpressed, with the exception of PTC, ci, CI and CIR, which are evenly expressed. When the en gene is dysfunctional it is observed that the gene expression that is required for normal segment formation does not occur in phase $9-11$ embryos. This is consistent with the biological experiments done by *silencing* en [1]. The same gene expression (Figure 6.14) is obtained even when wg is blocked. This is also in agreement with the experimental observations [1].

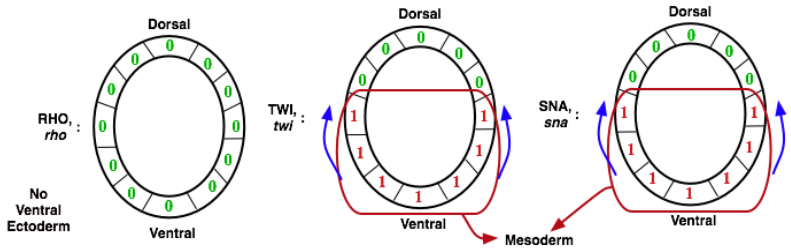


Figure 6.13: The alternate steady state. The steady state represents the absence of ventral ectoderm and a more pronounced mesoderm

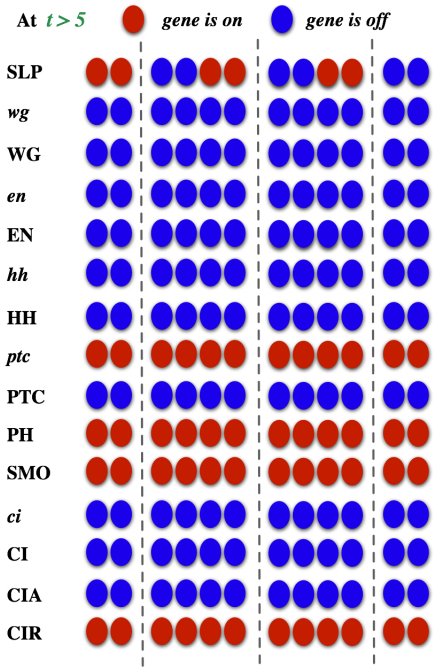


Figure 6.14: Steady state obtained after silencing *en* in the simulation.

Properties of the Threshold Logic Gene Model

The behavior of the gene models developed in this work closely resemble the behavior of actual gene systems in the following ways: **Stability:** The gene models developed converge to a very small number of steady states. This is a

property exhibited by gene systems as well. Even though there are exponential possible combinations of gene expressions only a few are found in an organism's body. Another reason why the models developed are stable is because any initial state considered converges to a steady state within a few steps of simulation. This indicates that the system will converge to a steady state even when perturbed, which is again a property exhibited by actual gene systems. **Homeostasis:** Homeostatis is the ability of a system to remain in a state of equilibrium. The gene models seen here exhibit a tendency to *settle* in a given state after a few steps into the simulation (for all the simulations discussed here the steady state was reached within 5 simulation steps). This is also a property of actual gene systems. All the steady states found by model simulation were point attractors. The threshold gene model introduced here is deterministic and this property is to be expected. More study on non-deterministic extensions of the current model will be done in the future. **Switch like behavior:** The gene systems exhibit a switch-like behavior, i.e they tend to settle in different steady states depending on the expression level of a gene. This means that the control gene acts as a switch and controls the state of the system. This property was observed in the models discussed here. For e.g: the *dl* protein to a very large extend determines which attractor the system will converge in. Similarly for the AP model the *wg*, *en* and *hh* genes act as switches and decide what state the system of genes will converge to. **Not all genes are equal:** In a gene system, it has been observed that the perturbation of some genes can have little or no effect on the system, at least at the phenotypic level. However, the system is very sensitive to the malfunctioning of few genes. This too was demonstrated by the models developed in this work. For e.g: the DV model functioned normally

under the perturbation of all genes except for *dorsal* and *twist*.

Comparison with Earlier Drosophila Models

Boolean logic has been used to model *Drosophila* embryogenesis before. In [81], a three-variable logic model is used to model the activity of *dorsal, twist* and *snail* genes. Multi-level logic and a truth-table based approach were used to model the interaction between genes. This approach is exhaustive as it enumerates all the preconditions required for the expression and non-expression of a gene. In contrast the approach presented provides a generic framework of threshold logic to model gene action. Our method also considers all the genes involved in the process and incorporates the spatial characteristics of embryo pattering (by use of different two-dimentional 12 segment construction). Our approach models intercellular interaction by incorporating the action of gene products on neighboring model segments. To the best of author's knowledge it is the first model for the DV germ layer formation that takes into account spatial considerations.

In [1], Boolean logic rules were used to model the anterior-posterior segmentation. As argued before threshold logic has benefits over generic Boolean logic (in modeling activators and inhibitors and different levels of gene influence). All the predictions reported by them was replicated by the proposed model (both wild-type steady states and the effects of gene mutations). The result of the extensive simulations on the model are also reported. This work shows that a very small subset of Boolean logic is expressive enough to model gene interaction. A major contribution of this work is to demonstrate the biological relevance of using threshold logic over generic Boolean logic to model gene regulation.

Appendix I

This Appendix we provides the results of all the steady states obtained from the simulation of the dorsal ventral model when different conditions of the dorsal gene was used. Only the phenotypic interpretation of the steady states obtained are listed (see Table 6.1). The following notation is used in the representation of the initial state.

DL_a : expression value of dorsal protein in amnioserosa.

DL_{de} : expression value of dorsal protein in dorsal ectoderm.

DL_{ve} : expression value of dorsal protein in ventral ectoderm.

DL_m : expression value of dorsal protein in mesoderm.

Appendix II

In this appendix the list all the attractors found from the simulation of the threshold logic model for the anterior-posterior segmentation of *Drosophila melanogaster* is given.

Seven attractors were obtained from simulations beginning with 1024 different initial states. All attractors were point attractors. The phenotypic interpretation of the attractors is based on the expression of the *wg, en* and *hh* genes, since they play a major role in the formation of segments. The complete sum of the initial states is also provided. These represent the minimal conditions required in the initial states that lead to the respective attractors. They provide useful information as to which kind of initial conditions result in a particular type of steady state. The subscripts of the genes in the complete sum represent the expression in the head (h) or tail (t) of each

Table 6.1: Steady States Obtained by extensive simulation of the DV model.

DL_a	DL_{de}	DL_{ve}	DL_m	**Steady State (phenotypic interpretation)**
0	0	0	0	Only dorsal ectoderm (DE) is formed, and it spans the entire D-V section.
0	0	0	1	Germ layers formed with normal spatial ordering. However, the mesoderm is reduced and the DE is enlarged.
0	0	1	0	DE present in both the dorsal and ventral ends. Ventral ectoderm (VE) is in between the two regions of DE.
0	0	1	1	Normal spatial location of germ layers. Extended mesoderm.
0	1	0	0	DE formed in the ventral region. VE is present in the dorsal region. No mesoderm is formed.
0	1	0	1	An extended VE that spans the entire embryo is formed.
0	1	1	0	Multiple regions of VE are formed. A reduced DE at the ventral tip of the embryo.
0	1	1	1	Mesoderm that covers 3/4th of the embryo from the ventral end. The rest is VE.
1	0	0	0	DE formed at it spans the entire D-V section except in the dorsal end, where the VE is formed. No mesoderm is formed.
1	0	0	1	Multiple bands of VE are formed.
1	0	1	0	VE spans the entire embryo. No other germ layers are formed.
1	0	1	1	Mesoderm in the ventral half and VE in the dorsal half of the embryo are formed.
1	1	0	0	The regions are inverted along the D-V axis, but no amnioserosa is formed.
1	1	0	1	Inverted embryo. But no DE is formed.
1	1	1	0	Germ layer location inverted.
1	1	1	1	Mesoderm spans the entire embryo.

segment.

The seven attractors are now listed together with phenotypic interpretation and the number of initial states that lead to them. The complete sum of initial conditions that resulted in each attractor is also given.

Attractor 1. (See Figure 6.15).

Abnormal segmentation (112/1024 initial states lead to this attractor).

Complete Sum : (wg_t and hh_t and ptc_t and ci_h and ci_t) or (wg_t and

ptc_h and ptc_t and ci_t) or (hh_h and hh_t and ptc_h and ptc_t and ci_t).

Attractor 2. (See Figure 6.16).

Abnormal segmentation (32/1024 initial states lead to this attractor).

Complete Sum : (wg_t and hh_t and ptc_t and ci_h and ci_t).

Attractor 3. (See Figure 6.17).

Normal segmentation; wide wg region (256/1024 initial states lead to this attractor).

Complete Sum : (hh_t and ptc_h and ci_t) or (hh_h and ptc_h and ptc_t and ci_h and ci_t) or (wg_t and ptc_h and ptc_t and ci_t) or (wg_t and ptc_h and ci_h and ci_t) or (wg_t and hh_t and ptc_h).

Attractor 4. (See Figure 6.18).

Abnormal segmentation (256/1024 initial states lead to this attractor).

Complete Sum : (wg_t and ptc_t and ci_t) or (hh_h and hh_t and ci_h and ci_t) or (hh_h and hh_t and ptc_t and ci_t).

Attractor 5. (See Figure 6.19).

Normal segmentation (16/1024 initial states lead to this attractor).

Complete Sum : (hh_h and hh_t and ptc_h and ptc_t and ci_h and ci_t).

Attractor 6. (See Figure 6.20).

Normal segmentation (16/1024 initial states lead to this attractor).

Complete Sum : (hh_h and hh_t and ptc_h and ptc_t and ci_h and ci_t).

Attractor 7. (See Figure 6.21).

Normal segmentation; wide wg region (336/1024 initial states lead to this attractor).

Complete Sum : (ptc_h and ptc_t and ci_h) or (hh_t and ptc_h and ci_t) or (hh_t and ptc_h and ptc_t and ci_t) or (wg_t and ptc_h and ci_h and ci_t) or (wg_t and hh_t and ptc_h) or (hh_h and hh_t and ptc_h and ptc_t).

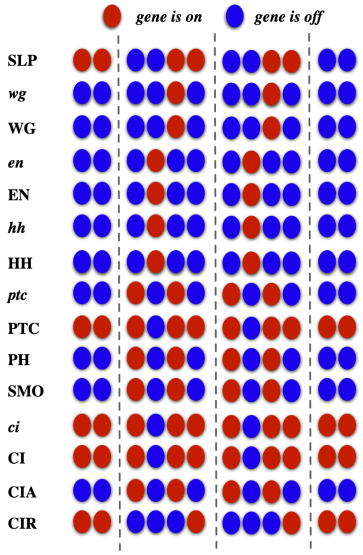


Figure 6.15: Attractor 1 of the extensive simulation of the anterior-posterior gene model.

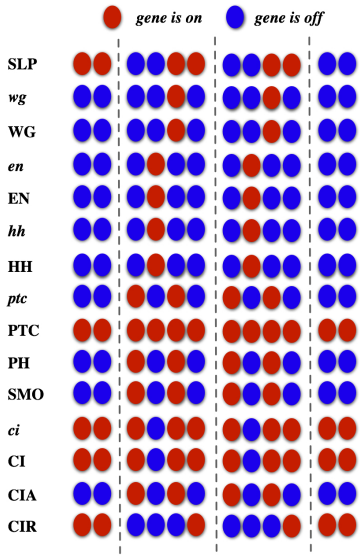


Figure 6.16: Attractor 2 of the extensive simulation of the anterior-posterior gene model.

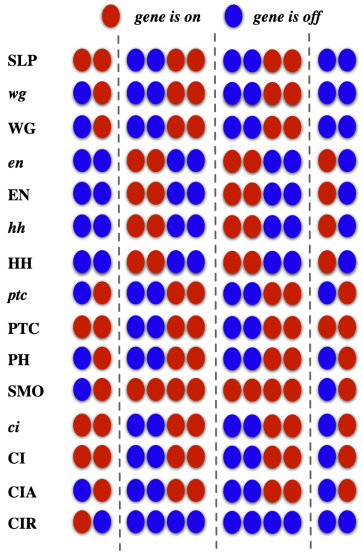


Figure 6.17: Attractor 3 of the extensive simulation of the anterior-posterior gene model.

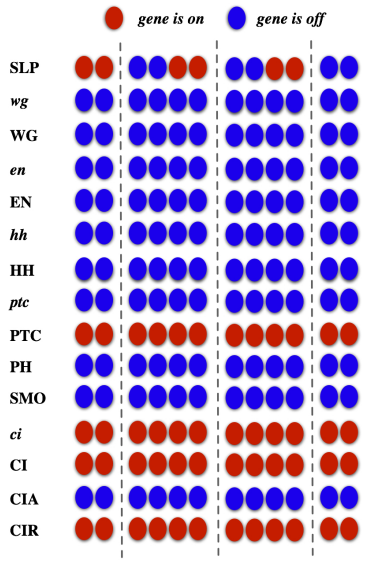


Figure 6.18: Attractor 4 of the extensive simulation of the anterior-posterior gene model.

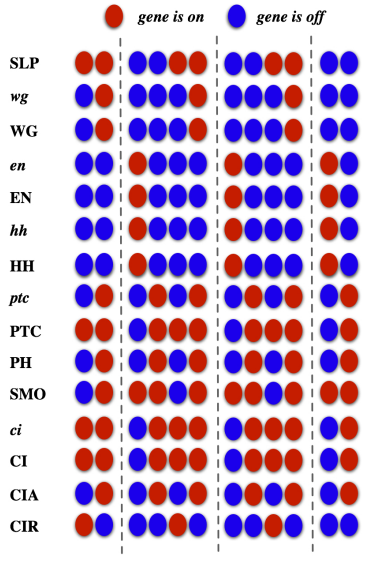


Figure 6.19: Attractor 5 of the extensive simulation of the anterior-posterior gene model.

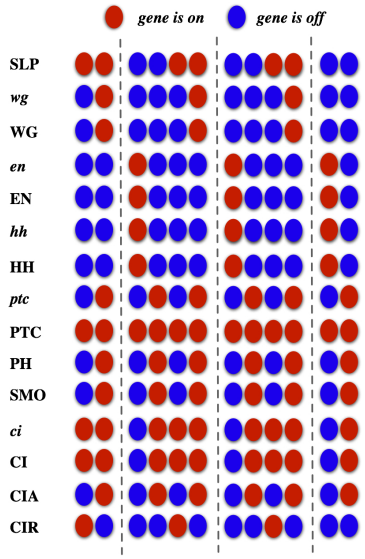


Figure 6.20: Attractor 6 of the extensive simulation of the anterior-posterior gene model.

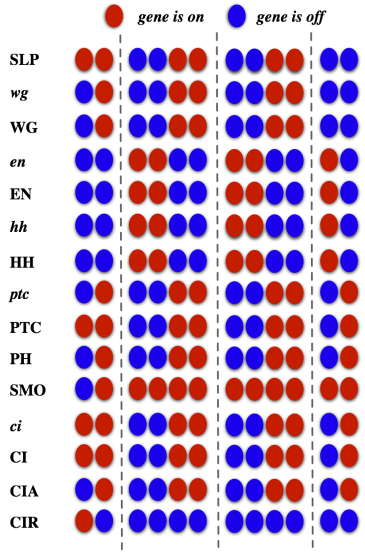


Figure 6.21: Attractor 7 of the extensive simulation of the anterior-posterior gene model.

Chapter 7

DETECTION OF PAIR-WISE GENE INTERACTION BY INFERRING THE THRESHOLD GENE MODEL

In this Chapter a method for inferring dependency among genes using data from DNA microarray experiments is presented. DNA microarrays are used to measure expression level of different genes [32, 82]. The procedure proposed in this chapter uses microarray data and constructs the gene interaction graph using the properties of threshold logic. The procedure can also be used to build the gene regulation model. However, in this chapter the discussion is only concerned with the accuracy of the gene interaction graph generated by the procedure. This work is an attempt to use the underlying circuit like behavior of genes (the circuit based model of gene regulation is gaining popularity [68]) to identify pair-wise interaction between genes.

The data provided for the DREAM2 network inference challenge was used (specifically the dataset InSilico3 which is part of Challenge 4). This data includes gene expression data from time course experiments. The data is used to identify interactions between these genes and construct the gene interaction graph. Since no additional biological knowledge is provided the procedure had to rely completely on the data provided. The result discussed here was chosen as the best performer among the entries that predicted the InSilico3 network in the DREAM2 inference challenge (in the category directed signed excitatory). Next, the technique is applied to an actual biological system. The method is also used to construct the gene interaction graph for a group of melanoma related genes [6]. Gene expression profile of 31 patients is used to construct the gene interaction graph. Then the gene interaction

graph obtained is validated using existing biological literature.

7.1 Method

The Procedure

In order to construct the gene interaction graph, given a set of all genes, G, a set of genes $G(g_t) \subseteq G$ are identified, such that each gene $g_s \in G(g_t)$, affects the expression of g_t (henceforth called the target gene). To identify $G(g_t)$, for the gene g_t, the threshold logic function $T(g_t)$, which best describes the regulatory function of g_t is inferred. $G(g_t)$ can then be obtained from $T(g_t)$, since $G(g_t)$ is simply the inputs to the function $T(g_t)$. As mentioned before threshold logic rules have been demonstrated to be expressive enough to describe gene interaction. The major contribution of this work is to identify gene networks using data from biological experiments, by making use of the concepts and theories from threshold logic research.

A threshold logic function is completely characterized by the inputs, input weights and threshold. In order to determine the threshold gene regulation rule for the target gene g_t, first the set of genes $G(g_t)$ that affect the gene expression of g_t, the input weights for each gene and the threshold have to be determined. It has been shown that the inputs that have the least impact on the output have the least absolute weight [37]. It would be good if the genes can be ordered in their order of impact so that the ones with less or no impact on gene g_t can be ignored. To do this two properties of threshold functions are used:

1. The ratio of the spectral coefficient of an input to its correlation with the output is a constant.

2. The ordering of input variables of a threshold function by their spectral coefficients is the same as the ordering of variables by their input weights.

(For proofs of the above two properties, refer to the Appendix).

From the above two properties, input genes can be ordered according to their impact on gene g_t, by ordering the genes in the descending order of the absolute value of their correlation with gene g_t (since the inputs with the least weight have the least impact [37]).

The time-series data (typically obtained from microarray experiments) is used to construct the threshold logic function $T(g_t)$. A time-series data gives the gene expression values for each gene at regular time steps. It is a matrix $TimeSeries$ of size nxm, where n is the number genes and m is the number of time steps. Therefore $TimeSeries[i,j]$ is the gene expression of gene i at time step j. It is assumed that the expression level of all the genes at time k (i.e ${g_a}^k, \forall g_a G$), is responsible for the expression of any gene g_t at time $(k+1)$ – ${g_t}^{(k+1)}$.

To generate the threshold logic rule $T(g_t)$ for gene g_t, first all the genes are ordered in the non-increasing order of their absolute correlation with gene g_t. This correlation value – $corr({g_a}^k, {g_t}^{(k+1)})$ is calculated between the value of gene g_a at time k and the value of gene g_t at time $(k+1)$, for all $k \in 1,2,....(m-1)$, where m is the number of time steps. After the value of $corr({g_a}^k, {g_t}^{(k+1)})$ is calculated for every $g_a \in G$, the genes are ordered in the non-increasing order of their absolute correlation values with g_t. This gives the ordering of all genes in the order of their impact on g_t.

The threshold rule $T(g_t)$ is derived, by using the correlation values as follows: One gene at a time (in the order described before) is added to $T(g_t)$. The input weights of these input genes are set to be equal to their correlation values with g_t. That is if g_a is an input of $T(g_t)$, $weight(g_a) = corr({g_a}^k, {g_t}^{(k+1)})$. The threshold of $T(g_t)$ is determined to be equal to the value of the least one-point. Using the Least Mean Square (LMS) Perceptron learning algorithm [92], these weights are fine-tuned such that the error (number of misclassified points) is minimized (as mentioned before a TE and the Perceptron are identical). More genes are added to the set of dependent genes as long as the error is reduced. The process of adding any more input genes to $T(g_t)$ is stopped when the number of misclassified points after adding the new input gene to $T(g_t)$ is more than the number of misclassified points without adding the new input gene to $T(g_t)$.

The detailed steps involved in determining the dependent genes for target gene g_t is given in Algorithm 19.

The InSilico Dataset

The first dataset on which the procedure was tested was the one provided by the DREAM2 conference for the DREAM2 network inference challenge (urlhttp://wiki.c2b2.columbia.edu/dream). The InSilico3 data which is part of the challenge 4 dataset is used in this work. This dataset consists of data from 3 different experiments Ð the heterozygous knockdown experiment, the null mutant experiment and the time-series experiments. For the proposed procedure only the data from the time-course experiments is required. This dataset consists of continuous expression values of 24 metabolites, 23 proteins and 20 genes.

Algorithm 19: Pseudo code for generating the gene interaction graph.

1: Determine $corr({g_i}^k, {g_t}^{k+1})$, genes g_i that are genes in the dataset.
2: $DepGenes = \{g_h\}$, where g_h is a gene such that $corr({g_h}^k, {g_t}^{k+1}) corr({g_i}^k, {g_t}^{k+1})$, $g_i \neq g_h$. Error = ř.
3: Initialize the TE for gene g_t, with inputs being $DepGenes$ (initialize ${w_g}^i = corr({g_i}^k, {g_t}^{k+1})$, $\forall g_i \in DepGenes$).
4: Evaluate the initial value of threshold T (T = the largest weighted sum for any input for which $g_t = 1$).
5: Use the LMS algorithm to learn weights of the equivalent Perceptron. Let $newError$ be the error of this Perceptron.
6: **if** $newError < Error$ **then**
7: $DepGenes = DepGenes \cup g_n$,
8: where g_n is a gene such that $corr({g_n}^k, {g_t}^{k+1}) corr({g_i}^k, {g_t}^{k+1})$, $g_i, g_n \in DepGenes$.
9: $Error = newError$.
10: GOTO Step 3.
11: **end if**
12: DepGenes are the dependent genes of gene g_t.

The continuous data was first discritized into 4 levels – 0, 1, 2, and 3. This granularity of discretization was chosen by inspection of the histograms of the expression values. Using the discretized data the correlation values were obtained and the procedure described was used to determine the dependent genes. There is however one modification to the standard procedure. Since the data is in 4 discrete levels, 3 TEs (cascaded TEs [3])are required. The first TE decides if the output is 3 or not, the second if the outputs is 2 or not, and the third if the output is 1 or 0. The set union of all the inputs genes of the three TEs decides the dependent genes for the target gene.

Gene Interaction Graph for Melanoma Genes

The above-described procedure was applied to an actual biological system to generate the gene interaction graph. Specifically it is used to find pair-wise interaction among melanoma disease-related genes. Ten genes are believed

to be closely associated with melanoma [99] (these are listed in Table 7.1). The gene expression profile for these 10 genes was obtained from 31 different melanoma tumors using DNA micro-arrays (refer to Bittner et al. [6]). Unlike the previous case, no time-course data was available. Gene profile of tumors is treated as a steady state data and the same procedure is applied. Since only steady state data is used $corr(gak, gt(k+1)) = corr({g_a}^{ss}, {g_t}^{ss})$, where ${g_a}^{ss}$ and ${g_t}^{ss}$ are expression values of genes ga and gt in the steady state. The gene expression is discretized into 3 levels – -1 (under expression), 0 (normal expression) and $+1$ (over expression), and two TEs per gene is needed to obtain the threshold rules which describe the gene regulatory rules. The dependent genes are obtained from the threshold regulatory rules as done in the previous experiment.

Gene	Entrez Gene Name	Synonym
pirin	pirin (iron-binding nuclear protein)	PIR
WNT5A	Wingless-type MMTV integration family, member 5A	
S100	S100 calcium binding protein B	S100B
PLGC1	phospholipase C, gamma 1	PHO-C
RET-1	ret proto-oncogene	
synuclien	synuclein, alpha (non A4 component of amyloid precursor)	SNCA
STC2	stanniocalcin 2	
MMP-3	matrix metallopeptidase 3 (stromelysin 1, progelatinase)	MMP3
MART-1	melan-A	MLANA
HADHB	hydroxyacyl-Coenzyme A dehydrogenase /3-ketoacyl-Coenzyme A thiolase/enoyl-Coenzyme A hydratase (trifunctional protein), beta subunit	

Table 7.1: The genes involved in melanoma manifestation and their synonyms.

A true time-series data is always preferred to use with this technique, since it captures the dynamics of the network more accurately. However in the absence of time-series data (which is often the case when working with biological systems) the steady state data can be used as was done in the case of melanoma related genes.

7.2 Results

The InSilico Dataset

After the proposed procedure was used to identify the dependent genes, the gene interaction graph was constructed. This gene interaction graph is shown in Figure 7.1. The InSilico3 data was generated *in-silico* and at the time the data was provided no biological knowledge regarding the gene products involved was given.

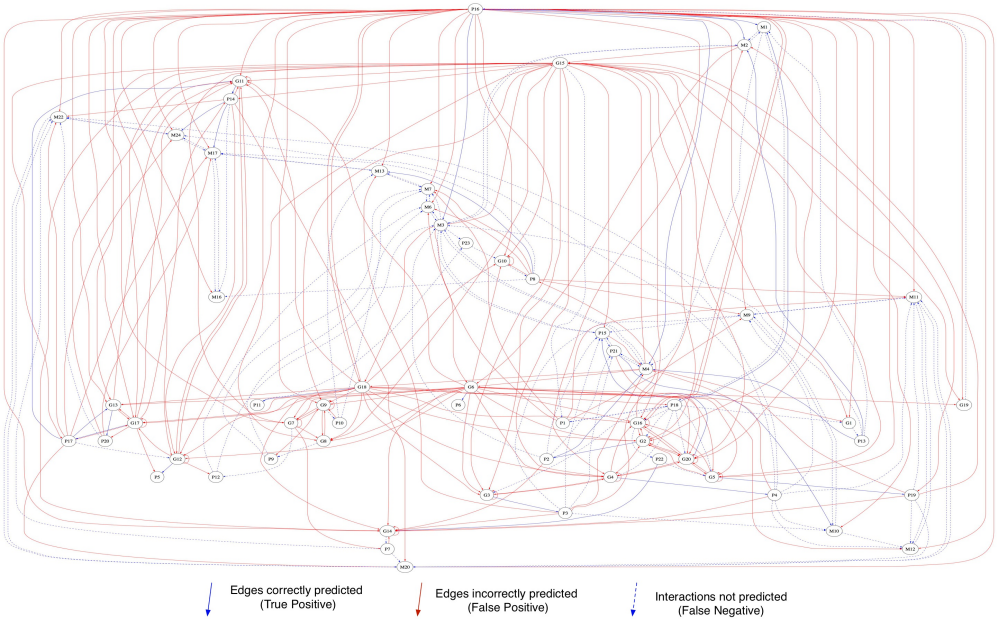


Figure 7.1: The predicted gene interaction graph for InSilico3 dataset.

The inferred network is compared against the gold standard. The statistics of this comparison is given in Table 7.2. Overall, many of the true interactions were not identified by the proposed method. The positive predictive value (PPV) was 12.87% and the sensitivity was only 21.67%. Higher negative predictive value (NPV) and specificity is misleading, as there exists significant bias toward non-interaction. The results of the DREAM2 challenge

were evaluated based on the Precision v/s Recall and the Receiver Operating Characteristic (ROC) curves. These are shown in Figure 7.2. For details of how these curves were obtained and more information on how the inferences were evaluated refer to Scoring Methodologies for DREAM2 (`http://wiki.c2b2.columbia.edu/dream/data/gold-standards/Scoring_Methodologies_for_DREAM2.doc`).

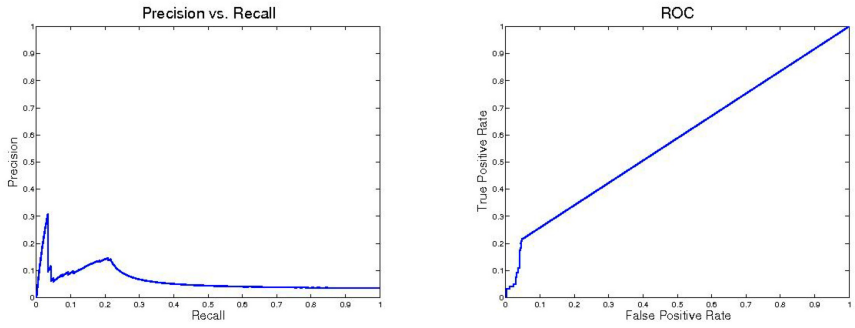


Figure 7.2: The precision versus recall and ROC curves obtained by comparing the predicted network against the InSilico 3 gold standard.

	Gold Standard		
Prediction	Edge exists	No Edge	
Inferred	26	176	PPV = 12.87%
Non-Inferred	94	4,193	NPV = 97.8%
	Sensitivity = 21.67%	Specificity = 95.97%	

Table 7.2: Statistical comparison of the predicted InSilico3 gene interaction graph against the gold standard network.

The Melanoma Dataset

After coming up with the threshold rules, the dependent genes for each gene were extracted from the rules. These were used to construct the gene interaction graph. The procedure not only identifies the pair-wise interactions, but also categorizes them into activation and inhibition. The transitive edges

are removed (e.g, the edge from MART-1 to MMP-3 is removed because MART-1 can inhibit MMP-3 via WNT5A) to emphasize only the strongest interactions. Outgoing edges from MMP-3 are removed as MMP-3 is the terminal node in the pathway involving these genes [99]. The final gene interaction graph is shown in Figure 7.3. Many different attempts have been made to infer the regulatory network for the genes involved in the WNT5A pathway and its neighborhood[27, 28]. This is an ongoing work and different pair-wise interactions predicted by this and the previous works needs to be validated.

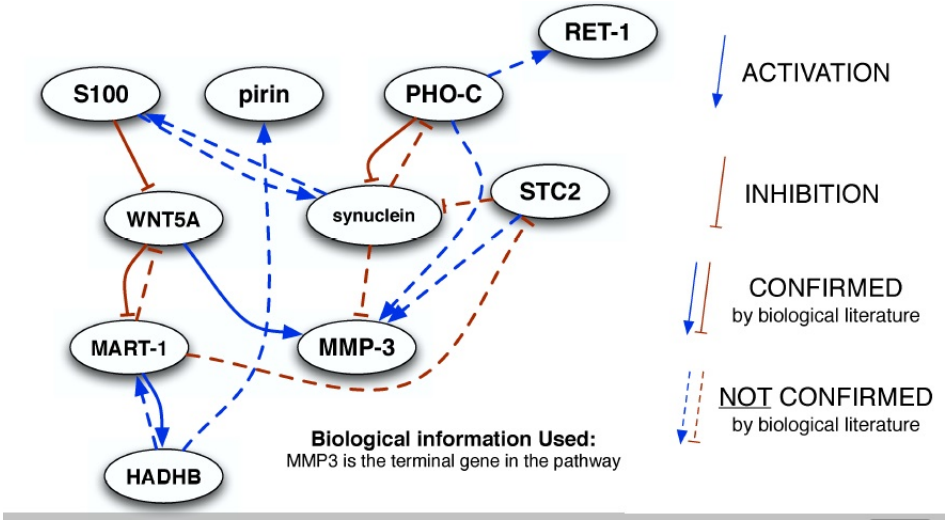


Figure 7.3: The gene interaction graph predicted for the melanoma dataset (transitive edges are removed).

These results are further discussed and pointers to future work are provided in the Discussion section.

7.3 Discussion

In-Silico Experiments

As seen from the results of the InSilico3 experiments it is clear that the procedure could not identify many of the edges present in the actual network (gold standard). This can be attributed to the following. Threshold logic can only capture relations that are mostly linear, and if the underlying process to generate InSilico3 data was non-linear, the proposed method could not have appropriately captured the functions used to generate the data. This implies the importance of the selection of appropriate mathematical models. However, when available biological data is limited, as often the case in biology, complex (i.e., non-linear) models cannot be used in general, due to errors involved in estimating necessary parameters, which increase as the complexity of model increases. More validation is required for the proposed method and this will be done in the future based on randomly generated networks. In the future studying the effect of the number of discrete levels on the accuracy of the model, based on a recent study [12] is planned. The automated discretization methods published recently which chooses the optimal number of discrete levels based on the given data [17] can also be integrated with the proposed approach.

Biological Data: Melanoma

The method however had more success in predicting the edges of the melanoma gene system. Many interactions could be verified by existing literature (see Figure 7.4). By utilizing existing literature and Ingenuity Pathway Analysis system (IPA, Ingenuity Systems, `http://www.ingenuity.com`),

the pathways obtained by the proposed method were investigated. Figure 7.4 shows a pathway constructed using IPA with the ten genes used in this study.

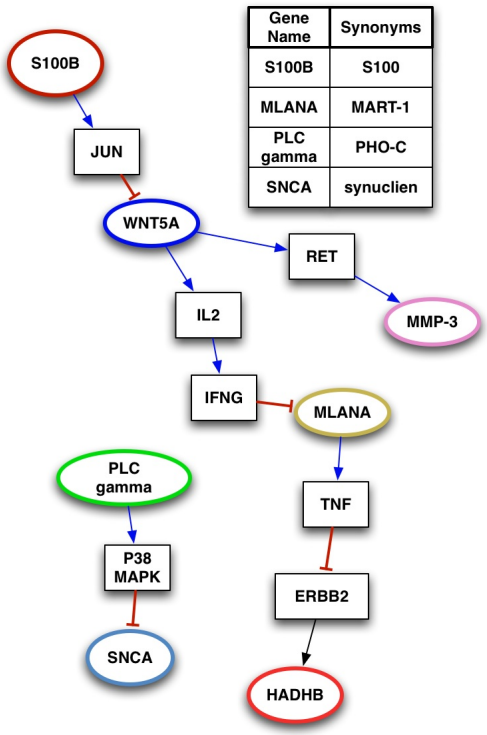


Figure 7.4: The pathways involving melanoma disease-related genes reported in literature.

As shown in Figure 7.3, S100 was predicted to inhibit the expression of WNT5A. From existing literature (as shown in Figure 7.4), S100 protein is known to increase the expression of JUN mRNA (and thus the JUN protein) [44] which in turn binds to the promoter region of the WNT5A gene [31]. This binding prevents WNT5A gene from being expressed. Through the mediation of JUN mRNA and protein, S100 inhibits the expression of WNT5A.

As per the model's prediction, WNT5A interacts with MMP-3. WNT5A causes an increased expression of RET mRNA [33] and the RET protein then

causes an increase in the expression of MMP-3 mRNA (and protein) [10]. Thus by the mediation of RET, WNT5A causes an increase in the level of MMP-3, as shown in Figure 7.4. Also shown in Figure 7.3 is that WNT5A and MART-1 interact with each other in inhibitory manner. WNT5A increases the expression of IL2 protein [80], IL2 protein increases the activation of IFNG gene [66], which in turn decreases the expression of MART-1 [63]. Therefore WNT5A inhibits the expression of MART-1 through a series of interaction involving other genes.

Our study predicted that PHO-C inhibits the expression of synuclein (SNCA). Campbell et al. [15] have shown that PLC-gamma (PHO-C) protein is necessary for the activation of P38-MAPK protein, which in turn binds to SNCA (synuclien) [49].

The TNF protein is increased by the anti-genic protein fragment from MLANA (MART-1) protein [29]. The TNF protein decreases the expression of ERBB2 protein [53]. The ERBB2 protein is involved in the expression of HADHB [62]. This suggests that MART-1 affects the expression of HADHB, which was also predicted by the proposed method (Figure 7.3).

Several other attempts have been made to predict the interaction between the core genes involved in melanoma manifestation, and this is still an active area of research[27, 28]. The author believes that the interactions suggested would aid in the identification of the actual bio-chemical pathways involved in the regulation of these genes.

Appendix

Lemma 7.3.1. *The ratio of the spectral co-efficient of an input to its Pearson correlation with the output is a constant.*

Proof. Consider a threshold function F and variable a that is an input of F and has a positive input weight.

Now in each row of the truth table of F, a can either agree with F (i.e $a = 0; F = 0$ or $a = 1; F = 1$) or it can disagree with F ($a = 1; F = 0$ or $a = 0; F = 1$). We first show that:

- For every case where $a = 1; F = 0$, there exists a case $a = 0; F = 0$.
- For every case where $a = 0; F = 1$, there exists a case $a = 1; F = 1$.

The first condition is trivial as a has a positive weight. Switching a form 1 to 0 when the other inputs are intact will not change the value of F from 0 to 1. Now since F is threshold it is also unate. Therefore the property $F|(a = 0) \subseteq F|(a = 1)$ holds. The second condition follows directly from this property.

Since the above two conditions hold, every row of the truth table on which a and F disagree can be paired uniquely to another row where they agree.

After such a pairing, the number of rows that are not part of any pair is equal to the spectral coefficient $s(a)$ (by definition).

Now consider the value of the product of the difference of each variable (a and F) and their mean:

$$H = (a - \mu_a)(F - \mu_F)$$

($\mu_a = 1/2$: in any truth table an input is 1 and 0 equal number of times)

This value is different for different value of a and F:

- If $a = 0; F = 0$: $H = \mu_F/2$
- If $a = 0; F = 1$: $H = -(1 - \mu_F)/2$
- If $a = 1; F = 0$: $H = -\mu_F/2$
- If $a = 1; F = 1$: $H = (1 - \mu_F)/2$

Note that for any of the previously described pairing the value of H for one is -1 times the value of H for the other.

$$var(a, F) = (\sum H)/(n - 1)$$

.

The values of H in the pairs cancel each other and we're left with:

$$\therefore var(a, F) = (s(a)/2)\sum H_{(a=1;F=1)} + (s(a)/2)\sum H_{(a=0;F=0)}/(n - 1)$$

.

Substituting the value of $var(a, F)$, in the formula for $corr(a, F)$ and simplifying, we get:

$$\kappa = \frac{corr(a, F)}{s(a)} = \frac{1}{2\sqrt{pq}}$$

.

where p is the number of times $F = 1$ and q is the number of times $F = 0$, which are constants for any given F.

This same formula can be similarly derived if a has a negative weight.

□

Lemma 7.3.2. *The ordering of input variables of a threshold function by their spectral co-efficients is the same as the ordering of variables by their input weights.*

Proof. Spectral co-efficients are closely related to *Chow parameters* used in the study of threshold logic [47, 73]. From Theorems 5.1.4 and 5.1.5 in [73] and Lemma 1, Lemma 2 follows. □

Chapter 8

AUTOMATED PROCEDURE FOR MODELING GENE REGULATORY DYNAMICS

In this Chapter a novel gene regulatory model based on threshold logic is proposed. The approach is developed by a combination of threshold logic properties and perceptron learning techniques. This work does not focus on determination of the pair-wise interaction among genes (which is discussed in the previous chapter). Instead, the objective of this work is to generate a model that will describe and predict phenomena associated with a biological system. The utility of the approach is demonstrated by modeling a cellular system of 50 genes. The model could effectively replicate both the steady state and the transient behavior of genes.

Threshold logic was first used to model gene regulatory networks in order to describe embryo development in *Drosophila* (see Chapter 6). A group of genes known as segment polarity genes are responsible for the formation of segments in a Drosophila embryo [94]. Since this group of genes is relatively small and the interaction between genes (and their gene products) is well understood, a threshold model could be constructed by hand that accurately replicates the biological role of segment polarity genes. This work however is specific and does not provide a generic method to construct these models. Modeling larger biological systems is infeasible by this method. To alleviate this drawback, an automated methodology that can construct the threshold gene model starting with gene expression data obtained from biological experiments is proposed.

There are several advantages of the proposed model over the generic

Boolean logic model. Principal among them is the combinatorial algorithm for the construction of the model. This method allows for the use of a finite number of discrete levels, and is not restricted to two levels as in generic Boolean logic. Implementing multi-level outputs in Boolean logic is also possible but significantly increases the complexity of the model. Threshold logic is intuitively related to the underlying gene system. This is discussed in detail later in the chapter.

The aim of this work is to develop techniques to infer gene regulatory models from gene expression data obtained from biological experiments. The model developed is easy to infer and the biological reasoning for the choice of threshold logic is provided. A series of cross validation experiments validate the merit of the proposed approach. Finally the utility of the model is demonstrated by using it to model the action of a group of 50 genes, to faithfully predict dynamic behavior of genes given in the data. Since the gene interaction graph is not available a priori the details of how to construct it from the available data is given. However, the focus of this work is not to determine the gene interaction graph. Instead the aim of this work is to generate a gene regulation model using available biological data that can mimic a system of interacting genes.

8.1 Method

Threshold logic (TL) is a proper subset of Boolean logic. Since threshold logic is similar to the single-layer perceptron with a step activating function, it implements a linearly separable function [79]. Threshold logic differs from the perceptron in that the input weights and thresholds are restricted to be integers. This is not a requirement of threshold logic but since its development

is closely linked to circuit design, integer weights are convenient as it makes the sizing of transistors (which is used in many TL circuit implementations) straightforward. It has to be noted however that restricting the weights to integers does not alter the expressiveness of threshold logic [73]. Another difference is that threshold logic is more suited for the study of Boolean logic properties (such as symmetry detection etc.) whereas perceptrons are mainly used to study properties of linearly separable functions.

Since the threshold element and the perceptron are closely linked a combination of threshold logic and perceptron learning methods can be used to infer the gene regulation rules from expression data. Another useful technique used here is a feed-forward threshold circuit. This construction is useful when more than two discrete levels are used for gene expression. A single threshold element can be used to describe a multi-output function. In general $n-1$ thresholds are needed to implement an n level output function. Even so, the feed-forward network of threshold gates is used as it simplifies the integration of perceptron learning into the methodology.

Input Data

The data used to demonstrate the utility of the proposed model is provided as part of the DREAM2 network inference challenge (`http://wiki.c2b2.columbia.edu/dream`). In this chapter the dataset InSilico1, which is one of the three data sets provided as part of the DREAM2 challenge 4 is used.

The InSilico1 data consists of gene expression data for a cellular system of 50 genes in different experiments. The data provided is obtained from three kinds of experiments:

1. Data from the first set of experiments is gene expression in the steady state for wild type and heterozygous knock-down strains of each gene.
2. The second set of data is the steady state for wild type and null mutant strains of each gene.
3. The third set consists of 23 time course experiments in which the expression of each gene is recorded in discrete time steps (26 for each experiment). Each experiment differs from the other in the initial state with which the experiment begins.

The first two data-sets (knock-down experiments) are used to generate a preliminary gene interaction graph which indicates target genes that are affected by the perturbation of a gene. This is done by comparing the knock-down (or null mutant) steady state expression against the wild type expression. The time course data is then used to generate the threshold logic model and to construct the final gene interaction graph from the preliminary gene interaction graph.

Discretizing the Data

The first step in generating the gene interaction graph and building the model is discretization of the data. The given gene expression is a continuous value. Using this data a histogram of gene expression of each gene is plotted. Then the clustering of gene expression in this histogram is used as a guide to discretize the gene expression values. It was empirically observed that 3 discrete levels were enough for the first two data sets. However, the gene expression data for the time course experiments was discretized into 4 levels (chosen again by empirical observation). Discretization has two distinct

benefits. First is to infer the preliminary gene interaction graph. Continuous data sometimes varies little between different experiments. These minute variations may be due to other factors and not necessarily due to the action of genes. Discretized data provides a better way to compare variations in gene expression. Secondly, since threshold logic works with discrete inputs and outputs, discrete data can be used as input data for the model.

Constructing the gene Interaction Graph

After discretizing the data from the heterozygous knock-down and the null-hypothesis experiments a preliminary gene interaction graph is constructed. The gene interaction graph has one node for each gene and directed edges representing interactions between the genes. This is constructed by comparing the gene expression (when a gene is knocked down or completely unexpressed) against the gene expression in the normal case (wild-type). The discretized gene expression from both heterozygous and null knock down experiments can also be found in supplementary material. These inferred edges are tentative as the effect of a gene on another can be due to direct interaction or due to a series of intermediate gene interactions. The data from the time course experiments is used to infer the final gene interaction graph.

Inferring Threshold Logic Network from Data

The algorithm uses a combination of threshold logic properties and perceptron learning techniques to infer a cascade of TEs for each gene. As mentioned earlier the single perceptron with the step function as the activating function is identical to a threshold element. The only difference being that in the

perceptron the threshold T of a threshold element is treated as another input which is always 1, and whose weight is $-T$. It is easy to see that this does not change the function implemented by the perceptron. Perceptron learning algorithms start out with an initial assignment of weights and iteratively adjust the weights by use of training data [43]. The assignment of initial weights is important for the quality of the solution and the time taken to generate it. In case the perceptron is to implement a function that is not linearly separable, it generates a set of weights so that the mean square error is minimal. The perceptron learning algorithm used is the Least Mean Square (LMS) algorithm [92].

The basic idea of the algorithm is to infer threshold rules that capture the behavior of each gene using minimal number of inputs (genes). The model consists of three TEs for each gene. The input of these threshold elements are other genes. Three TEÕs are needed since the time course data is discretized to have 4 discrete levels – 0, 1, 2 and 3. The cascade of TEs (feed-forward circuit) that determine the value of a given gene is as shown in Figure 8.1(a). If the input combination makes the weighted sum of TE 3 greater than its threshold the output is 3. If not, TE 2 is used to determine if the output is 2. If not, TE 1 is used to determine if the output is 1 or 0 (Figure 8.1(b)).

The ideal model will determine the output of each gene accurately for each gene. However inferring the right model (the right set of TEs for each gene) may not be possible using the given data. One reason is that there may be inherent contradictions within the given data. Example: a given set of inputs may correspond to the gene value 3 in one case and 1 in another. Therefore the algorithm is designed to minimize the error in inference and not to eliminate error all together.

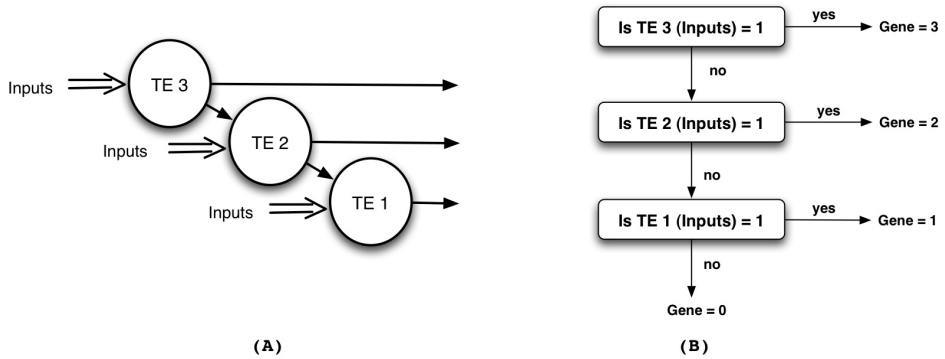


Figure 8.1: A feed-forward threshold circuit that describes a gene in the threshold model.

Another important feature of the algorithm is that it attempts to minimize the number of input genes required to determine the value of a gene. The tentative gene interaction graph gives all the genes on which the expression of a gene may depend. However, not all genes have the same impact. Therefore the algorithm picks the minimum set of inputs (ones that have the most impact) that can infer the value of the gene accurately. This is based on the property that the inputs that have the least weight (known as donÕt care variables) have the least impact in determining the output of a threshold function [37].

The outline of the algorithm is listed in Algorithm 20. It shows how the feed-forward circuit of TEs to describe one gene ($Gene$) in the threshold model is constructed.

The algorithm uses the time course data to generate correlation values between any two pairs of genes. Correlation values provide a good estimate of the weights (it can be shown that in a threshold function the ordering of input variables by their correlation to the output is the same as the ordering of variables by their input weights). The perceptron learning algorithm further fine-tunes the weights (line 8). In line 1 all the genes that have an outgoing

Algorithm 20: $generateThresholdRule(TCData, correlationData, prelimGIG, Gene)$

```
1:  allInputs = Inputs(Gene, prelimGIG)
2:  for all level ∈ (3downto1) do
3:    for all InputGenes ∈ (1, length(Inputs(Gene))) do
4:      T = guessThreshold(Gene, noofInputGenes)
5:      currentInputs = allInputs[0 : noofInputGenes]
6:      InputandWeights = [(g_i, w_i), T], ∀g_i ∈ currentInputs.
7:      * Input weights are assigned as the correlation of the input-output *
8:      Rule[Gene][level] =
        perceptronLearningAlgorithm(TCData, InputandWeights)
9:      if misclassifiedpoints > previousmisclassifiedpoints then
10:       break
11:     end if
12:   end for
13: end for
```

edge to Gene in the preliminary gene interaction graph (GIG) are added to the list $allInputs$. These genes are sorted according to their correlation with $Gene$. Line 2 of the algorithm makes sure that all three TEs are inferred (these three TEs together form the feed-forward circuit which describes the gene in the model. The range of level goes from 3 to 1 as the time course data is discretized into 4 discrete levels – 0, 1, 2, 3). Line 3 is used to progressively increase the number of inputs until a good TE (one with the least error) is obtained. In line 4 the initial value of the threshold is decided as the minimum of all the weighted sums that correspond to $level$ (which is 3, 2 or 1 depending on the iteration). The loop (line 3 to 11) is exited if the current $Rule$ has more misclassified points than the rule inferred in the previous iteration. The one with the least error is the one chosen as part of the TL model to describe the action of $Gene$. The rules iteratively get better (have fewer errors) as more inputs are added (Figure 8.2), but at some point the error increases. This is when the algorithm stops and uses the TL Rule inferred in the previous iteration. *Example:* As shown in Figure 8.2, for $G2$ gene the algorithm

iteratively generates TL rules with 1, 2 and 3 input genes that have the highest correlation with $G2$. Since with 3 inputs the error is greater than with just 2 inputs, the algorithm quits and chooses the first two genes to be the input. When the feed-forward circuit of TEs for each gene is inferred the construction of the TL model is complete.

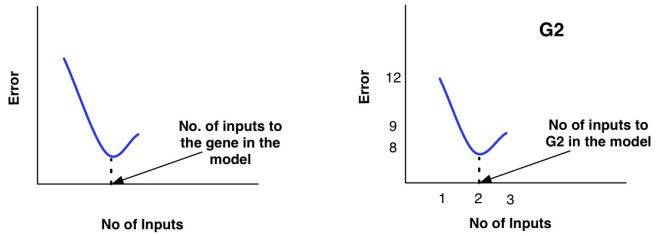


Figure 8.2: Determination of inputs to a threshold logic gene function by the algorithm.

The flow chart in Figure 8.3 describes the steps involved in generating the threshold logic gene regulation model by using the proposed procedure.

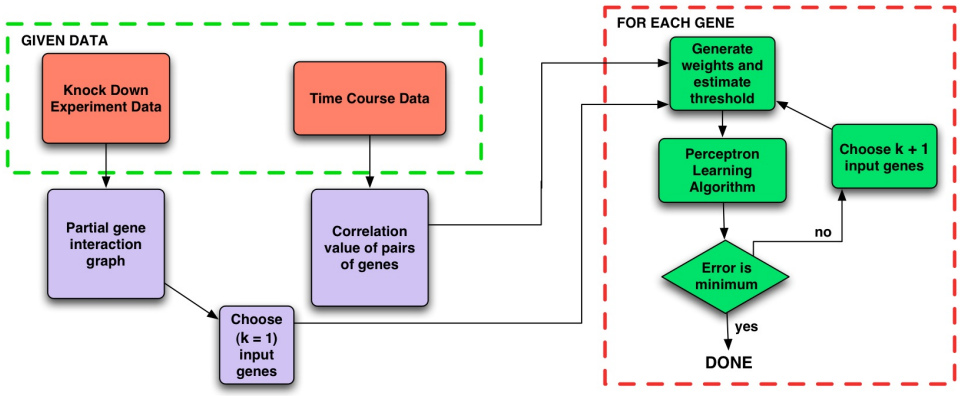


Figure 8.3: Flow chart of the proposed procedure.

Simulation

Once the model is generated it can be used to simulate the biological system it models. To check the accuracy of the model the time course data is used.

There are 23 different time course experiments and each starts with a different initial state of genes. Each case is simulated in the model to generate the steady state and the transient behavior. This simulated data is then compared against the actual data.

8.2 Results and Discussion

Validity of Using Threshold Logic to Model Gene Regulation

Since threshold logic is based on assigning weights to inputs it makes the modeling of gene interaction more intuitive. A gene can affect another gene by either activating it or inhibiting it [89]. In a threshold element (an element that implements a threshold logic function) an input that has a positive weight increases the weighted sum and thus attempts to make the output 1. This is similar to an activating gene as it too tries to switch on the target gene. If the input weight is negative then it reduces the weighted sum. This is equivalent to the action of an inhibitor, which tries to prevent the gene from expressing. Thus an activator gene can be modeled within threshold logic by using a positive input weight and an inhibitor can be modeled by assigning it a negative weight. It should be noted here that an input that has positive weight occurs in the positive form in the Boolean expression and an input that has negative weight occurs in the negated form [58]. Below is the biological reasoning to why threshold logic is a useful model for gene regulatory networks.

If two genes (say g_a and g_b) are known to activate another gene g_c and gene g_a has more impact than gene g_b; the threshold model can use a greater input weight for g_a than g_b in order to model this. This level of influence of a gene on another gene is neither intuitive nor straightforward in the Boolean

gene model.

As mentioned earlier threshold functions are a proper subset of Boolean functions. Even so a network of threshold gates can implement any Boolean function [73]. Threshold functions are also a subset of a class of functions called unate functions. A unate function is a Boolean function that has a Boolean representation in which every input variable appears only in the positive or negated form [40]. For example: $ab' + cd$ is a unate function since a appears in positive form a but not in the negated form a' ; b appear only in the negated form; c and d appear only in the positive form.

The relation between Boolean, unate and threshold functions is a follows:

Threshold functions $\subseteq$ Unate functions $\subseteq$ Boolean functions.

A Boolean function is unate if and only if the negative co-factor is contained in the positive co-factor (in case of an input occurring in the positive form) or vice-versa (input in negative form) [40]. Example: Consider the function $F = ab' + cd$. The positive co-factor of F with respect to a is $F(a = 1) = b' + cd$. Similarly the negative co-factor $F(a = 0) = cd$. It is clear that $F(a = 1) \subseteq F(a = 0)$. Similarly $F(b = 1) \subseteq F(b = 0)$.

First it is shown that under the assumption that a gene (say g_a) can either activate another gene (g_b) or inhibit it (but never both), the Boolean function used to represent the interaction of genes is a unate function. The assumption made is biologically reasonable as a gene inhibits or activates another gene by binding itself to the regulatory regions of the gene [89]. Hence the same gene when binding to the target gene cannot activate it under

one condition and inhibit it under some other condition.

Consider a gene g_z that is activated by gene g_a. Let the set of genes that affect g_z be GI. Therefore the Boolean function of g_z will have the elements of GI as inputs. Consider any input combination where $g_a = 0$ and for which $g_z = 1$. Now keeping all other inputs intact switching g_a from 0 to 1 will not change the value of g_z (since g_a is an activator).

Therefore:

$$g_z|(g_a = 0) \subseteq g_z|(g_a = 1).$$

Similarly if gb is an inhibitor of gene gz it can be shown that

$$g_z|(g_b = 1) \subseteq g_z|(g_b = 0).$$

Therefore the Boolean function g_z is a unate function.

It is now argued that the Boolean function representing gene interaction can be represented using threshold logic. It has been observed that the degree of separation between any two nodes in any cellular network is very small [52]. In such a scenario, any gene can inhibit any other gene through a series of regulatory interactions (which may involve other genes). In the Boolean representation of the gene function this would mean replacing a gene by the negation of another gene. We know that in a threshold function F replacing any variable x by y' will retain its threshold-ness (i.e it remains unate) [98]. Since weÕve already shown that gene functions are unate and any such variable replacements should retain the unateness property, is is assumed that gene regulatory functions are threshold functions. This argument only claims that if gene interaction is described by a threshold

function the assumption that a gene is either an activator or an inhibitor is not violated. This argument can be improved if it can be shown that a gene regulatory function has to be a threshold function, if the assumption that a gene is either an activator or a inhibitor (never both) has to be valid.

The performance evaluation of the model prediction

For this work the InSilico1 data provided in DREAM2 Challenge (`http://wiki.c2b2.columbia.edu/dream`) is used to illustrate the utility of the threshold logic. Since the true model is not known a priori and the network is not real biological system, no attempt was made to compare the graphical representation of inferred network. However, the inferred gene interaction graph is provided in the supplementary material.

Once the model is constructed from experimental data obtained from time-course experiments, it can be simulated to generate the predicted behavior which can then be compared the data used [12]. This is a more difficult problem than inferring the graphical representation of the gene network since networks with similar graphical representation could lead to significantly different dynamic behaviors if underlying functional relationships (rules) are different.

Two types of comparisons are made – the steady state gene expression and the transient gene expression before reaching the steady state. The steady states in the 23 experiments of the given data are identical and are as shown in Figure 8.4. The predicted steady state obtained by simulating the proposed model is also shown in the Figure. The 23rd time course data prediction did not match with the actual data for most genes. So only the steady state obtained for 22 out of the 23 experiments is reported

(considering the 23rd experiment as an outlier). Some genes attain different steady states for different experiments. These are also listed in the Figure. It can be seen from the table that for most of the genes (46 out of 50) the model could accurately predict the steady state.

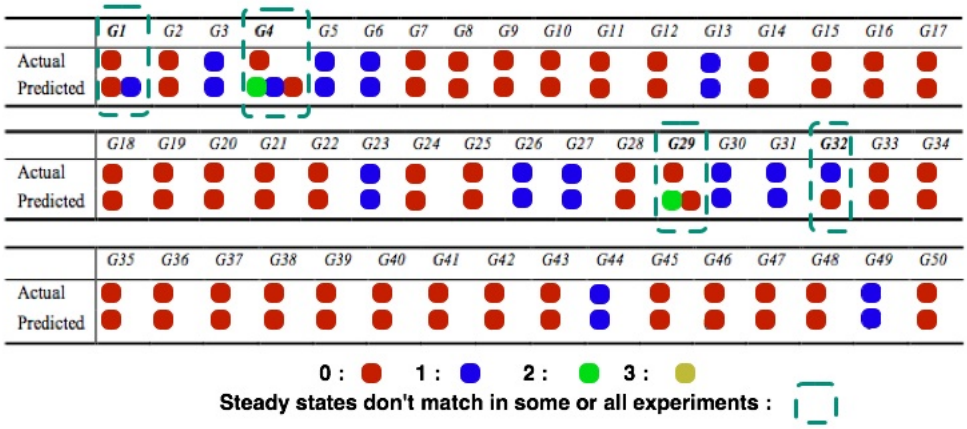


Figure 8.4: Actual steady state and the model predicted steady state.

To compare the predicted transient behavior with the actual transient data, the normalized mean absolute difference of two trajectories for each gene is calculated as follows:

$$\varepsilon_i = \frac{E[|{g_i}^p - {g_i}^a|]}{\delta_i}$$

where ${g_i}^p$ and ${g_i}^a$ are predicted and actual trajectories of each gene, respectively, and i is the range of the actual trajectory.

In order to verify the correctness of the proposed procedure, a 5-fold cross validation procedure was employed where the 23 time course experiments were randomly divided into 5 different groups; the data from 4 groups was used to build the model and the fifth group was used to test it. The process was repeated 15 times.

The majority of the genes have less than 2% error. This demonstrates that the method can be used for modeling a generic cellular system from the available experimental data and the model will reliably predict the results of other experiments. Overall error averaged over all genes is 3.39%. Table 8.1 gives a more detailed picture by listing the error range in which each gene lies (for the cross-validation experiments).

Error	$< 2\%$	$2 - 5\%$	$5 - 20\%$	$> 20\%$	**Avg. Error**
Genes	G2, G3, G5-14, G16, G17, G19-21, G24,	G1, G25 G27, G28, G30, G31, G33-45, G47-G50	G4, G15, G18, G22,	G29, G32 G23, G26, G46	3.7%

Table 8.1: Error metrics of genes (for cross validation experiments).

To give a better idea about the accuracy of the transient behavior predicted by the model, it is compared with the actual transient behavior (shown in Figure 8.5). It shows the plots of the actual and simulated data for gene $G3$ and gene $G22$. These plots compare the actual continuous data against the simulated discrete data (after it is normalized). As seen from the Figure, the threshold logic model was able to predict the dynamic behavior of each gene accurately. A major reason for this accurate prediction is the use of multi-level threshold logic to capture multi-level discrete states of gene expressions. Both $G3$ and $G22$ genes affect their own expression. This is true for a majority of genes ($> 60\%$ of genes affected their own expression).

In the future it is worth investigating how the accuracy of the model varies with the number of discrete levels used. As can be seen from Figure 8.5 the predictions generated are systematically 'faster' than the actual data. This could either be because of the number of discrete levels used or it could be inherent to the threshold model itself. The ability of the presented approach to model actual biological systems will also be tested in the future. The model

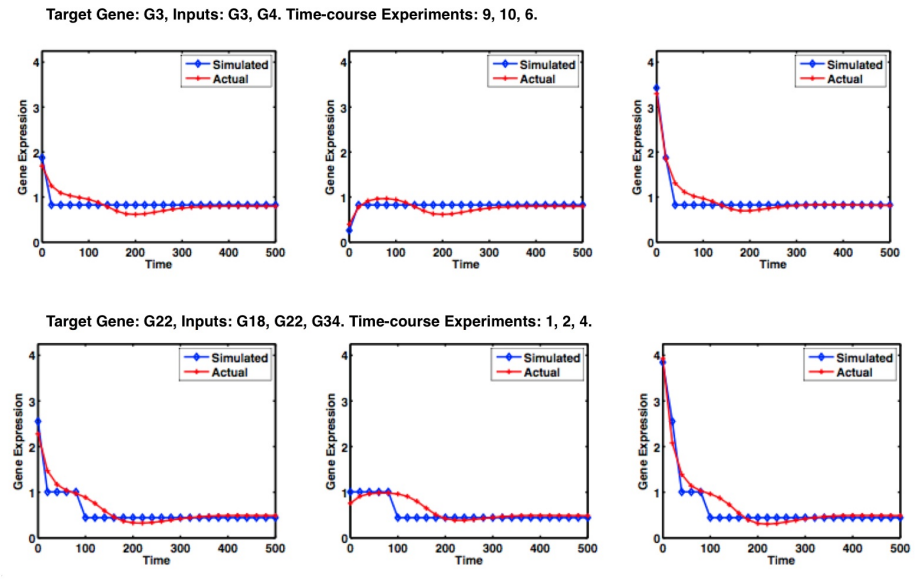


Figure 8.5: Comparison of the given data with the model generated data.

currently is deterministic. Gene models need to model the inherent non-determinism that exists in biological systems. The model can be improved to incorporate non-deterministic behavior.

Chapter 9

Conclusion

In conclusion, a review of the significant contributions of this thesis is now presented. Firstly, the thesis addresses the issue of threshold logic design automation. The main contribution of this thesis lies in the synthesis and equivalence checking of threshold logic. As illustrated in the thesis an crucial problem to do this is the identification of threshold functions.

This thesis proposes a novel co-factoring based algorithm to identify threshold functions. This is an alternative to the traditional ILP formulation. Since it is a co-factoring based method, it is well suited for a decomposition based synthesis method. The utility of this method is demonstrated in this thesis. However, many improvements to the proposed procedure can be made. Some such improvements include exploring decomposition heuristics that have not been tested in this work. The thesis also proposes 3 other synthesis methods and a novel equivalence checking method.

The second half of the thesis deals with the demonstration of threshold logic as a viable model to understand gene regulation and other complex biological processes. The thesis demonstrates the relevance of threshold logic for this purpose by modeling the embryo development in *Drosophila*. The model proposed could also model other gene systems (like the one provided by the DREAM2 contest). The largest system tested and validated in this work is one with 50 genes. In the future the model could be applied to larger gene systems. Other model enhancement, like non-deterministic behavior and the effect of external conditions can also be incorporated into the model.

REFERENCES

[1] Reka Albert and Hans G. Othmer. The topology of the regulatory interactions predicts the expression pattern of the segment polarity genes in Drosophila melangogaster. In J. Theor. Biol., 2003.

[2] M. A. Arbib. The handbook of brain theory and neural networks. 2002.

[3] M.J. Avedillo and J.M Quintana. A Threshold Logic Synthesis Tool for RTD Circuits. In Euromicro Sym. on Dig. Syst. Design, 2004.

[4] P. Avouris, J. Appenzeller, R. Martel, and S. J. Wind. Carbon nanotube electronics. Proceedings of the IEEE, 91(11):1772–1784, 2003.

[5] Valeriu Beiu, Jose M. Quintana, and Maria J. Avedillo. VLSI implementations of threshold logic-a comprehensive survey. In IEEE Transactions on Neural Networks, volume 14, 2003.

[6] M. Bittner, P.S.J. Meltzer, Y.D. Chen, Y. Jiang, E. Seftor, M. Hendrix, M. Radmacher, R. Simon, Z. Yakhini, A. Ben-Dor, N. Sampas, E. Dougherty, E.L. Wang, F. Marincola, C. Gooden, J. Lueders, A. Glatfelter, P.C.A. Pollock, J. Carpten, E. Gillanders, D. Leja, K. Dietrich, C. Beaudry, M. Berens, D. Alberts, V. Sondak, N. Hayward, and J. Trent. Molecular classification of cutaneous malignant melanoma by gene expression profiling. Nature, pages 536–540, 2000.

[7] E. P. Blair and C. S. Lent. Quantum-dot cellular automata: an architecture for molecular computing. In Proc. of SISPAD 2003.

[8] N. Bohr. Atomic structure. Nature, 107:104–107, 1921.

[9] Shekhar Borkar. Electronics beyond nano-scale cmos. In Proc. of DAC, 2006.

[10] Maria Grazia Borrello, Luisella Alberti, Andrew Fischer, Debora Degl'Innocenti, Cristina Ferrario, Manuela Gariboldi, Federica Marchesi, Paola Allavena, Angela Greco, Paola Collini, Silvana Pilotti, Giuliana Cassinelli, Paola Bressan, Laura Fugazzola, Alberto Mantovani, and Marco A. Pierotti. Induction of a proinflammatory program in normal human thyrocytes by the RET/PTC1 oncogene. Proceedings of the National Academy of Sciences of the United States of America, 102(41):14825–14830, 2005.

[11] D. Brand. Verification of large synthesized designs. In Proc. ICCAD, pages 456–459, 1993.

[12] Marcel Brun, Seungchan Kim, Woonjung Choi, and Edward R. Dougherty. Comparison of gene regulatory networks via steady-state trajectories. EURASIP Journal on Bioinformatics and Systems Biology, 2007, 2007.

[13] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput., 35(8):677–691, 1986.

[14] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers, 35:677–691, 1986.

[15] Kerry S Campbell. Signal transduction from the b cell antigen-receptor. Current Opinion in Immunology, 11(3):256 – 264, 1999.

[16] Webster K. Cavenee and Raymond L. White. The genetic basis of cancer. Scientific American, 272(3), 1995.

[17] Tae-Hoon Chung, Marcel Brun, and Seungchan Kim. Quantization of global gene expression data. pages 187–192, Dec. 2006.

[18] Joel E. Cohen. Mathematics is biology's next microscope, only better; biology is mathematics' next physics, only better. PLoS Biology, 2(12), 2004.

[19] F. S. Collins, M. Morgan, and A. Patrinos. The human genome project: lessons from large-scale biology. Science (New York, N.Y.), 300(5617):286–290, Apr 11 2003.

[20] Stephen A. Cook. The complexity of theorem-proving procedures. In Proceedings of the third annual ACM symposium on Theory of computing, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[21] Francis Crick. Central dogma of molecular biology. Nature, 227:561–563, August 1970.

[22] H. de Jong. Modeling and simulation of genetic regulatory systems: a literature review. Journal of computational biology : a journal of computational molecular cell biology, 9(1):67–103, 2002.

[23] Rina Dechter. Constraint Processing. Morgan Kaufmann, 2003.

[24] Michael Dertouzos. Threshold Logic:A Synthesis Approach. MIT Press, 1965.

[25] Public Domain. MCNC Benchmark Circuits. http://www.cbl.ncsu.edu/CBL_Docs/Bench.html, 1991.

[26] Edward R. Dougherty, Jianping Hua, and Michael L. Bittner. Validation of computational methods in genomics. Current Genomics, 8:1–19(19), 2007.

[27] Thomas A. Easton and Anonymous. Beyond the algorithmization of the sciences. Communications of the ACM, 49(5):31–33, 2006.

[28] Ellen M. Sentovich et al. SIS: A System for Sequential Circuit Synthesis. Technical report, Department of EECS, UC Berkeley, CA, 1992.

[29]Eric L et al. Inhibition of antigen-induced T cell response and antibody-induced NK cell cytotoxicity by NKG2A: association of NKG2A with SHP-1 and SHP-2 protein-tyrosine phosphatases. European Journal of Immunology, 28(1):264 – 276, 1998.

[30] Eileen E. M. Furlong, Erik C. Andersen, Brian Null, Kevin P. White, and Matthew P. Scott. Patterns of gene expression during drosophila mesoderm development. Science, 293(5535):p1629 –, 2001.

[31] Michael J. Gerdes, Maxim Myakishev, Nicholas A. Frost, Vikas Rishi, Jaideep Moitra, Asha Acharya, Michelle R. Levy, Sang-won Park, Adam Glick, Stuart H. Yuspa, and Charles Vinson. Activator Protein-1 Activity Regulates Epithelial Tumor Cell Identity. Cancer Res, 66(15):7578–7588, 2006.

[32] Greg Gibson. Microarray analysis. PLoS Biology, 1(1), 10 2003/10/1.

[33] Goncalo Castelo-Branco et al. Differential Regulation of Midbrain Dopaminergic Neuron Development by Wnt-1, Wnt-3a, and Wnt-5a. Proceedings of the National Academy of Sciences of the United States of America, 100(22):12747–12752, 2003.

[34] A. F. Gonzalez and P. Mazumder. Comparison of bistable circuits based on resonant-tunneling diodes. pages 493–498, 2003.

[35] Tejaswi Gowda, Samuel Leshner, Sarma Vrudhula, and Seungchan Kim. Threshold logic gene regulatory model: Prediction of dorsal-ventral patterning and hardware based simulation of drosophila. In Proceedings of the International Conference on Biomedical Electronics and Devices, pages 212 – 219, Funchal, Madeira, Portugal, 28 January 2008.

[36] Tejaswi Gowda, Samuel Leshner, Sarma Vrudhula, and Goran Konjevod. Synthesis of threshold logic using tree matching. In Proceedings of the European Conference on Circuit Theory and Design (ECCTD), Sevilla, Spain, 26 August 2007.

[37] Tejaswi Gowda and Sarma Vrudhula. A decomposition based approach for synthesis of multi-level threshold logic circuits. In Proceedings of the Asia and South Pacific Design Automation Conference, pages 125 – 130, Seoul, Korea, 21 January 2008.

[38] Tejaswi Gowda, Sarma B. K. Vrudhula, N. Kulkarni, and Krzysztof S. Berezowski. Identification of threshold functions and synthesis of threshold networks. IEEE Trans. on CAD of Integrated Circuits and Systems, 30(5):665–677, 2011.

[39] V. V. Gursky, J. Reinitz, and A. M. Samsonov. How gap genes make their domains: An analytical study based on data driven approximations. Chaos, 11:132–141, March 2001.

[40] Gary D. Hachtel and Fabio Somenzi. Logic Synthesis and Verification Algorithms. Boston: Kluwer Academic Publishers, 1996.

[41] F. O. Hadlock and C. L. Coates. Realization of sequential machines with threshold elements. IEEE Trans. Computers, 18(5):428–439, May 1969.

[42] Jeff Hasty, David McMillen, Farren Isaacs, and James J. Collins. Computational studies of gene regulatory networks: in numero molecular biology. Nat Rev Genet, 2(4):268–279, 04 2001/04//print.

[43] Simon Haykin and Anonymous. Neural Networks: A Comprehensive Foundation, volume 2nd. {Prentice Hall}, 1998.

[44] T. Herdegen and J. D. Leah. Inducible and constitutive transcription factors in the mammalian nervous system: control of gene expression by jun, fos and krox, and creb/atf proteins. Brain Research Reviews, 28(3):370 – 490, 1998.

[45] S Huang. Gene expression profiling, genetic networks, and cellular states: an integrating concept for tumorigenesis and drug discovery. In Journal of Molecular Medicine, 1999.

[46] Henrik Hulgaard, Poul Frederick Williams, and Henrik Reif Andersen. Equivalence Checking of Combinational Circuits using Boolean Expression Diagrams. In IEEE Transactions on CAD, volume 18, July 1999.

[47] Stanley L. Hurst, Jon C. Muzio, and D. Michael Miller. Spectral Techniques in Digital Logic. Academic Press, Inc., Orlando, FL, USA, 1985.

[48] Ivan Ivanov and Edward R. Dougherty. Modeling genetic regulatory networks:: Continuous or discrete? Journal of Biological Systems, 14(2):p219 – 229, 2006.

[49] Atsushi Iwata, Mieko Maruyama, Ichiro Kanazawa, and Nobuyuki Nukina. alpha -Synuclein Affects the MAPK Pathway and Accelerates Cell Death. J. Biol. Chem., 276(48):45320–45329, 2001.

[50] Cheoljoo Jeong and S.M. Nowick. Technology mapping and cell merger for asynchronous threshold networks. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 27(4):659 –672, apr. 2008.

[51] Cheoljoo Jeong and Steven M. Nowick. Optimization of robust asynchronous circuits by local input completeness relaxation. In ASP-DAC '07: Proceedings of the 2007 Asia and South Pacific Design Automation Conference, pages 622–627, Washington, DC, USA, 2007. IEEE Computer Society.

[52] H. Jeong, B. Tombor, R. Albert, Z. N. Oltvai, and A. L. Barabasi. The large-scale organization of metabolic networks. Nature, 407(6804): 651–654, Oct 5 2000.

[53] H Kalthoff, C Roeder, J Gieseking, I Humburg, and W Schmiegel. Inverse regulation of human ERBB2 and epidermal growth factor receptors by tumor necrosis factor alpha. Proceedings of the National Academy of Sciences of the United States of America, 90(19):8972–8976, 1993.

[54] Fumiaki Katagiri. Attacking Complex Problems with the Power of Systems Biology. Plant Physiol., 132(2):417–419, 2003.

[55] Stuart Kauffman. Homeostasis and differentiation in random genetic control networks. Nature, 224(5215):177–178, 10/11 1969.

[56] S. A. Kaufmann. Metabolic stability and epigenesis in randomly constructed genetic nets. In J. Theor. Biol., 1969.

[57] Jon Kleinberg and Eva Tardos. Algorithm Design. Addison-Wesley, 2005.

[58] Zvi Kohavi. Switching and Finite Automata Theory. New York: McGraw-Hill Book Company, 1970.

[59] Andreas Kuehlmann and Florian Krohm. Equivalence Checking Using Cuts and Heaps. In Proceedings of the 34th annual conference on Design automation, pages 263 – 268, 1997.

[60] Andreas Kuehlmann and Cornelis A.J. van Eijk. Logic Synthesis and Verification, chapter Combinational and Sequential Equivalence Checking. Kluwer Academic Publishers, 2001.

[61] W. Kunz and D.K. Pradhan. Recursive Learning: A New Implication Technique for Efficient Solution to CAD Problems— Test, Verification and Optimization. IEEE Transactions on Computer-Aided Design, pages 1143–1158, September 1994.

[62] Melissa D Landis, Darcie D Seachrist, Marjorie E Montanez-Wiscovich, David Danielpour, and Ruth A Keri. Gene expression profiling of cancer progression reveals intrinsic regulation of transforming growth factor-[beta] signaling in erbb2//neu-induced tumors from transgenic mice. Oncogene, 24(33):5173–5190, 05 2005/05/09/online.

[63] I. Caroline Le Poole, Adam I. Riker, M. Eugenia Quevedo, Lawrence S. Stennett, Ena Wang, Francesco M. Marincola, W. Martin Kast, June K. Robinson, and Brian J. Nickoloff. Interferon-gamma Reduces Melanosomal Antigen Expression and Recognition of Melanoma Cells by Cytotoxic T Cells. Am J Pathol, 160(2):521–528, 2002.

[64] K. K. Likharev. Single-electron devices and their applications. Proceedings of the IEEE, 87(4):606–632, 1999.

[65] Jorn Lind-Nielsen. BuDDy - A Binary Decision Diagram Package, Version 2.2.

[66] Marie Lipoldova and Vladimir Holan. Interleukin-2 activates the [gamma]-interferon gene in newborn mice. Immunol Cell Biol, 69(6): 423–426, 12 1991/12//print.

[67] A Martizez Arias, NE Baker, and PW Ingham. Role of segment polarity genes in the definition and maintenance of cell states in the Drosophila embryo. Development, 103(1):157–170, 1988.

[68] Harley H. McAdams and Adam Arkin. Gene regulation: Towards a circuit engineering discipline. Current Biology, 10(8):R318 – R320, 2000.

[69] Giovanni De Micheli. Synthesis and Optimization of Digital Circuits. McGraw-Hill Higher Education, 1994.

[70] Ethan Mollick. Establishing moore's law. IEEE Annals of the History of Computing, 28(3):62–75, 2006.

[71] G. E. Moore. Cramming more components onto integrated circuits. Proceedings of the IEEE, 86(1):82–85, 1998.

[72] André Auto Moreira and Luís A. Nunes Amaral. Canalizing kauffman networks: Nonergodicity and its effect on their critical behavior. Phys. Rev. Lett., 94(21):218702, Jun 2005.

[73] Saburo Muroga. Threshold Logic and Its Applications. New York: WILEY-INTERSCIENCE, 1971.

[74] New York Academy of Sciences. Dream2 network inference challenge). http://wiki.c2b2.columbia.edu/dream/index.php/DREAM2_ Challenges, Online.

[75] Jan M. Rabaey. Digital Integrated Circuits: A Design Perspective. Prentice Hall, 1995.

[76] John Reinitz and David H. Sharp. Mechanism of eve stripe formation. In Mechanisms of Development, number 49, pages 133–158, 1995.

[77] David B. Roberts. Drosophila melanogaster: the model organism. Entomologia Experimentalis et Applicata, 121:93–103(11), 2006.

[78] J. Rogers, M. C. Mahaney, W. G. Beamer, L. R. Donahue, and C. J. Rosen. Beyond one gene–one disease: Alternative strategies for deciphering genetic determinants of osteoporosis. Calcified Tissue International, 60(3):225–228, 03 1997/03/24/.

[79] Stuart Russell and Peter Norvig. Artificial Intelligence: A Modern Approach. Prentice-Hall, Englewood Cliffs, NJ, 2003.

[80] Lisa L. Salazar Murphy and Christopher C. W. Hughes. Endothelial Cells Stimulate T Cell NFAT Nuclear Translocation in the Presence of Cyclosporin A: Involvement of the wnt/Glycogen Synthase Kinase-3beta Pathway. J Immunol, 169(7):3717–3725, 2002.

[81] L. Sanchez, J. van Helden, and D. Thieffry. Establishment of the dorso-ventral pattern during embryonic development ofdrosophila melanogaster: a logical analysis. Journal of Theoretical Biology, 189(4):377–389, 1997.

[82] Mark Schena, Dari Shalon, Ronald W. Davis, and Patrick O. Brown. Quantitative Monitoring of Gene Expression Patterns with a Complementary DNA Microarray. Science, 270(5235):467–470, 1995.

[83] Uwe Scherf, Douglas T. Ross, Mark Waltham, Lawrence H. Smith, Jae K. Lee, Lorraine Tanabe, Kurt W. Kohn, William C. Reinhold, Timothy G. Myers, Darren T. Andrews, Dominic A. Scudiero, Michael B. Eisen, Edward A. Sausville, Yves Pommier, David Botstein, Patrick O. Brown, and John N. Weinstein. A gene expression database for the molecular pharmacology of cancer. Nature Genetics, 24(3):p236 –, 2000.

[84] I. Shmulevich, E. R. Dougherty, S. Kim, and W. Zhang. Probabilistic boolean networks: a rule-based uncertainty model for gene regulatory networks. Bioinformatics (Oxford, England), 18(2):261–274, 2002.

[85] Ilya Shmulevich, Harri L, Edward R. Dougherty, Jaakko Astola, and Wei Zhang. The role of certain Post classes in Boolean network models of genetic networks. Proceedings of the National Academy of Sciences of the United States of America, 100(19):10734–10739, 2003.

[86] Sandeep K. Shukla and Iris R. Bahar. Nano, Quantum and Molecular Computing. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[87] Michael Sipser. Introduction to the Theory of Computation. Course Technology, December 1996.

[88] J.L. Subirats, J.M. Jerez, and L. Franco. A new decomposition algorithm for threshold synthesis and generalization of boolean functions. Circuits and Systems I: Regular Papers, IEEE Transactions on, 55(10):3188–3196, Nov. 2008.

[89] J. D. Watson, N. H. Hopkins, J. W. Roberts, A. M. Wiener, and . Anonymous. Molecular Biology of the Gene. The Benjamin/ Cummings Publishing Company, 1987.

[90] James D. Watson. The Double Helix: A Personal Account of the Discovery of the Structure of DNA. Touchstone, June 2001.

[91] Neil Weste and David Harris. CMOS VLSI Design: A Circuits and Systems Perspective (3rd Edition). Addison Wesley, 3 edition, May 2004.

[92] B. Widrow and M. Hoff. Adaptive switching circuits. In Western Electronic Show and Convention, 1960 1960.

[93] Poul Frederick Williams, Henrik Hulgaard, and Henrik Reif Andersen. Boolean Expression Diagram Tool – Version 2.5.

[94] L. Wolpert, , R. Beddington, , T. Jessell, P. Lawrence, E. Meyerowitz, and J. Smith. Principles of Development. Oxford University Press, 2002.

[95] Bai Zhang, Huai Li, Rebecca B. Riggins, Ming Zhan, Jianhua Xuan, Zhen Zhang, Eric P. Hoffman, Robert Clarke, and Yue Wang. Differential dependency network analysis to identify condition-specific topological changes in biological networks. Bioinformatics, 25(4):526–532, 2009.

[96] Li Zhang. Threshold Logic Network Synthesis Suite. Master's thesis, Delft University of Technology, 2005.

[97] Rui Zhang, Pallav Gupta, and Niraj K. Jha. Synthesis of Majority and Minority Networks and Its Applications to QCA, TPL and SET Based Nanotechnologies. In VLSID'05, pages 229–234, 2005.

[98] Rui Zhang, Pallav Gupta, Lin Zhong, and Niraj K. Jha. Threshold Network Synthesis and Optimization and Its Application to Nanotechnologies. In IEEE Transactions on CAD, 2005.

[99] Xin Zhang, Chitta Baral, and Seungchan Kim. An algorithm to learn causal relations between genes from steady state data: Simulation and its application to melanoma dataset. Artificial Intelligence in Medicine, pages 524–534, 2005.