A prolog based silicon compiler for topological layouts of switching-graphs.

Arindam Das
University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation
https://scholar.uwindsor.ca/etd/715

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.
NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.
A Prolog based silicon compiler for topological layouts of Switching-graphs

by
Arindam Das

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science in Partial
Fulfillment of the Requirements for the Degree of
Master of Science at the
University of Windsor
Windsor, Ontario, Canada
1993
The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Abstract

Traditional logic minimization techniques have attempted to minimize a circuit in terms of logic gates. For example, PLAs require minimization of a function in terms of AND columns and OR columns. Synthesis with standard cell also involves minimization in terms of standard gates such as NAND, NOR, inverters etc. However, such minimization at gate level (or even at the transistor level) do not necessarily lead to the most efficient VLSI realizations since the logic gate model does not reflect the restrictions of actual placement and routing in a VLSI circuit. A graph-theoretic model has already been proposed by the VLSI research group of University of Windsor, which is isomorphic to a realizable VLSI structure. It has been recognized that reducing the transistor count does not necessarily minimize the layout area and that the cost of routing wires is also vitally important. Therefore, taking into account the routing complexity in the VLSI array, the goal has been to manipulate the graph-based model to achieve area-optimized, multi-level CMOS VLSI circuits. In this thesis, we have implemented the graph theoretic model in the paradigm of logic programming and developed a topological layout generator based on that model and thereby demonstrated the feasibility of its application to the realistic ASIC design problems.
To Baba O Ma
Acknowledgments

The two years I have spent in the VLSI research group of University of Windsor had been the most exciting for me. I am indebted to my research advisor, Dr. Subir Bandyopadhyay for creating a challenging environment and for being a constant source of inspiration and encouragement. I am thankful to Dr. Joan Morrissey for introducing me to the world of logic programming and helping me to restructure my syllogistic thoughts.

In his busiest schedules, it was Dr. Graham A. Jullien, who helped me with stimulating ideas that led to the work in this thesis. I am also thankful to Professor Neil M. Wigley for his academic support during the last stage of my thesis work.

Discussions with Dr. Richard A. Frost had always been stimulating for me. I learned from him, how to extrapolate ideas from one field of science to the other. As a student in the School of Computer Science, I was fortunate to have Stephen Karamatos, Walid Mnaymneh, Margaret Hazael and Mary Mardegan. Finally, I am thankful to Arunita Jaekel for being a great didi, working closely with whom was always enjoyable.

I would like to thank Arathi Ranganathan, Dr. Dimitris Phoukas, Farook Wadia, George Mak, Jianan Wang, Jingming Yan, Rajesh Shenoy, Viren Parasram, William Bealor, Shakil Siddiq, Professor Zongde Wang and Shashank Agarwal for their friendship and support.

I am thankful to Tarun Mal who helped me to get out of stressful conditions at times and had always been a great friend. Tarun da's tea-parties in the purple paradise will always remain unforgettable.

Finally, I am extremely grateful to my parents and my sisters who always believed in my abilities and always prayed to the God for my well-being.
# TABLE OF CONTENTS

Abstract .......................................................... v

Acknowledgments ................................................... vii

1 INTRODUCTION ................................................. 1
  1.1 Motivation for the work ....................................... 1
  1.2 Overview of the work ......................................... 3
    1.2.1 Specific tasks undertaken ................................. 3
      1.2.1.1 Survey of the related work ............................ 4
      1.2.1.2 Development of a switching-graph synthesizer ....... 4
      1.2.1.3 Development of a topological layout generator .... 4
  1.3 Thesis organization ......................................... 5

2 BACKGROUND .................................................. 8
  2.1 An Introduction to the Logic Programming Language PROLOG 8
    2.1.1 Introduction .............................................. 8
    2.1.2 Characteristics of Prolog Programs ..................... 8
    2.1.3 Data Objects ............................................ 9
  2.2 Silicon Compilation ......................................... 9
    2.2.1 Introduction ............................................. 9
    2.2.2 Behavioral Representation ............................... 10
2.2.3 Structural Representation .................................. 11
2.2.4 Geometric Representation ................................. 12
2.3 The Switching-graph Approach ................................ 14
  2.3.1 Review of related work .................................. 14
  2.3.2 Dynamic CMOS Logic ................................. 15
2.4 Summary .......................................................... 16
3 THE SWITCHING-GRAPH SYNTHESIZER ......................... 17
  3.1 Introduction .................................................. 17
  3.2 Synthesis of Switching-graphs .............................. 20
  3.3 Models for the Switching-graph ......................... 28
    3.3.1 Connectivity Model .................................. 28
    3.3.2 Contiguity Model ................................ 31
  3.4 VLSI Realization of Switching-graphs ................. 35
    3.4.1 Local and Nonlocal Merge ......................... 37
    3.4.2 Sneak Path ........................................ 40
  3.5 A Heuristic Procedure for synthesizing Switching-graphs ... 42
    3.5.1 The top level algorithm .......................... 42
  3.6 Summary .................................................. 45
4 THE TOPOLOGICAL LAYOUT GENERATOR ......................... 47
  4.1 Introduction ................................................ 47
5 USE OF LOGIC PROGRAMMING IN THE SYNTHESIS OF
SWITCHING-GRAphS AND TOPOLOGICAL LAYOUT ...... 56
5.1 Introduction ........................................ 56
5.2 The Database Schema ............................... 57
  5.2.1 Switching-graph database ....................... 57
  5.2.2 Topological layout matrix database ............. 59
5.3 Summary ............................................. 62

6 CONCLUSIONS AND FUTURE WORK .................... 63
6.1 Main Features ...................................... 63
6.2 Conclusions ........................................ 64
6.3 Future Work ....................................... 65
BIBLIOGRAPHY ........................................ 66

A SAMPLE RESULTS .................................... 69

B ABSTRACT ALGEBRAIC INTERPRETATION OF THE
FLIPPABLE-NONFLIPPABLE STRUCTURE .................. 72
B.1 Introduction ...................................... 72
B.2 Concept of a Poset ................................ 72
B.3 Concept of a Permutation Group .................. 74
Chapter 1 INTRODUCTION

1.1 Motivation for the work

In this thesis we are reporting our investigations on prototyping layout generators. In this study we have looked at two problems — the first is to investigate the suitability of logic programming for implementing topological layout generators; the second is to automatically generate area efficient layouts for high speed VLSI circuits for a new class of building blocks for digital circuits which we call switching-graphs.

During the early years of digital circuit design, the main problem was to minimize the number of relay circuits, valves or transistors since the cost of a circuit was entirely determined by the number of such elements used in the circuit. In small scale integrated circuits (SSI) or medium scale integrated (MSI) circuit, the same trend continued since the silicon area needed for interconnections was small and hence negligible in terms of the area occupied by the active elements. Extensive investigations over the last 40 years have led to well established and widely used techniques for minimizing a switching circuit with respect to the number of active elements. The best known method is to obtain a minimized sum of products form for any given function and to implement the function using a PLA or NAND gates [5]. Other techniques include multilevel logic synthesis and synthesis techniques for other building blocks (e.g., threshold gates
[23]). These techniques may be all categorized as minimization techniques with respect to a building block (e.g., NAND gates) and results in a network of such building blocks. The objective of these minimization procedures is to minimize the number of building blocks and to reduce the complexity of these building blocks (e.g., reduce the number of inputs to NAND gates). To design an actual chip to implement a given Boolean function, we have a two step procedure. The first step is to synthesize a minimized network of building blocks and the second step is to map this network into silicon using as little silicon area as possible. It is important to note that these two steps are isolated from each other. The silicon area needed in step two is determined by two factors — the complexity of interconnections between the building blocks and the number of building blocks used in the network. The complexity of interconnections is completely ignored in step one. As we moved to LSI and VLSI (very large scale integrated circuits), the complexity of interconnection has become as important as the area occupied by transistors in determining the area of a layout.

In 1992, Jullien [20] introduced a new building block for realizing switching functions called switching-graph. And in 1993, Jaeckel, Bandyopadhyay and Jullien [18] developed a new layout driven, multi-level, non-series-parallel logic design approach to realize a switching-graph from a single-output or multi-output switching function specified using a truth-table. Finally, Jullien [21] developed a topological layout matrix for describing the symbolic layout of the switching-
graph in a technology independent fashion. Logic synthesis using switching-graphs is different from conventional approaches since it attempts to minimize using a model for the network which may be implemented directly in silicon. The synthesis procedure for switching-graphs always keeps track of the area needed to implement the model on silicon. This involves considerable increase in the amount of information that we need to consider and it involves substantially increased computation, relative to standard minimization techniques. Given the throughput rates of present day workstations, additional computation needs is not a bottleneck. Our expectation is that, since we are minimizing the layout area directly, our synthesis procedure will give more compact circuits. In this thesis, we report our attempt to develop a prototype layout generator which implements the heuristics reported in [18] and to develop and implement an algorithm for translating an abstract switching-graph representation to a topological layout matrix. We have carried out all the implementations in the logic programming language, Prolog. Our expectation was that simple list processing techniques and built-in backtracking facility in Prolog would be helpful in rapid prototyping of the heuristics for synthesizing switching-graphs and generating the topological layout.

1.2 Overview of the work

1.2.1 Specific tasks undertaken

The following tasks were carried out as part of the thesis work.
1.2.1.1 Survey of the related work

Owing to the inter-disciplinary nature of the thesis it was necessary to undertake a substantial survey of literature on relevant topics in Computer Science and in VLSI Design. In particular, we surveyed the work on application of logic programming language Prolog in development of silicon compilers and graphical layout editors, automated logic synthesis, full-custom and semi-custom layout methodologies, graph-theoretic algorithms for VLSI layout and automated layout generators.

1.2.1.2 Development of a switching-graph synthesizer

We have implemented the abstract heuristics for converting a switching function to a switching-graph. We call the implementation: A Switching-graph Synthesizer. The synthesizer takes input, from the user, in terms of essential minterms and don’t-care minterms. The output of the synthesizer is a switching-graph specified using a hypergraph model [3] and a list of nodes representing all possible relative position of nodes in the VLSI layout. Since logic programming has convenient facilities for list manipulation and for backtracking[32], the choice of Prolog for this phase was very convenient.

1.2.1.3 Development of a topological layout generator

We have developed an algorithm for converting the abstract switching-graph into a topological layout matrix. This matrix is an array of ASCII characters representing transistors and wires connecting them. The symbols are placed in
the matrix in such a way that the relative position of each symbol in the matrix is exactly the same as the relative position of the corresponding transistors or wires in the mask layout. The motivation behind the development of this generator was to obtain a technology independent symbolic layout for switching-graphs which is portable so far as change in fabrication processes and updates in design rules are concerned. Further, the automatic generation of mask layout is still a slower process than the generation of symbolic layout because of larger sizes of the graphical layout geometries. Also each of these graphical objects in the layout consumes much more memory than the corresponding operator, which is nothing but an ASCII character. The advantages of technology-independence, portability with respect to design rules, increase in the speed of layout generation and less need for memory by layout primitives are motivations for this phase of the work. The implementation was carried out in Prolog for the same reasons described in the previous section.

1.3 Thesis organization

In chapter 2 we have reviewed relevant background material for this thesis. We have briefly looked at the logic programming language Prolog, discussed the concept of silicon compilation and the reasons for its growing importance in the domain of VLSI design. We have also briefly looked at dynamic CMOS circuits and the switching-graph approach.
In chapter 3, we have reviewed, in detail, the work done by Jackel, Bandyopadhyay and Jullien [18] on logic synthesis using the switching-graph approach. We have described the structure of a switching graph, discussed the connectivity and the contiguity models for a switching-graph, described the VLSI realization of switching-graphs, and finally presented the heuristic synthesis technique that we have implemented.

In chapter 4, we have presented the algorithm for translating a switching-graph to a topological layout.

In chapter 5, we have discussed the use of logic programming paradigm in the synthesis of switching-graphs and topological layout. We have discussed the database schema for our database corresponding to the switching-graph and for storing the relevant information about the operators in the topological layout matrix.

In chapter 6, we have made concluding comments with suggestions for possible future work.

In Appendix-A we have presented some of the results related to the switching-graph synthesizer and the topological layout generator, in Appendix-B we have attempted an interpretation of the contiguity model for the switching-graph using group theory, in Appendix-C we have provided the Prolog source code for the switching-graph synthesizer and in Appendix-D we have provided the Prolog
source code for the topological layout generator.
Chapter 2  BACKGROUND

2.1 An Introduction to the Logic Programming Language PROLOG

2.1.1 Introduction

This section introduces the logic programming language, Prolog. It is a simple, yet powerful declarative symbolic language based on predicate logic. Its application domain includes expert systems, natural language processing, symbolic algebra, VLSI Design Automation, relational databases and more recently, image processing [30]. Prolog is mostly used in the early stage of system development for quick implementation. A good account of this language is given in [10].

2.1.2 Characteristics of Prolog Programs

A Prolog program is a set of clauses (logical sentences) written in a subset of first-order logic called Horn clause logic which contains at most one conclusion but one or more conditions [24]. Every condition or conclusion is expressed using a relation containing a relation name and arguments. In Prolog, a relation name is represented by a predicate. Predicates are often referred to by the pair name/arity [37].

The Prolog execution works as a simple theorem prover. Given a set of clauses and a query, Prolog attempts to construct values for the variables in the query that makes the query true. The execution works using the built-in depth-first-search
strategy and automatic backtracking [11]. Because of the depth-first-search, the clauses in the program are tried in the order they are listed and the predicates inside each clause (called subgoals) are invoked from left to right in the physical order they appear in the clause. This strict order imposed on the execution makes Prolog rather weak as a theorem prover, but useful as a programming language, especially since it can be implemented efficiently, much more so than a more general theorem prover [34].

2.1.3 Data Objects

The data objects and their manipulation are modelled after first order logic. Simple objects in Prolog are atoms, variables and numbers [12]. There is another type of data object called structure or compound term [37]. A variable instantiation is a single-assignment, i.e. a variable may be instantiated once only. Variables can be arguments of predicates or they can be the arguments of compound structures. The lexical scope of variables is one clause only which implies that the same variable name in two clauses means two different variables.

2.2 Silicon Compilation

2.2.1 Introduction

A VLSI circuit may be described at many levels of detail — ranging all the way from description of polygons constituting the chip to the input/output characteristics of the circuit. High level description for VLSI gives the behavior
of the circuit and does not include any details about the physical devices in the circuit. The transformation of such a high level description into a physical layout is a very difficult problem [1]. One solution to this problem is the concept of silicon compilation introduced by Dave Johannsen at Caltech [19]. In a broader sense, we can define silicon compilation as a translation process from a higher-level description into a layout level description. This translation process can be broken down into several steps, and each step can be considered to be a compiler for the lower level. The term silicon compilation has also been used in a variety of different contexts [15].

The Y-chart of Gajski [15] (figure 2.1) presents different aspects of silicon compilation. Multiple levels of abstraction are represented along each of the three axes. They represent three different domains of description: behavioral, structural and geometrical. The levels of abstraction increase as one moves away from the vertex.

2.2.2 Behavioral Representation

The behavioral representation of a circuit specifies the input output behavior of the circuit and not how the circuit is actually implemented. The design in this domain consists of a specific set of inputs, a set of outputs, and a set of functions describing the behavior of each output as a function of the inputs [15].
2.2.3 Structural Representation

A structural representation is a one-to-many mapping of a behavioral representation onto a set of components and connections under constraints such as cost, area and time [15]. The translation of Boolean expression(s) (behavioral representation) to the corresponding structural representation is termed as logic synthesis. The structural representation, in essence, is a netlist specifying the connectivity of components and the component interfaces (ports). The structural representation does not specify any physical parameters such as the coordinates
of the components.

In silicon compilation, the stage of logic synthesis is of importance because it is the first step towards obtaining an area minimized, technology-independent description of a Boolean network [5]. Logic synthesis in terms of two-level or multilevel minimization of AND-OR gates is a well known example of this process [5], [38].

### 2.2.4 Geometric Representation

The structure-to-geometry mapping is a two-step process where the first step, (usually called the *topological layout*), determines relative (or, in some systems, approximate) positions for all the structural elements and the second step (*mask layout*) determines the absolute positions of the elements.

Topological layout is an important part of the silicon compilation process since the need for more effective data description techniques is becoming critical with the growing complexity of VLSI circuits [22]. Design verification from mask artwork data alone proves to be computationally intractable. The use of a topological design description, which allows the designer or synthesis program to express circuit structure as well as maintain full connectivity information, can reduce dramatically the burden placed on verification tools [22]. The most significant advantage of a topological layout is that it is technology-independent. Therefore, when design rules change, more compact mask layout can be generated.
using any previously existing topological description, automatically, after the
design specification file has been updated. A topological design system, classical
[25] [16] or non-classical [27], permits the circuit structure to be captured in the
design and this information can be used by verification tools to speed up their
analysis [22].

As mentioned in the previous section, the mask layout is the second step in
the structure-to-geometry mapping. After the relative positions of the structural
elements have been determined in the topological layout step, the next stage is the
generation of the mask layout. The mask layout generator takes the topological
layout as input and generates the absolute coordinates for the mask rectangles
realizing the devices and the interconnects. This generation is carried out using
different placement and routing heuristics. The placement heuristics are used for
positioning the devices and routing heuristics are used for routing the interconnects
in order to realize the desired connections between the devices.

Unlike the topological layout, this phase of geometric representation is tech-
nology dependent [27]. The cost and the area for the mask layout of any circuit
is dependent on the technology being used to realize the circuit. A mask layout
generator always takes a technology file for a specific technology as its auxiliary
input. This technology file is a database containing information on the minimum
allowed distance of separation between the different mask rectangles, minimum
overlapping distance allowed between them, names of the layers which realize
the mask rectangles and various technology-related rules that govern the error-free mask artwork generation.

2.3 The Switching-graph Approach

2.3.1 Review of related work

Since Shannon [35] developed, in 1938, the theory of networks of relays, switching theory for digital systems has developed along two branches [28]. One of them is called switching circuit theory using Boolean gates and the other is called switching network theory using switches. Since early logic circuits were composed of bipolar transistors which could not act effectively as switches, switching circuit theory using gates was intensively investigated for designing digital systems. At the same time, switching network theory using switches was not extensively used in electronic logic circuits, although it had been well established and used in relay circuits [17] [8]. With the advent of metal-oxide-semiconductor field effect transistor (MOSFET), the switch concept was accepted in the area of electronic logic design because MOSFETs are similar to bilateral switches. The designers of MOS circuits use the switch concept with only certain regular structures such as symmetric functions* since there is no available theory for minimizing MOS circuits [28].

* An n-variable Boolean function \( f \) is a symmetric function if \( f(x_1, x_2, ..., x_n) = f(x_1, x_2, ..., x_{j_1}, ..., x_{j_m}) \) for example, \( f(x, y) = x'y + xy' \) is a symmetric Boolean function.
Based on Shannon's work, in 1958 Lee [26] introduced a graph-based notation called binary-decision diagram to represent Boolean functions. This work was further popularized by Bryant [6]. In 1992, Jullien [20] introduced a building block for realization of a Boolean function called Switching-graph. This building block has been designed for bit-level pipelined computations to perform real-time digital signal processing for high bandwidth data stream applications [21] [4]. Later, in section 2.3 we have discussed, in detail, the behavioral semantics of this new building block. In order to realize the switching-graph, we have selected a design style. In the next section, we have reviewed the dynamic CMOS logic which is useful in synthesizing switching-graphs.

2.3.2 Dynamic CMOS Logic

An important property of a dynamic CMOS gate is its ability to store charge in the parasitic capacitances for relatively large periods of time. Dynamic circuits are faster and require less area than their static CMOS counterparts [36]. In a dynamic CMOS circuit the pmos network in the corresponding static CMOS circuit is replaced by a single pmos transistor for precharging. This technique retains the low power consumption of static logic, but requires a clock signal to drive the precharging transistor. The advantages of this type of logic family are: smaller area, reduced load capacitance (only one pmos switch), higher speeds and easy pipelining [20]. It is well known that dynamic logic is not viable in bipolar
Junction Transistor (BJT) since BJT's have a large leakage current.

2.4 Summary

In this chapter we have discussed the relevant background material for our thesis work. We have briefly looked at the logic programming language Prolog, discussed the concept of silicon compilation and the reasons for its growing importance in the domain of VLSI design. Finally, we have briefly looked at dynamic CMOS circuits and the switching-graph approach. In the next chapter we discuss the switching-graph synthesizer.
Chapter 3  THE SWITCHING-GRAH SYNTHESIZER

3.1 Introduction

In this chapter we discuss the logic synthesizer for building single or multi-output combinational logic blocks for pipelined dynamic logic. We look at this problem as a multilevel, non-series-parallel pass-transistor logic network synthesis unlike the traditional approaches such as NAND-NOR logic, PLA [29] or multilevel AND/OR network synthesis [2].

In this chapter we informally discuss what a switching-graph is, and present the synthesis technique that we have used.

The combinational circuit shown inside the dotted rectangle in figure 3.1 represents what we call a switching-graph. In the figure we have shown a single output function. This circuit has one pmos transistor (p1) and an nmos transistor (n1) driven by clock clk, and a network of nmos transistors (inside the dotted rectangle). A multi-output switching-graph will have one pmos transistor for every output. The network of nmos transistors inside the dotted rectangle, determines the Boolean function realized by the circuit. The intuitive explanation of the operation of an nmos transistor is that of an ideal switch with only two states, on or off, depending on whether the input to the transistor is logic level 1 or 0. When the input to a nmos transistor is 1 (0), the switch is closed (open).
Switching-graphs are based on dynamic logic so that there is a precharge phase followed by an evaluation phase [36].

In the circuit shown in 3.1, during the precharge (clk=0) phase, the input to the pmos transistor p1 is logic 1 so that the transistor is turned on and the output node (O) becomes logic 1. During the evaluation phase (clk=1), this logic level
may or may not be maintained depending on the inputs applied to the network of nmos transistors and the configuration of the network itself. If, in this network of nmos transistors, there exists any path from the output node O to the ground such that all transistors in the path has input 1, the charge stored at the output node is discharged and the output node goes to logic 0. If, however, there is no such path from O to ground, the charge stored at output O remains unaltered and the output continues to be at logic 1. It is easy to verify that, for the circuit shown in Figure 3.1 when \( x_1 = 1, x_2 = 1, x_3 = 1, x_4 = 0 \) and \( x_6 = 1 \), there is a path from \( O \) to ground. Therefore, for this input combination, during evaluation phase, the charge at \( O \) will discharge and the output is 0. Similarly, when \( x_1 = 0 \) and \( x_2 = 1 \), there is no path from \( O \) to ground so that for this combination, the output is 1.

Synthesis of a switching-graph to realize a Boolean function specified in terms of sum of minterms, is the process of designing the network of nmos transistors so that there is a path from the evaluation node to the ground for all of the essential minterms of the function and there is no path from the evaluation node to the ground for all those minterms which are neither essential nor don’t-care. Area efficient synthesis of a switching-graph is a synthesis in such a way that the network of nmos transistors may be mapped into silicon using a minimum layout area. This is the topic we have investigated in this dissertation. We now describe the approach we have used for synthesis and two models we need for the switching-graphs.
3.2 Synthesis of Switching-graphs

Any n-variable switching function of n variables \(x_n, \ldots, x_1\) can be completely specified by a set of essential minterms called on-set and a set of don't care minterms called don't-care-set. We shall denote the on-set by \(e(x_n, \ldots, x_1)\), the don't-care-set by \(d(x_n, \ldots, x_1)\) and the switching function by \(e(x_n, \ldots, x_1) / d(x_n, \ldots, x_1)\).

Based on Shannon's expansion theorem [23], the n-variable switching function can be expressed as

\[
e(x_n, \ldots, x_1) / d(x_n, \ldots, x_1) = x_n e_1(x_{n-1}, \ldots, x_1) / d_1(x_{n-1}, \ldots, x_1) + \overline{x_n} e_0(x_{n-1}, \ldots, x_1) / d_0(x_{n-1}, \ldots, x_1)
\]

We call this as the form-I expansion where

\[
e_1(x_{n-1}, \ldots, x_1) = \{m_j - 2^n \mid m_j \in e(x_n, \ldots, x_1) \text{ and } m_j \geq 2^n\}
\]

\[
d_1(x_{n-1}, \ldots, x_1) = \{m_j - 2^n \mid m_j \in d(x_n, \ldots, x_1) \text{ and } m_j \geq 2^n\}
\]

\[
e_0(x_{n-1}, \ldots, x_1) = \{m_j \mid m_j \in e(x_n, \ldots, x_1) \text{ and } m_j < 2^n\}
\]

\[
d_0(x_{n-1}, \ldots, x_1) = \{m_j \mid m_j \in d(x_n, \ldots, x_1) \text{ and } m_j < 2^n\}
\]
We call \( e_1(x_{n-1}, ..., x_1) / d_1(x_{n-1}, ..., x_1) \) and \( e_0(x_{n-1}, ..., x_1) / d_0(x_{n-1}, ..., x_1) \) as reduced functions [23].

**Definition:**

Two switching functions \( e_1(x_n, ..., x_1) / d_1(x_n, ..., x_1) \) and \( e_2(x_n, ..., x_1) / d_2(x_n, ..., x_1) \) are said to be compatible if \( e_1(x_n, ..., x_1) \subseteq \{ e_2(x_n, ..., x_1) \cup d_2(x_n, ..., x_1) \} \) and \( e_2(x_n, ..., x_1) \subseteq \{ e_1(x_n, ..., x_1) \cup d_1(x_n, ..., x_1) \} \). The property of this two switching functions being compatible is termed as *compatibility*.

Next we introduce three rules where we present three expansions similar to Shannon's expansion. Our implementation of the switching-graph synthesizer which we have discussed in section 3.5, is primarily based on these rules.

**Rule 1:** If the reduced functions \( e_1(x_{n-1}, ..., x_1) / d_1(x_{n-1}, ..., x_1) \) and \( e_0(x_{n-1}, ..., x_1) / d_0(x_{n-1}, ..., x_1) \) in the form-I expansion of the switching function \( e(x_n, ..., x_1)/d(x_n, ..., x_1) \) are compatible, then \( e(x_n, ..., x_1)/d(x_n, ..., x_1) \) can be expanded as

\[
(x_n + \overline{x_n}).e^{new}(x_{n-1}, ..., x_1)/d^{new}(x_{n-1}, ..., x_1)
\]

\[
= 1.e^{new}(x_{n-1}, ..., x_1)/d^{new}(x_{n-1}, ..., x_1)
\]

\[
= e^{new}(x_{n-1}, ..., x_1)/d^{new}(x_{n-1}, ..., x_1)
\]

Thus \( e(x_n, ..., x_1)/d(x_n, ..., x_1) \) is compatible with the function

\[
e^{new}(x_{n-1}, ..., x_1)/d^{new}(x_{n-1}, ..., x_1)
\]
where

\[ e_{\text{new}}^{n}(x_{n-1}, \ldots, x_{1}) = [c_{1}(x_{n-1}, \ldots, x_{1}) \cup c_{0}(x_{n-1}, \ldots, x_{1})] \]

and

\[ d_{\text{new}}^{n}(x_{n-1}, \ldots, x_{1}) = [d_{1}(x_{n-1}, \ldots, x_{1}) \cap d_{0}(x_{n-1}, \ldots, x_{1})] \]

When the conditions for rule 1 is satisfied, we will call the expansion of \( e(x_{n}, \ldots, x_{1}) / d(x_{n}, \ldots, x_{1}) \) to the form \( e_{\text{new}}^{n}(x_{n}, \ldots, x_{1}) / d_{\text{new}}^{n}(x_{n}, \ldots, x_{1}) \) as \textbf{form-II} expansion.

We find that the property of compatibility is important because if any two switching functions are compatible, they can be combined to form an expression of type form-II which realizes the functionality of both the switching functions. Also the functionality will change if either any element is excluded from the on-set of that form-II expression or any element other than that belonging to the don't-care-set of the form-II expression is included in its on-set. In synthesizing switching-graphs, we use this notion of compatibility.

\textbf{§ Rule 2:} If the reduced functions \( e_{1}(x_{n-1}, \ldots, x_{1}) / d_{1}(x_{n-1}, \ldots, x_{1}) \) and \( e_{0}(x_{n-1}, \ldots, x_{1}) / d_{0}(x_{n-1}, \ldots, x_{1}) \) in the form-I expansion of the switching function \( e(x_{n}, \ldots, x_{1}) / d(x_{n}, \ldots, x_{1}) \) satisfy the following condition:

\[ e_{1}(x_{n-1}, \ldots, x_{1}) \subseteq [ e_{0}(x_{n-1}, \ldots, x_{1}) \cup d_{0}(x_{n-1}, \ldots, x_{1}) ] \]

* Note that the expansion \( e_{\text{new}}^{n}(x_{n-1}, \ldots, x_{1}) / d_{\text{new}}^{n}(x_{n-1}, \ldots, x_{1}) \), is an n-variable function which is independent of variable \( x_{n} \).
then \( c(x_n, \ldots, x_1)/d(x_n, \ldots, x_1) \) is compatible with function

\[
    c_1(x_{n-1}, \ldots, x_1)/d_1^{new}(x_{n-1}, \ldots, x_1) + \overline{c}_0^{new}(x_{n-1}, \ldots, x_1)/d_0^{new}(x_{n-1}, \ldots, x_1)
\]

where

\[
    d_1^{new}(x_{n-1}, \ldots, x_1) = d_1(x_{n-1}, \ldots, x_1) \cap [c_0(x_{n-1}, \ldots, x_1) \cup d_0(x_{n-1}, \ldots, x_1)]
\]

\[
    c_0^{new}(x_{n-1}, \ldots, x_1) = c_0(x_{n-1}, \ldots, x_1) - c_1(x_{n-1}, \ldots, x_1)
\]

\[
    d_0^{new}(x_{n-1}, \ldots, x_1) = d_0(x_{n-1}, \ldots, x_1) \cup c_1(x_{n-1}, \ldots, x_1)
\]

When the conditions for rule 2 is satisfied, we will call the expansion of

\( e(x_n, \ldots, x_1)/d(x_n, \ldots, x_1) \) to the form

\[
    c_1(x_{n-1}, \ldots, x_1)/d_1^{new}(x_{n-1}, \ldots, x_1) + \overline{c}_0^{new}(x_{n-1}, \ldots, x_1)/d_0^{new}(x_{n-1}, \ldots, x_1)
\]

as form-III expansion.

**Proof:**

Let \( w_n \) be the binary weight of the variable \( x_n \) in form-1 expansion. Let

\[
    E_1 = \{ w_n + \psi_1 \mid \psi_1 \in c_1(x_{n-1}, \ldots, x_1) \} \cup c_0(x_{n-1}, \ldots, x_1)
\]

\[
    D_1 = \{ w_n + \psi_2 \mid \psi_2 \in d_1(x_{n-1}, \ldots, x_1) \} \cup d_0(x_{n-1}, \ldots, x_1)
\]

\[
    \psi_1, \psi_2, \ldots, \psi_{n-1} 
\]

23
\[ E_2 = \{ w_n + \psi_1 \mid \psi_1 \in e_1(x_{n-1}, \ldots, x_1) \} \cup e_1(x_{n-1}, \ldots, x_1) \cup e_0^{ncw}(x_{n-1}, \ldots, x_1) \]

\[ D_2 = \{ w_n + \psi_3 \mid \psi_3 \in d_1^{ncw}(x_{n-1}, \ldots, x_1) \} \cup d_1^{ncw}(x_{n-1}, \ldots, x_1) \cup d_0^{ncw}(x_{n-1}, \ldots, x_1) \]

Since

\[ e_0^{ncw}(x_{n-1}, \ldots, x_1) = e_0(x_{n-1}, \ldots, x_1) - e_1(x_{n-1}, \ldots, x_1) \]

therefore

\[ e_1(x_{n-1}, \ldots, x_1) \cup e_0^{ncw}(x_{n-1}, \ldots, x_1) = e_1(x_{n-1}, \ldots, x_1) \cup e_0(x_{n-1}, \ldots, x_1) \]

Hence,

\[ E_2 \cup D_2 = \{ w_n + \psi_1 \mid \psi_1 \in e_1(x_{n-1}, \ldots, x_1) \} \cup e_1(x_{n-1}, \ldots, x_1) \]

\[ \cup e_0^{ncw}(x_{n-1}, \ldots, x_1) \cup d_1(x_{n-1}, \ldots, x_1) \cup d_0^{ncw}(x_{n-1}, \ldots, x_1) \]

\[ = \{ w_n + \psi_1 \mid \psi_1 \in e_1(x_{n-1}, \ldots, x_1) \} \cup e_1(x_{n-1}, \ldots, x_1) \]

\[ \cup e_0(x_{n-1}, \ldots, x_1) \cup d_1(x_{n-1}, \ldots, x_1) \cup d_0^{ncw}(x_{n-1}, \ldots, x_1) \]

\[ \supseteq E_1 \]

Thus \( E_1 \subseteq E_2 \cup D_2 \)

Also

\[ e_1(x_{n-1}, \ldots, x_1) \cup e_0^{ncw}(x_{n-1}, \ldots, x_1) \]

24
\[ E_1 \cup D_1 \supseteq \{ w_n + \psi_1 \mid \psi_1 \in e_1(x_{n-1}, \ldots, x_1) \} \cup e_0(x_{n-1}, \ldots, x_1) \cup d_0(x_{n-1}, \ldots, x_1) \]
\[ \supseteq \{ w_n + \psi_1 \mid \psi_1 \in e_1(x_{n-1}, \ldots, x_1) \} \cup e_1(x_{n-1}, \ldots, x_1) \cup e_0^{new}(x_{n-1}, \ldots, x_1) \]
\[ = E_2 \]

Thus \( E_2 \subseteq E_1 \cup D_1 \)

Since \( E_1 \subseteq E_2 \cup D_2 \) and \( E_2 \subseteq E_1 \cup D_1 \), the switching function \( e(x_n, \ldots, x_1) \) is compatible with the switching function

\[ e_1(x_{n-1}, \ldots, x_1) / d_1^{new}(x_{n-1}, \ldots, x_1) + \pi_n e_0^{new}(x_{n-1}, \ldots, x_1) / d_0^{new}(x_{n-1}, \ldots, x_1) \]

\[ \square \]

§Rule 3: If the reduced functions \( e_1(x_{n-1}, \ldots, x_1) / d_1(x_{n-1}, \ldots, x_1) \) and \( e_0(x_{n-1}, \ldots, x_1) / d_0(x_{n-1}, \ldots, x_1) \) in the form-I expansion of the switching function \( e(x_n, \ldots, x_1) / d(x_n, \ldots, x_1) \) satisfy the following condition:

\[ e_0(x_{n-1}, \ldots, x_1) \subseteq \left[ e_1(x_{n-1}, \ldots, x_1) \cup d_1(x_{n-1}, \ldots, x_1) \right] \]

then \( e(x_n, \ldots, x_1) / d(x_n, \ldots, x_1) \) is compatible with function

\[ x_n e_1^{new}(x_{n-1}, \ldots, x_1) / d_1^{new}(x_{n-1}, \ldots, x_1) + e_0(x_{n-1}, \ldots, x_1) / d_0^{new}(x_{n-1}, \ldots, x_1) \]

where

\[ d_0^{new}(x_{n-1}, \ldots, x_1) = e_0(x_{n-1}, \ldots, x_1) \cap [e_1(x_{n-1}, \ldots, x_1) \cup d_1(x_{n-1}, \ldots, x_1)] \]
\[ e_1^{new}(x_{n-1}, \ldots, x_1) = e_1(x_{n-1}, \ldots, x_1) - e_0(x_{n-1}, \ldots, x_1) \]

\[ d_1^{new}(x_{n-1}, \ldots, x_1) = d_1(x_{n-1}, \ldots, x_1) \cup c_0(x_{n-1}, \ldots, x_1) \]

When the conditions for rule 3 is satisfied, we will call the expansion of \( e(x_n, \ldots, x_1)/d(x_n, \ldots, x_1) \) to the form

\[ x_n e_1^{new}(x_{n-1}, \ldots, x_1)/d_1^{new}(x_{n-1}, \ldots, x_1) + e_0(x_{n-1}, \ldots, x_1)/d_0^{new}(x_{n-1}, \ldots, x_1) \]

as form-IV expansion.

**Proof:**

The proof is similar to that of rule 2.

It is important to note that in both form-III and form-IV, some new don't-care terms are generated. For example, in form-III the don't-care-set \( d_0^{new}(x_{n-1}, \ldots, x_1) \) includes all the elements of on-set \( e_1(x_{n-1}, \ldots, x_1) \) in addition to the elements of the don't-care-set \( d_0(x_{n-1}, \ldots, x_1) \). We term these additional elements in \( d_0^{new}(x_{n-1}, \ldots, x_1) \) contributed by \( e_1(x_{n-1}, \ldots, x_1) \) as implicit don't-care minterms. Similarly in form-IV, \( d_1^{new}(x_{n-1}, \ldots, x_1) \) includes all the elements of \( e_0(x_{n-1}, \ldots, x_1) \) as implicit don't-care minterms in addition to the elements of \( d_1(x_{n-1}, \ldots, x_1) \).
Since implicit don’t-care minterms are generated in form-III and in form-IV, the possibility of obtaining compatible reduced functions increases when we choose these forms as opposed to form-I.

Example:

Let \( e(x_3, x_2, x_1)/d(x_3, x_2, x_1) = \)

\[
x_3.e_1(x_2, x_1)/d_1(x_2, x_1) + \overline{x_3}.e_0(x_2, x_1)/d_0(x_2, x_1)
\]

be a 4–variable switching function where

\[
e(x_3, x_2, x_1)/d(x_3, x_2, x_1) = \{1, 5, 6\}/\{0, 3, 4\}
\]

\[
e_1(x_2, x_1)/d_1(x_2, x_1) = \{1\}/\{0, 3\}
\]

\[
e_0(x_2, x_1)/d_0(x_2, x_1) = \{1, 2\}/\{0\}
\]

then an alternate form of the function will be

\[
e_1(x_2, x_1)/d_1^{new}(x_2, x_1) = \{1\}/\{0\}
\]

where

\[
e_1(x_2, x_1)/d_1^{new}(x_2, x_1) = \{1\}/\{0\}
\]

and

\[
e_0^{new}(x_2, x_1)/d_0^{new}(x_2, x_1) = \{2\}/\{0, 1\}
\]
In fact, this alternate form represents the form-III expression of the given switching function where the element 1 in the don't-care-set \( u^{nc}(x_2, x_1) = \{0, 1\} \) is the implicit don't-care minterm.

### 3.3 Models for the Switching-graph

In order to generate area efficient VLSI layouts, we adopt two semantic models for representing switching-graphs. The two models are the connectivity model and the contiguity model. The connectivity model is based on the hypergraph representation [3] and indicates which transistors in a switching-graph are connected. The contiguity model is based on predicate logic [24] and indicates the relative position of transistors with respect to each other.

#### 3.3.1 Connectivity Model

We use a hypergraph [3] to represent the connectivity information for a switching-graph. In a hypergraph for a switching-graph having inputs \( x_1, x_2, ..., x_n \), there are two types of nodes: **type-I** (which we represent using circles) and **type-II** (which we represent using squares). Each type-I node has a triplet \( T_1 \) as a label where \( T_1 = (level, id, flag) \), \( 1 \leq level \leq n \), \( id \) is a unique integer for all nodes and \( flag \) is \(-1, 0 \) or \( 1 \). Each type-I node corresponds to a possible position where a transistor may occur in the switching-graph. If a type-I node has \( level \) \( i \) and \( flag = -1 \) \( (1) \) it corresponds to a transistor having input \( \overline{x}_i \) \( (x_i) \).
If a type-I node has level i and flag 0 this indicates the absence of a transistor (i.e., the transistor is replaced by a wire).

Each type-II node has a triplet T2 as a label where T2 = (level, id, etd), 1 ≤ level ≤ n, id is represented by a unique integer with subscript a for all nodes and etd corresponds to the on-set and the don’t-care-set of the reduced function represented by the node.

A node at jth level is connected only to some selected nodes at (j-1)th level and to some selected nodes at (j+1)th level, for all j ≥ 1. During the synthesis of a switching-graph, if we have nodes with level 1, 2, ..., j, then:

i) we have only type-II nodes with level = j

ii) we have only type-I nodes with level = 1, 2, ..., j-2

iii) we have type-I and type-II nodes with level = j-1

iv) a type-II node with level = 1 is not connected to any type-I node.

Example: We shall illustrate the step-by-step process of generating a hypergraph representation of a switching-graph for a 4-variable switching function e(x4, x3, x2, x1)/d(x4, x3, x2, x1) = {0, 1, 3, 6, 9, 14} / {} up to level = 3. Here we don’t describe how we carry out the process of transforming a type-II node into a subgraph of type-I and type-II nodes.

i) Figure 3.2(a) shows a type-II node (with id 1a) having triplet (1, 1a, {0, 1, 3, 6, 9, 14} / {}).
ii) The triplet \( (1, 1_a, [0, 1, 3, 6, 9, 14]) / {} \) is then replaced by two type-I nodes with triplets \((1, 2, -1)\) and \((1, 3, 1)\) at level = 1 and also two type-II nodes are generated with triplets \((2, 2_a, [0, 1, 3, 6]) / {}\) and \((2, 3_a, [1, 6]) / {}\) at level = 2. This is shown in figure 3.2(b).

iii) Next, the type-II node with triplet \((2, 2_a, [0, 1, 3, 6]) / {}\) is replaced by two type-I nodes with triplets \((2, 4, -1)\) and \((2, 5, 1)\). The type-II node with triplet \((2, 3_a, [1, 6]) / {}\) is replaced by two type-I nodes with triplets \((2, 6, \ldots)\).
-1) and (2, 7, 1) at level = 2 and also four type-II nodes are generated with triplets (3, 4a, {0, 1, 3} / {1}), (3, 5a, {2} / {1}), (3, 6a, {1} / {1}) and (3, 7a, {2} / {1}) at level = 3. This is shown in figure 3.2(c).

iv) We find that the reduced functions represented by the two type-II nodes with id 5a and 7a of figure 3.2(c) are compatible. Hence we can combine 5a and 7a to generate the function {2} / {1} and the corresponding type-II node is labelled with the triplet (3, 7a, {2} / {1}). The horizontal dotted line in figure 3.3(a) implies that the two type-II nodes 5a and 7a have been found compatible and therefore have been combined to form the type-II node 7a.

3.3.2 Contiguity Model

The connectivity model described in the previous section does not indicate the physical location of the nodes and cannot be used to determine the relative
positions of nodes with respect to one another. Therefore, in order to maintain the information regarding the positions of the nodes with respect to each other, we use a contiguity model based on predicate logic which we have described using relational structure concept [13].

Let the relational structure for a contiguity model be a quadruple

\[ < E, N, R, H > \]

where:

- \( E \) is the set of node-ids of type-I nodes, \( \{ x \mid x > 1, x \text{ is a natural number} \} \).
- \( N \) is the set of distinguished entities such that \( N \subseteq E \). In the present case, since all the entities in \( E \) are identifiable, \( N \) is equal to \( E \).
- \( R = \{ f, n \} \), where \( f \) and \( n \) are binary relations as described below.
- \( H \) is the set of functions each of which is defined on \( E \). In the present case, it is an empty set.

Let \( x \) and \( y \) be either distinct node-ids in a switching-graph or any relation in \( R \). We call the relation \( f \) a \textit{flip}-relation and the relation \( n \) a \textit{nonflip}-relation. \( f \in R \) defined on \( E \) is a binary relation such that

\[ \forall x \forall y \ f(x, y) \rightarrow f(y, x) \]

and \( n \in R \) defined on \( E \) is a binary relation such that

\[ \forall x \forall y \ n(x, y) \rightarrow \sim n(y, x) \]
We call \( x \) and \( y \) to be \textit{flippable} if the symmetric relationship defined above holds between \( x \) and \( y \). Otherwise, we call \( x \) and \( y \) to be \textit{non-flippable} if the anti-symmetric relationship defined above holds between \( x \) and \( y \). Later in our discussion, we have termed the \( f(x, y) \) and \( n(x, y) \) together as \textit{flippable-nonflippable structure}.

The figure 3.4(b) and figure 3.5(b) shows the contiguity model for the switching-graph realizing the function \( \Sigma \{0, 1, 3, 6, 9, 14\} / \Sigma \{\} \). We have also included the corresponding connectivity models in figure 3.4(a) and figure 3.5(a) respectively. For the benefit of explanations, we have included two extra nodes in each of the connectivity model (i.e., the hypergraph representation). We call these nodes as dummy nodes because they do not have \(-1\) or \(1\) or \(0\) associated with them as the value of \textit{flag} which we have discussed for type-I nodes in the connectivity model. The contiguity model in figure 3.4(b) shows that all the nodes at every level of the switching-graph are flippable. This property is denoted by the presence of the relation \( f \) only at every level of the switching-graph. For example, if we consider the given switching-graph only up to level 2, then we find that nodes 4 and 5 (nodes 6 and 7) are flippable.

The contiguity model in figure 3.5(b) implies that the we have interchanged the positions of nodes 6 and 7 and that the relative positions of these two nodes have been fixed. Informally we say that these nodes have been \textit{flipped}. This flipping has been done because we want to place them next to each other. In the
next section, while describing the VLSI implementation of this switching-graph, we will explain the reason for placing those nodes next to each other. Once the flipping has been done, the nodes 4 and 5, 7 and 6 have become nonflippable. Hence the flip-relation \( f \) is replaced by non-flip relation \( n \) wherever this happens. Thus \( f(4, 5) \) becomes \( n(4, 5) \) and \( f(6, 7) \) becomes \( n(7, 6) \). The relation \( n(f(8, 9), 10) \) at level 3 implies that although 8 and 9 can interchange their positions (i.e. they are flippable), the relative positions of \( f(8, 9) \) and 10 cannot be interchanged.
3.4 VLSI Realization of Switching-graphs

So far we have discussed the connectivity and the contiguity models of a switching-graph. However, our primary objective is to realize an area-efficient VLSI implementation of any arbitrary switching-graph. We now discuss, informally, using an example, how we realize the VLSI layout from the two models of a switching-graph.

We show the connectivity and the contiguity models of the switching-graph realizing the function $\sum\{0, 1, 3, 6, 9, 14\} / \sum\{\}$ (3.4 and 3.5). In these models, we
have generated type-I nodes for all four levels. In order to realize the switching-graph corresponding to the two models, we have to accomplish the following:

- each type-I node is to be replaced by an nmos transistor
- each hyperedge is to be replaced by a net†
- the position of a dummy node in the hypergraph representation is not occupied by an nmos transistor; it is kept vacant

There are however, two major implementation problems which arise during the VLSI implementation of a switching-graph. One of them arises due to the bidirectional property of an nmos transistor which may lead to the creation of undesired paths from evaluation node to the ground. This problem is analogous to the problem of sneak paths [23] in the case of relay based circuits. We will later discuss, in detail, how the sneak paths creep in, how to detect sneak paths and how to avoid them. The second problem is related to the optimization of the VLSI layout area (when we map the switching-graph into silicon). The area of a VLSI layout is determined by the width and the height of the layout. The width of the layout is determined by the maximum number of type-I nodes in a level. Therefore it is vitally important to keep the number of type-I nodes at any given level to a minimum. We have used a greedy heuristic procedure which attempts to reduce the number of type-II nodes at any given level to a minimum

† a net is an equipotential wire-connection
by determining which type-II nodes have compatible switching functions and combining such type-II nodes to the maximum extent possible. We will term this process of combining two type-II nodes as a *merge*. It turns out that there are two types of merges. The first type of merge may increase the height of the layout since it involves additional channel width for routing. In contrast, the second type of merge does not involve any additional channel width. For reasons that will be clear later we term the first (second) type of merge as *nonlocal* (*local*).

### 3.4.1 Local and Nonlocal Merge

In a local (nonlocal) merge, it is (it is not) possible to make the two type-II nodes, that we wish to merge, physically next to one another. In a local merge, the wire link connecting a sub-network of nmos transistors to another sub-network of nmos transistors, does not occupy any extra VLSI layout area. For instance, in figure 3.6(a), the type-II nodes with id = 5<sub>a</sub> and id = 7<sub>a</sub> are compatible and therefore can be merged. It is also clear from the figure that the type-I nodes with id = 6 and id = 7 can be flipped such that the type-II nodes to be merged can be made physically adjacent. In other words, a local merge can be carried out between those type-II nodes. The figure 3.6(b) depicts the local merge between the type-II nodes in question using a dashed line and the change of position of those nodes as a consequence.

Next, we present an example of nonlocal merge using figures 3.7 and 3.8. In figure 3.7, the type-II nodes with id = 9<sub>a</sub> and id = 12<sub>a</sub>, being compatible, are
possible candidates for a merge. However, due to a local merge that has already taken place in level $= 2$, it is not possible to bring the type-II nodes currently being considered for a merge, next to each other. Hence we carry out a nonlocal merge between those nodes. This nonlocal merge has been depicted in figure 3.8.
by a dashed line at level \(= 3\). Such a nonlocal merge introduces non-planarity in the switching-graph which in turn forces the VLSI routing to be done in more than one metal layer in order to avoid any possible short-circuit. Consequently the design rules governing the dimensions and separations of the wires and the connecting primitives comes into play which contributes in increasing the routing area and thereby the overall silicon area.

In the present version of our work, we have not implemented non-local merge. However, we have made provisions so that non-local merge may be easily incorporated in our tool in future.
3.4.2 Sneak Path

Jaekel, Bandyopadhyay and Jullien [18] have developed a heuristic to detect and completely avoid the creation of any sneak path during the process of switching-graph generation. The principle governing the heuristic for sneak path detection has been described next using a simple example.

Figure 3.9(a) shows the connectivity model of a switching-graph whose type-I nodes have been generated up to level 2. All the type-I nodes correspond to nmos transistors except the type-I node with id = 5, which has flag = 0 (indicated by the \( \phi \) in the figure) implying that the node corresponds to a wire and not an nmos transistor. \( 4_a, 5_a \) and \( 6_a \) are the type-II nodes having the onsets \{0,1\}, \{0\}
and \{0\} respectively. We find that the type-II nodes $5_a$ and $6_a$ are compatible and therefore they can be merged. Figure 3.9(b) shows how this merge creates a sneak path if the type-II nodes $5_a$ and $6_a$ are merged. If we carry out the merge, the charge at the evaluation node gets a path to ground through the type-II node $4_a$. The desired path that we expect is for the minterm $x_1x_2'x_3'$ only. But we find that an extra path also gets generated for minterm $x_1x_2'x_3$ which leads to incorrect functionality of the circuit. Whenever we find that a merge leads to a sneak path, we refrain from carrying out the merge.
3.5 A Heuristic Procedure for synthesizing Switching-graphs

The heuristic procedure for obtaining the switching-graph from a single-output or a multi-output Boolean function has been developed in an abstract form by Jaekel, Bandyopadhyay and Jullien [18]. In this section, we elaborate on these heuristics.

The switching-graph minimization algorithm assumes a predefined ordering of the input variables \((x_1, x_2, \ldots, x_n)\) and considers each variable in sequence according to their order from top to bottom. Choosing a good ordering is a problem in itself which we are not discussing here. The algorithm proceeds in a level 3.3 by level fashion.

3.5.1 The top level algorithm

Consider a switching function of \(n\) variables \(x_n, \ldots, x_1\). We will denote the current level by \(l\)

Begin

\(l := 1;\)

Repeat

- generate the children and the alternate forms (i.e. form-I and form-III or form-IV expansions) at level \(l\). Let the number of type-II nodes at level \(l\) be \(m\).
• For $j := 1$ to $m - 1$ do

• For $k := j$ to $m$ do

Begin

Check whether type-II nodes $j_a$ and $k_a$ may be merged using local merge.

If local merge is possible and sneak path does not arise then carry out the merge

End

$l := l + 1$;

Until $l = n$

End

When the algorithm is first called, the switching function to be implemented is specified by means of a single type-II node with two sets of minterms corresponding to the on-set and the don't-care-set of minterms respectively of the function to be implemented. When the algorithm finishes, we have a set of type-I nodes for all the levels 1 to $n$.

In the above algorithm we have considered only single-output switching functions as possible inputs to the algorithm. However, given a multi-output switching function in the form of a number of $n$-variable switching functions, we
can always combine them into a single composite function. We illustrate this with a simple example involving 2 functions of n variables each.

Let \( f_1(x_n, \ldots, x_1) \) and \( f_2(x_n, \ldots, x_1) \) be two n-variable switching functions.

Using a new variable \( x_{n+1} \) whose binary weight is double the binary weight of \( x_n \), we can generate a composite function \( f(x_{n+1}, x_n, \ldots, x_1) \) of \( n+1 \) variables such that

\[
f(x_{n+1}, x_n, \ldots, x_1) = \overline{x_{n+1}}f_1(x_n, \ldots, x_1) + x_{n+1}f_2(x_n, \ldots, x_1)
\]

The function \( f(x_{n+1}, x_n, \ldots, x_1) \) may now be used as an input to our heuristic procedure. After the switching-graph has been synthesized, the type-1 nodes at level = 1 corresponding to the new variable \( x_{n+1} \), are deleted as shown in figure 3.10 to obtain the two-output function.†

† Figure 3.10(a) shows the single-output function and the figure 3.10(b) shows the two-output function with the nodes in the level = 1 deleted. \( 4, 5, 6, \) and \( 7 \) refer to the the subgraphs which are connected to the type-1 nodes with ids \( 4, 5, 6 \) and 7 respectively. It is possible that these subgraphs have some nodes and edges in common.
3.6 Summary

In this chapter we have introduced the concept of switching-graph, discussed its structure, presented the connectivity and the contiguity models for a switching-graph, discussed the concept of local and nonlocal merge, elaborated the principle
governing the heuristic for sneak path detection and finally presented the heuristic synthesis technique that we have implemented.
Chapter 4  THE TOPOLOGICAL LAYOUT GENERATOR

4.1 Introduction

While developing a topological style, we need to take a number of considerations into account in order to satisfy the requirements for a VLSI circuit layout. The layout style for automated generation of the layout should have an ordered structure, subject to the consideration that the topological style must not involve an excessive amount of silicon area since area is of primary importance in determining the cost of the chip [21]. Further, VLSI process technology is changing rapidly and it is extremely important that a given layout may be adapted to any new technology and design rules without having to modify the layout generation strategy radically.

In 1980, Lopez and Law [27] developed a method of topological layout called Gate-Matrix layout which consists of a set of characters arranged in a specific order (depending on the circuit it represents) where each character represents a layout primitive. In 1993, Jullien [21] introduced a new style for topological layout for representing switching-graphs, and, in this dissertation, we will refer to this representation as the \textit{topological layout matrix}.

The operators used in the topological layout matrix are as follows:

- \textbf{W}# represents a wire segment
- G# represents an nmos transistor with $x_i$ as input
- R# represents an nmos transistor with $\bar{x}_i$ as input
- b# represents a blank space

We now discuss the principle behind the mapping of the switching-graph to the corresponding topological layout matrix. Let us assume that the topological layout matrix has $k_1, k_2, ..., k_j$ columns where $j$ is a natural number. Figure 4.1(a) depicts a partial view of a switching-graph having only type-I nodes. In the figure, we find two type-I nodes with triplets $(i, m, -1)$ and $(i, n, 1)$ respectively. Also, in the figure, $m_k(n_k)$ refers to the subgraph which is connected to the type-I node $m(n)$. It is possible that $m_k$ and $n_k$ have some nodes and edges in common. If we assume that the type-I nodes in level $i$ correspond to the operators in $r^{th}$ row of the topological layout matrix and that type-I node $m(n)$ corresponds to the $k_1^{th}$ ($k_p^{th}$) column of the topological layout matrix where $k_p > k_1$, $p$ is a positive integer, then

i) in the $r^{th}$ row of the matrix, we place R# in its $k_1^{th}$ column and b# in every column of the matrix starting from the $k_1 + 1^{th}$ column to $(k_p - 1)^{th}$ column

ii) in the $(r - 1)^{th}$ row of the matrix, we place W# in every column of the matrix starting from the $k_1^{th}$ column to $(k_p - 1)^{th}$ column

iii) in the $r^{th}$ row of the matrix, we place G# in its $k_p^{th}$ column and b# in every column of the matrix starting from the $(k_p + 1)^{th}$ column to $k_j^{th}$ column
iv) in the \((r - 1)^{th}\) row of the matrix, we place \(b\#\) in every column of the matrix starting from the \(k_p^{th}\) column to \(k_j^{th}\) column.

We show the partial topological layout matrix corresponding to 4.1(a) in table 4.1.

Figure 4.1
Similarly if we consider figure 4.1(b) where one of the type-1 nodes at level = i corresponds to the triplet (i, m, 0), rest remaining the same as figure 4.1(a), then

i) in the r\textsuperscript{th} row of the matrix, we place W# in its k\textsubscript{1}\textsuperscript{th} column and b# in every column of the matrix starting from the (k\textsubscript{1} + 1)\textsuperscript{th} column to (k\textsubscript{p} - 1)\textsuperscript{th} column

ii) in the (r − 1)\textsuperscript{th} row of the matrix, we place b# in every column of the matrix starting from the k\textsubscript{1}\textsuperscript{th} column to (k\textsubscript{p} - 1)\textsuperscript{th} column.
iii) in the $r^{th}$ row of the matrix, we place $G#$ in its $k_p^{th}$ column and $b#$ in every column of the matrix starting from the $(k_p + 1)^{th}$ column to $k_j^{th}$ column

iv) in the $(r - 1)^{th}$ row of the matrix, we place $b#$ in every column of the matrix starting from the $k_p^{th}$ column to $k_j^{th}$ column

We show the partial topological layout matrix corresponding to 4.1(b) in table 4.2.

It is evident that figure 4.1(b) corresponds to form-IV expansion of the switching-graph in figure 4.1(a). If we interchange the positions of type-1 node $m$ and its subgraph $m_s$ with type-1 node $n$ and its subgraph $n_s$, we get the form-III expansion of the switching-graph in figure 4.1(a). Using a similar line of explanation, we get the topological layout matrix of the form-III expansion as illustrated in table 4.3.

<table>
<thead>
<tr>
<th>cols $\rightarrow$</th>
<th>$k_1$</th>
<th>$k_2$</th>
<th>$\ldots$</th>
<th>$k_p - 1$</th>
<th>$k_p$</th>
<th>$\ldots$</th>
<th>$k_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>*</td>
<td>*</td>
<td>$\ldots$</td>
<td>*</td>
<td>*</td>
<td>$\ldots$</td>
<td>*</td>
</tr>
<tr>
<td>$(r-1)^{th}$ row</td>
<td>b#</td>
<td>b#</td>
<td>$\ldots$</td>
<td>b#</td>
<td>b#</td>
<td>$\ldots$</td>
<td>b#</td>
</tr>
<tr>
<td>$r^{th}$ row</td>
<td>G#</td>
<td>b#</td>
<td>$\ldots$</td>
<td>b#</td>
<td>W#</td>
<td>$\ldots$</td>
<td>b#</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>*</td>
<td>$\ldots$</td>
<td>*</td>
<td>*</td>
<td>$\ldots$</td>
<td>*</td>
</tr>
</tbody>
</table>

Table 4.3

Using similar of explanations, we can recursively map the whole switching-graph completely into the corresponding topological layout matrix. We illustrate
a switching-graph for the sum-bit of an one-bit full-adder, consisting of only type-I nodes and the corresponding topological layout matrix below.

Example: The figure 4.2(c) represents the topological layout matrix for the sum-bit of a one-bit full-adder whose truth table is in figure 4.2(a) and whose connectivity model for the switching-graph consisting only of type-I nodes is shown in 4.2(b). In a topological layout matrix, every even row, except the last

```
row 1  W#W# b# b#
row 2  R# b# G# b#
row 3  W# b# W# b#
row 4  G# R# G# R#
row 5  b#W# b# b#
row 6  R# b# G# R#
row 7  W#W#W# b#
row 8  b# b# b# R#
```

(c)  Topological-matrix for the sum bit of one-bit full-adder

Figure 4.2

even row, corresponds to a level in the switching-graph and therefore contains
the operators R#, G#, W# or b# while the odd rows contain W# or b#. For example, in the topological layout matrix shown in figure 4.2, the R# and G# in the second row represents the nmos transistors corresponding to the nodes 2 and 3 respectively. The two W#'s in the first row implies that the drains of the two nmos transistors in the second row are connected by wire. The last even row contains one or more R# representing the ground-switches necessary for dynamic CMOS logic implementations. For example, in figure 4.2, the R# in the eighth row represents an nmos transistor for realizing one ground-switch. In the next section we describe an algorithm for generating the topological layout matrix from the contiguity model switching-graph.

4.2 Algorithm for Generating Topological Layout

The input to the algorithm is the list of nodes at the last level of the switching-graph and the output is a set of lists each of which represents a column of nodes (or nmos transistors) in the topological layout matrix of the switching-graph.

We take the flippable-nonflippable structure corresponding to the last level of the contiguity model of the switching-graph and run a traversal algorithm on it to collect all the ids of the type-I nodes, present in that flippable-nonflippable structure, in a list XT, in the same sequence in which they occur in the structure. For example, if the flippable-nonflippable structure corresponding to the last level
of the contiguity model of a switching-graph is \( f(n(4.5), n(7.0)) \), then the list XT should be \([4, 5, 7, 6]\).

The Algorithm:

Consider XT to be \([n_1, n_2, \ldots, n_j]\) where \(n\) and \(j\) are integer.

Let CT_1, CT_2, \ldots, CT_j be lists each of which is initialized to \([1]\).

Begin

\( j := 1; \)

Repeat

Step 1: Delete element \(n_j\) from XT and insert it in CT_j;

Step 2: \(father := n_j \text{ div } 2;\)

Step 3: If \(father = 1\) or \(father\) belongs to CT_k, for some \(k, 1 \leq k < j\)

\then Goto Step 5 Else insert \(father\) at the front of CT_j;

Step 4: \(n_j := father;\) Goto Step 2;

Step 5: \(j := j + 1;\)

Until XT is empty

End

For instance, in figure 4.2(b) we find that the flippable-nonflippable structure corresponding to the last level of the switching-graph is \( f(n(10.8), n(15.12)) \). Hence XT will be \([10, 8, 15, 12]\). On running the above algorithm we will obtain CT_1 = \([10, 5, 2]\), CT_2 = \([8, 4]\), CT_3 = \([15, 7, 3]\) and CT_4 = \([12, 6]\) which
corresponds to the first, second, third and fourth columns respectively from left to right in the topological layout matrix in figure 4.2(c). The node-ids, from left to right in each list, correspond to sixth, fourth and second rows respectively wherever necessary.

4.3 Summary

In this chapter we have discussed, in detail, the topological layout matrix developed by Jullien [21], explained the principle behind the mapping of the switching-graph to the corresponding topological layout matrix and finally presented our algorithm for translating a switching-graph to a topological layout matrix.
Chapter 5 USE OF LOGIC PROGRAMMING IN
THE SYNTHESIS OF SWITCHING-GRAHPS
AND TOPOLOGICAL LAYOUT

5.1 Introduction

The use of logic programming paradigm in the field of CAD for VLSI is relatively new [14]. Although propositional logic has long been used for describing the truth functions of the combinational circuits, the more powerful predicate calculus on which logic programming is based has seen relatively little use in design automation [9]. For the past few years there has been a few successful attempts in the implementation of VLSI graphic Editors, VLSI Routers and silicon compilers [32] [14] [7] using the logic programming language Prolog.

In our work, we have used Prolog for implementing the switching-graph synthesizer and the topological layout generator. We have chosen Prolog as the language of implementation for the following reasons:

i) it helps in the declarative description of the database describing the switching-graph,

ii) its clean semantics and uniform data and program structure make the source code more readable than that based on traditional imperative programming languages [31] [33],

iii) its popularity as a rapid prototyping language,
iv) its simple schemes for list manipulation and its facility for dynamic dismantling of data structures and

v) its built-in backtracking facility which enables a program to undo all the effects of an operation and thereby restore to its original state when the program comes to an impasse. Thus the backtracking facility allows automatic maintenance of the integrity of the database unlike traditional programming languages which demands extra effort for the implementation of backtracking and careful attention to preserve the database integrity.

In the next section we will briefly describe the database schema used to describe our switching-graph models in Prolog.

5.2 The Database Schema

5.2.1 Switching-graph database

We now describe the database schema for storing the relevant information about the nodes in the switching-graph models. For every node, we use a relation:

\[
\text{node}(\text{Nid}, \text{Level}, \text{Label}, E, D, \text{Match\_Boolean}, \text{Left\_node\_id})
\]

where

- \text{node} is the relation name
- \text{Nid} is the id of the node (a natural number)
- \text{Level} is the level of the node
• **Label** is 1, -1 or 0. If Label is 1(-1) then the node corresponds to an nmos transistor with uncomplemented (complemented) input. If Label is 0, then the node corresponds to a piece of wire.

• **E** is the onset corresponding to the node, specified as a list of minterms.

• **D** is the don’t-care-set corresponding to the node, specified as a list of minterms.

• **Match Boolean** is either *matched* or *notmatched* or *dummy* implying whether the node is merged or the node is not merged or the node is dummy (i.e. the node does not correspond to either an nmos transistor or a wire), respectively.

• **Left node id** is by default 0. Otherwise it is the id of another node to which this node is merged.

A node at any level can have at most two children with alternate forms (i.e. form-III or form-IV) if possible. During the synthesis process of a switching-graph, we represent the children of a node in the following manner in the Prolog knowledge base:

If A is a node at level j and its children are

• **A1** (if it has only one child) or

• **A1 and A2** (if it has two children) or

• **A1 and A2, A1' and A2'** (if A has two children A1 and A2 corresponding to form-I and also has two children A1' and A2' corresponding to either
then the representation of the children will be:

- \( A1 \) for one child
- \( A1 \times A2 \) for two children
- \( A1 \) or \( A1' \times A2 \) or \( A2' \) for two children and their alternate form.

### 5.2.2 Topological layout matrix database

Next we describe the database schema for storing the relevant information about the \( W\#, R\#, G\# \) and \( b\# \) operators of the topological layout matrix during the synthesis of that matrix. The \( W\# \) operator corresponds to a wire, \( R\# \) corresponds to an nmos transistor with complemented input, \( G\# \) corresponds to an nmos transistor with uncomplemented input and \( b\# \) corresponds to a blank space. With respect to the switching-graph models discussed in section 3.3, every \( G\#, R\# \) or \( W\# \) in an even row of a topological layout matrix corresponds to type-I nodes of the corresponding switching-graph whereas a group of \( W\#'s \) in an odd row of a topological layout matrix corresponds to a hyperedge in the corresponding switching-graph.

For every \( W\# \) operator in the topological layout matrix, we use a relation as follows:

| wire(Nid, Row, Column) |

59
where

- **wire** is the relation name implying that the relation corresponds to an wire operator \( W\# \)
- **Nid** is the id of the node (an integer) in the switching-graph model which the wire operator corresponds to
- **Row** is the row of the wire operator in the topological layout matrix
- **Column** is the column of the wire operator in the topological layout matrix

For every \( R\# \) operator in the topological layout matrix, we use a relation as follows:

\[
red(Nid, Row, Column)
\]

where

- **red** is the relation name implying that the relation corresponds to an \( R\# \) operator
- **Nid** is the id of the node (an integer) in the switching-graph model which the \( R\# \) operator corresponds to
- **Row** is the row of the \( R\# \) operator in the topological layout matrix
- **Column** is the column of the \( R\# \) operator in the topological layout matrix

For every \( G\# \) operator in the topological layout matrix, we use a relation as follows:

\[
green(Nid, Row, Column)
\]
where

- *green* is the relation name implying that the relation corresponds to an G# operator
- *Nid* is the id of the node (an integer) in the switching-graph model which the G# operator corresponds to
- *Row* is the row of the G# operator in the topological layout matrix
- *Column* is the column of the G# operator in the topological layout matrix

For every b# operator in the topological layout matrix, we use a relation as follows:

```
blank(Nid, Row, Column)
```

where

- *blank* is the relation name implying that the relation corresponds to an b# operator
- *Nid* can be any integer. This attribute is in fact, redundant in this relation. It has been included in this relation only in order to maintain the attribute-triplet just as in other three relations wire, red and green for the ease of implementation
- *Row* is the row of the b# operator in the topological layout matrix
- *Column* is the column of the b# operator in the topological layout matrix
Such choices of the relational database schema enables us to enhance the readability of the code and allows us to accomplish the Prolog-based implementation in a natural way.

5.3 Summary

In this chapter we have attempted to substantiate the use of logic programming paradigm in the implementation of the switching-graph synthesizer and the topological layout generator. Also we have discussed the schema for the switching-graph database and the topological layout matrix database.
Chapter 6  CONCLUSIONS AND FUTURE WORK

This chapter summarizes the results of the thesis and outlines some important extensions.

6.1 Main Features

In this thesis, we have described a prototype topological layout generator for switching-graphs — a novel building block based on dynamic CMOS logic. The interesting features of this work are given below:

- we have used a logic programming language (Prolog) to implement the prototype.
- we have implemented a logic synthesizer which keeps track of layout cost/benefit during the synthesis process itself.
- we have discussed the development and the implementation of an algorithm to generate the topological layout.
- we have described how a switching-graph may be modelled in terms of a connectivity model and a contiguity model.
- we have developed a group theoretic model for the contiguity model (refer to appendix B).

The input to the switching-graph synthesizer is the on-set and the don’t-care-set of minterms for the function we wish to implement. The output of
the synthesizer is a file containing a specification for switching-graph using a hypergraph model [3] and a list of nodes representing all possible relative positions of nodes in the VLSI layout. This file is the input for the topological layout generator. The output generated by the topological layout generator is a technology-independent topological layout matrix. This matrix when used as input to an existing module generator [21] generates an area-optimized mask layout.

6.2 Conclusions

We found that the logic programming paradigm is quite appropriate for implementing the switching-graph synthesizer and the topological layout generator due to the following reasons:

- the rule-based environment of Prolog with its simple list manipulation techniques and built-in backtracking has aided in the quick development of the frameworks for both the logic synthesis and layout generation.
- Prolog's declarative nature has made the source code more readable than the programs based on traditional imperative languages
- considering the fact that we used fairly complex heuristics and list processing, the code was remarkably compact.*

* It has been reported in [31] [32] [33] that if a program is codified both in Prolog and an imperative language such as C, the approximate ratio of lines of code in Prolog to that in C is 1 : 10.
6.3 Future Work

During this work, we had originally intended to implement the concept of both local merge as well as and nonlocal merge. Although we have successfully implemented of the local merge, we have not implemented nonlocal merges. We have made provisions in our program so that non-local merge may be easily incorporated in our tool at a future date. Besides, in order to incorporate the concept of non local merge in the topological layout, we need to modify the structure of the layout matrix.

In our implementation of the layout generator, we generate technology independent topological layouts. It is however, always possible to extend our Prolog program to implement a strategy-independent mask layout generator as well. Such a generator would be useful when experimenting with alternate strategies.
BIBLIOGRAPHY


Appendix A
SAMPLE RESULTS

Here we present some of the results related to the switching-graph synthesizer and the topological layout generator developed by us. Below, we illustrate the topological layout matrix for pipelined modulo-7 multiplier. In figure A.1, we show the corresponding mask layout using a 1.2µ n-well CMOS process.
Next we present two tables, table A.1 and A.2. Table A.1 illustrates the CPU time required to execute the switching-graph synthesizer for modulo-7 and modulo-17 multipliers while table A.2 illustrates the CPU time required to execute the topological layout generator. We have used Quintus Prolog® running on SUN SPARC IPX®.

<table>
<thead>
<tr>
<th>Execution time for switching-graph synthesizer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic logic block</td>
</tr>
<tr>
<td>Modulo-7 multiplier</td>
</tr>
<tr>
<td>Modulo-17 multiplier</td>
</tr>
</tbody>
</table>

Table A.1

<table>
<thead>
<tr>
<th>Execution time for topological layout generator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic logic block</td>
</tr>
<tr>
<td>Modulo-7 multiplier</td>
</tr>
<tr>
<td>Modulo-17 multiplier</td>
</tr>
</tbody>
</table>

Table A.2
Appendix B
ABSTRACT ALGEBRAIC INTERPRETATION OF
THE FLIPPABLE-NONFLIPPABLE STRUCTURE

B.1 Introduction

In this appendix we attempt to interpret the flippable-nonflippable structure discussed in subsection 3.3.2 of section 3.3 and the concept of local merge discussed in subsection 3.4.1 of section 3.4 using an approach based on abstract algebra. We first review the concept of a poset, related abstract algebraic definitions and the concept of a permutation group and then use those concepts for our interpretations.

B.2 Concept of a Poset

Let \( P \) be a non-empty set. A binary relation \( R \) on \( P \) that is reflexive, antisymmetric and transitive is called a partial order. If a set \( P \) has a partial order defined on it then \( P \) is called a poset ordered over the relation \( R \).

Definitions

LEQ (Left-of or equal to): Let \( P = \{a_1, a_2, a_3, b_1, b_2, b_3\} \). Then LEQ is a
binary relation such that

\[ a_1 \text{ LEQ } a_1, \ a_2 \text{ LEQ } a_2, \ a_3 \text{ LEQ } a_3 \]
\[ b_1 \text{ LEQ } b_1, \ b_2 \text{ LEQ } b_2, \ b_3 \text{ LEQ } b_3 \]
\[ a_1 \text{ LEQ } a_2, \ a_2 \text{ LEQ } a_3, \ a_1 \text{ LEQ } a_3 \]
\[ b_1 \text{ LEQ } b_2, \ b_2 \text{ LEQ } b_3, \ b_1 \text{ LEQ } b_3 \]

The first two rows insure that relation LEQ is reflexive, and the next two rows insure that it is transitive. It is also evident that the relation is antisymmetric. Hence LEQ is a partial order on set P. Henceforth, we will represent LEQ by the symbol \( \preceq \).

It is to be noted that every pair of element need not be related to each other in a poset. In fact, if \( x \) and \( y \) belong to a poset \( P \) ordered over some relation \( R \) but not related to each other by \( R \), then \( x \) and \( y \) are said to be incomparable. Otherwise they are comparable.

An element \( b \) in a poset \( P \) is said to cover an element \( a \) in \( P \) if there is an element \( y \) such that if \( a \preceq y \preceq b \) and whenever \( a \preceq y \preceq b \), either \( y = a \) or \( y = b \). In other words, \( b \) covers \( a \) if \( a \preceq b \) and there are no elements of \( P \) “between” \( a \) and \( b \).

Consider the poset \( P = \{ a_1, a_2, \ldots, a_n \} \), such that \( n \) is a natural number, ordered over the relation \( \preceq \). Traversing \( P \) from left to right, the leftmost element \( a_1 \) is called the minimum element and the rightmost element \( a_n \) is called the maximum element.
**Poset-append:** Let P1 and P2 are two posets each ordered over relation \( \preceq \) such that all the elements of P1 are comparable to each other and all the elements of P2 are comparable to each other. If P2 is appended to P1 to generate a new poset P which is ordered over relation \( \preceq \) such that all the elements of P are comparable to each other then the append operation is termed as *poset-append*.

### B.3 Concept of a Permutation Group

On a finite set S of some elements, a **permutation** \( \pi \) is a bijective mapping \( \pi : S \rightarrow S \). The cardinality of the set S, \( |S| \) is referred to as the **degree** of permutation. The permutation is an **identity permutation** if every element of S takes into itself.

If \( \pi_i \) and \( \pi_j \) are the two different permutations on the set S, then the **composition** of these two permutations, \( \pi_j \circ \pi_i \), where \( \circ \) denotes the binary operation called **composition**, is the permutation obtained by permuting the elements of S by the application of \( \pi_i \) followed by an application of \( \pi_j \).

A collection of such permutations \( \{\pi_1, \pi_2, \ldots, \pi_m\} \) acting on a set S with cardinality \( k \) forms a group under **composition**, if closure, associativity, identity and inverse properties are satisfied. The set of all \( k! \) permutations on the set S form a permutation group called **symmetric group** of order \( k! \) and degree \( k \).
B.4 The flippable-nonflippable structure: A permutation group

At any level of the switching-graph, a flippable-nonflippable structure of the contiguity model at that level, can be interpreted in terms of a poset ordered over the relation \( \preceq \). For example, if the flippable-nonflippable structure at level = 3 of a switching-graph is \( f(n(f(8, 9), 10), n(15, 12)) \), it can be represented as a poset \( P = \{8, 9, 10, 15, 12\} \) ordered over \( \preceq \).

In a flippable-nonflippable structure such as \( f(n(f(8, 9), 10), n(15, 12)) \), we shall call the left argument \( n(f(8, 9), 10) \) of the structure, the left sub-structure and the right argument \( n(15, 12) \) of the structure, the right sub-structure.

**Proposition:** Let \( P_l = \{a_1, a_2, \ldots, a_{(i+1)}, a_i\} \) be the poset ordered over relation \( \preceq \) containing the nodes of the left sub-structure, one of whose node is a potential candidate for getting merged locally with a node in the right sub-structure. Then the set of all possible permutations of the nodes of the left sub-structure form the symmetric group of order \( i! \) and degree \( i \) if and only if every functor in the sub-structure is a flippable functor \( f \). Same arguments hold for the nodes of the right sub-structure.

**Proof:** The proof is obvious because all the functors of the sub-structure being \( f \), any node can take into any other node in a permutation. In other words, every
node can interchange its position with every other node.

**Proposition:** Let $P_1 = \{a_1, a_2, \ldots, a_{(i-1)}, a_i\}$ be the poset ordered over containing the nodes of the flippable-nonflippable structure at any level. If every functor in the structure is a non-flippable functor $n$ then the only possible permutation acting on $P_1$ is an identity permutation.

**Proof:** All the functors being $n$, it is not possible for any node to interchange its position with any other node. Hence the only possible permutation corresponds to the bijective function $\pi : P_1 \rightarrow P_1$ which takes every element into itself.

**B.5 Interpretation of a local-merge**

Let $P_1 = \{a_1, a_2, \ldots, a_{(m-1)}, a_m\}$ (m is a positive integer) be the poset ordered over containing the nodes of the left sub-structure, one of whose node is a potential candidate for getting merged locally with a node in the right sub-structure. Let the poset containing the nodes of the right sub-structure ordered over be $P_2 = \{b_1, b_2, \ldots, b_{(n-1)}, b_n\}$ (n is a positive integer). For carrying out the local merge, let $\pi_1$ be the permutation done on $P_1$ to generate a poset $P_{1_{new}}$ and $\pi_2$ be the permutation done on $P_2$ to generate a poset $P_{2_{new}}$ where both $P_{1_{new}}$ and $P_{2_{new}}$ are ordered over relation $\leq$. Let $P_{new}$ be the poset obtained by poset-appending $P_{2_{new}}$ to $P_{1_{new}}$. The local merge is said to be successful if and only if the minimum element of $P_{2_{new}}$ covers the maximum element of $P_{1_{new}}$ in the poset $P_{new}$. 
Appendix C

PROLOG CODE FOR SWITCHING-GRAH SYNTHESIZER

In the code below we have used the terms matched and notmatched implying merged or not merged respectively.

:-op(300, xfy, or).
:- use_module(library(sets)).
:- use_module(library(lists)).
:- use_module(library(basics)).

The following rules are used to reverse a list.

/*****************************/
d_rev([], X/X).
d_rev([X|Y], Z/Z2):- d_rev(Y, Z/0[X|Z2]).
/*****************************/

The following rule calls a rule power to calculate the value of 2 to the power of the number of variables in a multi-output function and then calls the rule make_one_function which converts the multi-output switching function to a single-output switching function.

/*****************************/
combined_function(ListOfFunctions, InitialNumOfVar, EResult/DResult):-
    power(2, InitialNumOfVar, RaisedToValue),
    make_one_function(0, ListOfFunctions, RaisedToValue, [], [], EResult/EResult),
    d_rev(RevEResult, EResult/[]),
    d_rev(RevDResult, DResult/[]).
/*****************************/

The following rule converts the multi-output switching function to a single-output switching function.

/*****************************/
make_one_function(_, [[]], _, X/Y, X/Y):- !.
make_one_function(Counter, [EH/DH | T], RaisedToValue, ESofar/DSofer, EResult/DResult):-
    add_product_to_H_elements(Counter, RaisedToValue, EH, ESofar, ENewSofar),
    add_product_to_H_elements(Counter, RaisedToValue, DH, DSofer, DNewSofar),
    NewCounter is Counter + 1,
    make_one_function(Counter, T, RaisedToValue, ENewSofar/DNewSofar, EResult/DResult).
/*****************************/

This function calculates the value of 2 to the power of a number.
The format is: power(Base, Exponent, Result).

power(_, 0, 1) :- !.
power(N, M, F) :- Mnew is M-1,
               power(N, Mnew, Fnew),
               F is Fnew * N.

The following rules add the product of the counter
and a number (which is the value of 2 to the power
of the number of variables of the multi-output
switching function) and add that number to the
elements of on-set and don't-care-set of the
relevant switching function in the multi-output
function.

add_product_to_H_elements(_, _, [], X, X) :- !.

add_product_to_H_elements(Counter, RaisedToValue,
                          [H|T], Hsofar, Hfinal):-
                         ProductOf.Counter_and_RaisedToValue is
                         Counter * RaisedToValue,
                         Hnew is
                         ProductOf.Counter_and_RaisedToValue + H,
                         add_product_to_H_elements(Counter,
                          RaisedToValue, T, [Hnew|Hsofar], Hfinal).

The following rule gives the number of extra
variables required for accommodating a multi-output
function into one single-output function. It calls
the rule get_no_of_vars_from_no_of_states which in
turn calls the rule do to accomplish the job.

The format is:

get_extra_no_of_vars(
    No_of_functions_in_the_multioutput_function,
    No_of_extra_vars)

get_extra_no_of_vars(No_of_functions,
    No_of_extra_vars):-
    get_no_of_vars_from_no_of_states(No_of_functions,
    No_of_extra_vars).

get_no_of_vars_from_no_of_states(No_of_states,
    No_of_vars):-
    do(1, No_of_states, No_of_vars).

do(_, 1, 0) :- !.

do(Index, No_of_states, Index):-
    power(2, Index, Result1),
    OneLessThanIndex is Index - 1,
    power(2, OneLessThanIndex, Result2),
    No_of_states <= Result1,
    No_of_states > Result2, !.

do(Index, No_of_states, No_of_vars):-
    UpdatedIndex is Index + 1,
do( UpdatedIndex, No_of_states, No_of_vars).

This rule first combines the multi-output switching function (if any) to a single-output switching function and then proceeds to synthesize the switching-graph corresponding to the single-output switching function. If the input is a single-output switching function, then the rule retains that switching function and proceeds to synthesize the switching-graph using that function.

begin(List_of_E_and_D_set, InitialNumOfVar):-
  combine_function(List_of_E_and_D_set,
                   InitialNumOfVar, ECombined/DCombined),
  no_of_functions(List_of_E_and_D_set,
                  No_of_functions),
  get_extra_no_of_vars(No_of_functions,
                        ExtraNumOfVars),
  FinalNumOfVar is InitialNumOfVar + ExtraNumOfVars,
  One_less_than_FinalNumOfVar is FinalNumOfVar - 1,
  power(2, One_less_than_FinalNumOfVar, Wt),
  synthesize({{1, 0, 0, ECombined, DCombined, notmatched, 0}, Wt, 1}).

This rule executes the synthesizer. It opens a file graphNew and writes all the nodes of the switching-graph in the format of the relation node(Nid, Level, Label, E, D, Match_boolean,

Left_node_id).

synthesize(List_of_nodes_at_a_level, Wt, Starting_CStruct):=  
  tell(graphNew),
  assertz_each_node(List_of_nodes_at_a_level),
  write_each_node(List_of_nodes_at_a_level),
  make_graph(List_of_nodes_at_a_level, Wt,
             Starting_CStruct),
  nl, nl, nl, nl, nl,
  write('flip'), write('('),
  write(Starting_CStruct), write(')'), write('.'), nl,
  told.

The following rule is activated when the last level of the switching-graph is reached. It generates the children for the last level in the node relation format, asserts each child in the dynamic database and writes each child to the output file. It also computes the actual number of rows for the corresponding topological-matrix (which is equal to the double of the last level-number) and writes it to the output file.

make_graph(List_of_nodes_at_a_level, 1,
generate_children_and_alternate_forms(
    List_of_nodes_at_a_level, 1,
    Children_at_next_level,
    Starting_CStruct,
    Final_CStruct),
separate_children(Children_at_next_level,
    List_of_all_single_child),
assertz_each_node(List_of_all_single_child),
write_each_node(List_of_all_single_child),
write_number_of_rows_required_for_electric_matrix(
    List_of_all_single_child).

The following rule is activated when the list
List_of_nodes_at_a_level has only one element,
the starting node. It generates the nodes with
id 2 and 3, generates the corresponding flippen-
nonflippable structure, assert the nodes in
dynamic database, writes them in the output file,
calculates the weight of the next level and then
recurses with the new list containing the nodes 2
and 3, the weight of the next level of the
switching-graph and the new flippable-nonflippable
structure.

make_graph([(1,0,0,E,D,notmatched,0)], Wt,
    Starting_CStruct) :- !,  
    generate_children_and_alternate_forms(
        [(1,0,0,E,D,notmatched,0)], Wt,
        Children_at_next_level1,
        Starting_CStruct,
        Next_level_CStruct),
    separate_children(Children_at_next_level1,
        List_of_single_child2and3),
    assertz_each_node(List_of_single_child2and3),
    write_each_node(List_of_single_child2and3),
    Wt_at_next_level is Wt 7/ 2,
    make_graph(List_of_single_child2and3,
        Wt at next level,
        Next_Level_CStruct).

This rule is activated when the rule (i) and (ii) are
not satisfied. It generates the children and their
alternate forms (i.e. either form-III or form-IV
expansions, if any), invokes the rule
generate_shared_list_of_nodes which returns a list
Modified List_of_single_nodes_in_ascending_order,
of all the nodes (including locally merged nodes ,
if any) at the level corresponding to the weight Wt,
the new flippable-nonflippable structure in
Next_level_CStruct, assert the nodes in dynamic
database, writes them in the output file, calculates the weight of the next level and then recurse with the list

```
Modified_List_of_single_nodes_in_ascending_order,
the weight of the next level of the switching-graph
and the newly generated flippable-nonflippable
structure.
```

```
make_graph(List_of_nodes_at_a_level, Wt,
       Starting_CStruct) :-

    generate_children_and_alternate_forms(
      List_of_nodes_at_a_level, Wt,
      Children_at_next_level1,
      Starting_CStruct,
      Next_level1_CStruct),

    get_the_shared_list_of_nodes(Wt,
      Children_at_next_level1,
      Next_level1_CStruct,
      [], _,
      Modified_List_of_single_nodes_in_ascending_order,
      Locally_flipped_CStruct,
      localpass), !, !,

    assert_each_node(
      Modified_List_of_single_nodes_in_ascending_order),

    write_each_node(
      Modified_List_of_single_nodes_in_ascending_order),

    Wt_at_next_level is Wt // 2,

    make_graph(
      Modified_List_of_single_nodes_in_ascending_order,
      Wt_at_next_level,
      Locally_flipped_CStruct).
```

The following rules convert all the children having X*Y form into single children X and Y
and retain the children of the form without *
(i.e. single child). Among the following rules
the second rule is required when the switching-
graph generates the children at last level. Since
our algorithm does not proceed to check the
possibility of any local merge for the nodes
generated at the last level, the children
of the form X1 or X2*Y1 or Y2 at the last level
are converted to two single children X1 and Y1.

```
/*
separate_children([], []).- !.
separate_children([X1 or X2*Y1 or Y2|T],
               [X1,Y1|Rest]);- !,
    separate_children(T, Rest).
separate_children([X*Y|T], [X,Y|Rest]);- !,
    separate_children(T, Rest).
separate_children([X|T], [X|Rest]);- !,
    separate_children(T, Rest).
*/
```
The following rules assert each of a list to the dynamic database.
assertz_each_node([]):- !.
assertz_each_node([X|T]):- 
    X = (A,B,C,D,E,F,G), 
    assertz(node(A,B,C,D,E,F,G)), 
    assertz_each_node(T).

The following rules write each of a list to an output file.
write_each_node([]):- !.
write_each_node([X|T]):- write(node), write('('), write(X), write(')'), 
    write_each_node(T).

The following rule
write_number_of_rows_required_for_electric_matrix
writes the number of rows (last level multiplied by 2) that would be present in the topological matrix.
write_number_of_rows_required_for_electric_matrix([], 
    _'(Level,_'-,_'-,_'-,_'-[_]),:- 
    nl, nl, nl, nl, 
    write(num rows), write('('), 
    Num rows is Level * 2, 
    write(Number of rows), write(')'), 
    write(')').

The following rule finds the number of switching functions present in the multi-output function
no_of_functions(List_of_E_and_D_set, 
    _NC_of_functions):- 
    length(List_of_E_and_D_set, No of functions).

The following rule computes the reduced function(s) of a given switching function.
reduced_function([],_, [], []). 
reduced_function([Head|Tail], Wt, 
    [Head|Small], Big):- Head < Wt, !, 
    reduced_function(Tail, Wt, Small, Big). 
reduced_function([Head|Tail], Wt, Small, 
    [Head new|Big]):- 
    Head new is Head - Wt, 
    reduced_function(Tail, Wt, Small, Big).

The following rule generates the form-III and
form-IV expansion of a switching function expressed in form-I, if in the form-I expansion, the onset of one of the reduced functions is the subset of the union of the on-set and the don't-care-set of the remaining reduced function.

covers(Essential_first, Don't care_first,
Essential_second, Don't care_second,
Don't care_first_new,
Essential_second_new,
Don't care_second_new):-
union(Essential_second,
Don't care_second, Result),
subset(Essential_first, Result),
intersection(Don't care_first, Result,
Don't care_first_new),
union(Don't care_second, Essential_first,
Don't care_second_new),
subtract(Essential_second,
Essential_first,
Essential_second_new).

The following rule checks if essential/don't-care set E0/D0 of a switching function and essential/don't-care set E1/D1 of another switching function are equivalent.

equivalent(E0, D0, E1, D1, E_new, D_new):-
union(E0, D0, Result0),
subset(E1, Result0),
union(E1, D1, Result1),
subset(E0, Result1),
union(E0, E1, E_new),
intersection(D0, D1, D_new).

The following rule generates children and alternate forms calls the rule generate children that generates the children of a node and its alternate forms and also generates the children in the compound-flip-structure (i.e. flippable-nonflippable structure).

generate_children_and_alternate_forms([], [[]], Struct, Struct):-!.
generate_children_and_alternate_forms([Nid, Level, Label, E, D, Match_boolean, Nid_of_first_child]Tail],
New_wt,
[Children_for_head|Children_for_Tail],
Cstructure_at_level1,
Cstructure_at_level2):-
generate_children([Nid, Level, Label,]
E, D, Match_boolean,
Nid_of_first_child,
New_Wt, Children_for_head,
Cstructure_at_level1,
Semi_Cstructure_at_level1),
generate_children_and_alternate_forms(Tail,
New_Wt, Children_for_Tail,
Semi_Cstructure_at_level1,
Cstructure_at_level2).

/**************** generate_children rules generate the children of a
node and its alternate form
*******************************/

The following generate_children rule is for
generating node ids 2 and 3 from 1
*******************************************************************************/
generate_children((1, _, _, E, D, notmatched, 0), Wt,
((2,1, 1, E_new0, D_new0, notmatched, 0),
(3,1, 1, E_new1, D_new1, notmatched, 0)),
Cstructure_at_level1,
Cstructure_at_level2):- !,
reduced_function(E, Wt, E_new0, E_new1),
reduced_function(D, Wt, D_new0, D_new1),
generate_the_nid_in_compound_structure(1,
_, Cstructure_at_level1,
Cstructure_at_level2).

/**************** generate_children rule is for
generating dummy child from the dummy parent
*******************************************************************************/
generate_children((Nid, Level, Label, E, D, dummy, 0),
(Nid_dummy, Level_new, Label, E, D, dummy, 0),
Cstructure_at_level1,
Cstructure_at_level2):- !,
Level_new Is Level + 1,
Nid_dummy is Nid * 2,
generate_the_nid_in_compound_structure(Nid,
(Nid_dummy, _, _, _, _, _, _),
Cstructure_at_level1,
Cstructure_at_level2).

The following generate_children rule is for
generating a dummy child from a 'matched' parent
if the 'matched' parent is at the immediate right
of a node which is also 'matched' to a node at
its left whose left_node_id's value is not 0
E.g. if A B C D are 4 matched nodes then C and D's
children should be dummy
*******************************************************************************/
generate_children((Nid, Level, Label, _, _, matched,
null
generate_children((Nid, Level, Label, E, D, matched,
    Left_node_id), Wt,
    Child_for_Head,
    Cstructure_at_level1,
    Cstructure_at_level2):-
node(Nid, Level, Label, _, _, matched,
    Left_node_id),
node(Right_node_id, _, _, _, notmatched, 0), !,
reduced_function(E, Wt, _E_new1),
reduced_function(D, Wt, D_new1),
f_c_right(Nid, Level, E_new1, D_new1,
    Child_for_Head),
generate_the_nid_in Compound_structure(Nid,
    Child_for_Head,
    Cstructure_at_level1,
    Cstructure_at_level2).

/****************************
The following rule generates a left child, phi
E.g. If A B C D are matched nodes then C and D
becomes dummy already and now A and B are left
to be taken care of. Now if A and B are equivalent
then the child from A is made phi according to the
following rule (The child for B has already been
made dummy using the rule given before the last
rule).
*****************************/
generate_children((Nid, Level, _, E, D, notmatched,
0), Wt,
    Child_for_Left_node,
    Cstructure_at_level1,
    Cstructure_at_level2):-
node(Nid, _, _, _, notmatched, 0),
node(Right_node_id, _, _, _, matched, Nid),
reduced_function(E, Wt, E_new0, E_new1),
reduced_function(D, Wt, D_new0, D_new1),
equivalent(E_new0, D_new0, E_new1, D_new1,
    E_new, D_new), !,
make_left_child_phi(Nid, Level, E_new,
    D_new, Child_for_Left_node),
generate_the_nid_in Compound_structure(Nid,
    Child_for_Left_node,
    Cstructure_at_level1,
    Cstructure_at_level2).

/****************************
The following rule takes care of the situation when
the reduced E/D's of left and right matched fathers
are not equivalent
*****************************/
generate_children((Nid, Level, _, E, D, notmatched,
0), Wt,
Child for Head,
Cstructure at level 1,
Cstructure at level 2):-

node (Nid, _, _, _, notmatched, 0),
node( , _, _, _, matched, Nid), !,
reduced_function(E, Wt, E_new0, _),
reduced_function(D, Wt, D_new0, _),
f_c_left(Nid, Level, E_new0, D_new0,

Child for Head),
generate_the_nid_in_compound_structure(Nid,
Child for Head,
Cstructure at level 1,
Cstructure at level 2).

The following rule takes care of the situation when
the alternatives given above are not satisfied
*******************************************************************************/
generate_children((Nid, Level, _, E, D, notmatched, 0), Wt,

Child for Head,
Cstructure at level 1,
Cstructure at level 2):-

reduced_function(E, Wt, E_new0, E_new1),
reduced_function(D, Wt, D_new0, D_new1),
f_c_a_a_f(Nid, Level, E_new0, E_new1,

D_new0, D_new1,

Children for Head),
generate_the_nid_in_compound_structure(Nid,

Children for Head,
Cstructure at level 1,
Cstructure at level 2).

*******************************************************************************/
The following rules, f_c_a_a_f, actually creates
the children and their alternate forms (if any)
*******************************************************************************/
*******************************************************************************/
when right child’s Essential set is []
*******************************************************************************/
f_c_a_a_f(N, L, EO, [], D0, _, ((New_node_id,

Level_new, -1, EO, D0, notmatched, 0))):-

New_node_id is (N * 2),
Level_new is L + 1, !.
*******************************************************************************/
when left child’s Essential set is []
*******************************************************************************/
f_c_a_a_f(N, L, [], EL, _, D1, ((New_node_id,

Level_new, 1, EL, D1, notmatched, 0))):-

New_node_id is (N * 2) + 1,
Level_new is L + 1, !.
*******************************************************************************/
when neither left child’s nor right child’s
Essential set is empty then there are three options
as follows:

% first option: merge the left and right siblings if
% they are equivalent
f_c_a_a_f(N, L, E0, E1, D0, D1, ((New_node_id,
    Level_new, 0, E_new, D_new,
    notmatched, 0))):-
    New_node_id is (N * 2),
    Level_new is L + 1,
    equivalent(E0, D0, E1, D1,
    E_new, D_new), !.

% second option: if first option is not true then get
% alternate forms if covering is true
% second sub-option1: E0 is covered by union of E1, D1
f_c_a_a_f(N, L, E0, E1, D0, D1,
    (New_node_id0, Level_new, -1, E0, D0,
    notmatched, 0)\or
    (New_node_id0, Level_new, 0, E0, D0_new,
    notmatched, 0)\or
    (New_node_id1, Level_new, 1, E1, D1,
    notmatched, 0)\or
    (New_node_id1, Level_new, 1, E1_new,
    D1_new, notmatched, 0))):-
    New_node_id0 is N*2,
    New_node_id1 is (N * 2) + 1,
    Level_new is L + 1,
    covers(E0, D0, E1, D1, D0_new,
    E1_new, D1_new), !.

% E0 is covered by union of E1, D1
% second sub-option2: E1 is covered by union of E0, D0
f_c_a_a_f(N, L, E0, E1, D0, D1,
    (New_node_id0, Level_new, -1, E0, D0,
    notmatched, 0)\or
    (New_node_id0, Level_new, -1, E0_new,
    DC_new,notmatched, 0)\or
    (New_node_id1, Level_new, 1, E1, D1,
    notmatched, 0)\or
    (New_node_id1, Level_new, 0, E1, D1_new,
    notmatched, 0))):-
    New_node_id0 is N*2,
    New_node_id1 is (N * 2) + 1,
    Level_new is L + 1,
    covers(E1, D1, E0, D0, D1_new,
    E0_new, D0_new), !.

% E1 is covered by union of E0, D0
% third option: if first or second option is not true
% i.e. merging is not possible and also alternate
% forms do not exits
f_c_a_a_f(N, L, E0, E1, D0, D1,
    (New_node_id0, Level_new, -1, E0, D0,
    notmatched, 0)\or
    (New_node_id1, Level_new, 1, E1, D1,
    notmatched, 0))
notmatched, 0)):-

    New_node_id0 is N2,
    New_node_id1 is (N * 2) + 1,
    Level_new is L + 1.

/**************************
The following rules are required when children are to be created from the matched nodes. It is to be noted that only two of the nodes from a group of matched nodes will give rise to one child each. Such a child may be a true child or a dummy child
**************************/
% when right child’s Essential set is [] then make the % right child dummy
f_c_right(N, L, [], _,
    ((New_node_id, Level_new, 1, [], [], dummy, 0))):-
    New_node_id is (N * 2) + 1,
    Level_new is L + 1, !.
% when right child’s Essential set is not empty then % create the right child
f_c_right(N, L, El, DL,
    ((New_node_id, Level_new, 1, El, DL, notmatched, 0))):-
    New_node_id is (N * 2) + 1,
    Level_new is L + 1, !.
% when left child’s Essential set is [] then make % the left child dummy
f_c_left(N, L, [], _,
    ((New_node_id, Level_new, -1, [], [], dummy, 0))):-
    New_node_id is (N * 2),
    Level_new is L + 1, !.
% when left child’s Essential set is not empty then % create the left child
f_c_left(N, L, EL, DL,
    ((New_node_id, Level_new, -1, EL, DL, notmatched, 0))):-
    New_node_id is (N * 2),
    Level_new is L + 1, !.
/**************************
The following rules are required when the reduced E/D’s of left and right matched fathers are equivalent. The left child is made phi and the right child is made dummy
**************************/
make_left_child_phi(N, L, E_new, D_new,
    ((New_node_id, Level_new, 0, E_new, D_new, notmatched, 0))):-
    New_node_id is (N * 2),
    Level_new is L + 1, !.
make_right_child_dummy(N, L, _, _,
    ((New_node_id, Level_new, 1, [], []),
副书记, 0)):-
New_node_id is \((N \times 2) + 1\),
Level_new is \(L + 1\), !.

The following rules generate the node-id in the
compound structure which is necessary to maintain
the contiguity model of the switching-graph. The
format is:
generate_the_nid_in_compound_structure(Parent_Nid,
Child,
Cstructure_at_level1,
Cstructure_at_level2).

*******************************************************************************
generate_the_nid_in_compound_structure(1,
\(f(2,3)\)) :- !.

generate_the_nid_in_compound_structure(Nid,
\((\text{Child_of_Nid},\_\_\_\_\_\_\_\_\_\_)\),
Cstructure_at_level1,
Cstructure_at_level2):-
arg(1, Cstructure_at_level1, X),
arg(2, Cstructure_at_level1, Y),
replace(Nid, Child_of_Nid, X,
Xnew), !,
functor(Cstructure_at_level1,
Functor, 2),
functor(Cstructure_at_level2,
Functor, 2),
arg(1, Cstructure_at_level2, Xnew),
arg(2, Cstructure_at_level2, Y).

generate_the_nid_in_compound_structure(Nid,
\((\text{Child_of_Nid},\_\_\_\_\_\_\_\_\_)\),
Cstructure_at_level1,
Cstructure_at_level2):-
arg(1, Cstructure_at_level1, X),
arg(2, Cstructure_at_level1, Y),
replace(Nid, Child_of_Nid, Y, Ynew), !,
functor(Cstructure_at_level1, Functor, 2),
functor(Cstructure_at_level2, Functor, 2),
arg(1, Cstructure_at_level2, X),
arg(2, Cstructure_at_level2, Ynew).

generate_the_nid_in_compound_structure(Nid,
\((\text{Child_of_Nid1},\_\_\_\_\_\_\_\_\_\_\_\_)\)*
\((\text{Child_of_Nid2},\_\_\_\_\_\_\_\_\_\_\_\_)\),
Cstructure_at_level1,
Cstructure_at_level2):-
arg(1, Cstructure_at_level1, X),
arg(2, Cstructure_at_level1, Y),
replace(Nid,
\(f(\text{Child_of_Nid1}, \text{Child_of_Nid2}),\)
X, Xnew), !,
functor(Cstructure_at_level1, Functor, 2),
functor(Cstructure_at_level2, Functor, 2),
arg(1, Cstructure_at_level2, Xnew),
arg(2, Cstructure_at_level2, Y).
generate_the_nid_in_compound_structure(Nid,
  ((Child_of_Nid1, _, _, _, _, _, _),(Child_of_Nid2, _, _, _, _, _, _)),
  Cstructure_at_level1, Cstructure_at_level2):-
  arg(1, Cstructure_at_level1, X),
  arg(2, Cstructure_at_level1, Y),
  replace(Nid, f(Child_of_Nid1, Child_of_Nid2), Y, Xnew), !,
  functor(Cstructure_at_level1, Functor, 2),
  functor(Cstructure_at_level2, Functor, 2),
  arg(1, Cstructure_at_level2, X),
  arg(2, Cstructure_at_level2, Y).
generate_the_nid_in_compound_structure(Nid,
  ((Child_of_Nid1, _, _, _, _, _),(Child_of_Nid2, _, _, _, _, _)),
  Cstructure_at_level1, Cstructure_at_level2):-
  arg(1, Cstructure_at_level1, X),
  arg(2, Cstructure_at_level1, Y),
  replace(Nid, f(Child_of_Nid1, Child_of_Nid2), X, Xnew), !,
  functor(Cstructure_at_level1, Functor, 2),
  functor(Cstructure_at_level2, Functor, 2),
  arg(1, Cstructure_at_level2, Xnew),
  arg(2, Cstructure_at_level2, Y).

The child of Nid can be either a number or a compound structure f(N1,N2) which is equivalent to N1*N2 in the connectivity model of the switching-graph.
The following rule replaces an Nid by its child(ren)
in the flippable-nonflippable structure and generates
a modified flippable-nonflippable structure.
The format of the following rules: replace(Nid, Child_of_Nid, Struct, Struct_new).

*******************************************************************************/
% when the struct is_same as Nid, then replace the Nid by Child
replace(Nid, Child, Nid, Child):- !.
replace(Nid, Child_of_Nid, Struct, Struct_new):-
    arg(1, Struct, Arg1),
    arg(2, Struct, Arg2),
    replace(Nid, Child_of_Nid, Arg1, Arg1_new), !,
    functor(Struct, Functor, 2),
    functor(Struct_new, Functor, 2),
    arg(1, Struct_new, Arg1_new),
    arg(2, Struct_new, Arg2).
replace(Nid, Child_of_Nid, Struct, Struct_new):-
    arg(1, Struct, Arg1),
    arg(2, Struct, Arg2),
    replace(Nid, Child_of_Nid, Arg2, Arg2_new), !,
    functor(Struct, Functor, 2),
    functor(Struct_new, Functor, 2),
    arg(1, Struct_new, Arg1),
    arg(2, Struct_new, Arg2_new).

/***************************************************************************/
get_the_shared_list_of_nodes(Wt, Input_List,
    Input_CStruct, Initial_Empty_List,
    Modified_List_of_single_nodes_in_ascending_order,
    Final_CStruct, Pass):-
    check_for_sharing(Wt, Input_List,
        Input_CStruct, Initial_Empty_List,
        Reversed_Final_List, Final_CStruct, Pass),
    make_list_of_single_nodes(
        Reversed_Final_List,
        List_of_single_nodes_in_reverse_descending_order),
    put_equivalent_E_and_D_on_every_matched_node(
        List_of_single_nodes_in_reverse_descending_order,
        Modified_List_of_single_nodes_in_ascending_order).

***************************************************************************
The following rule put the equivalent onset E and don't-care set D on every node in a group of matched nodes. This rule calls the rule put which actually carries out the job.
***************************************************************************
put_equivalent_E_and_D on every matched node:
  List of single nodes in reverse order,
Modified List of single nodes in ascending order):=
  put(List of single nodes in reverse order, []), []).
Modified List of single nodes in ascending order).
put([], _, X, X) :- !.

put([[(Nid1, Level, Label, Eq, Deq, matched, Nid2)]
  Tail],
  Symbol_table_list1,
  TempList, FinalList):-
absent in_symbol_table((Nid1, _, _), Nid2,
  Eq, Deq, Symbol_table_list1,
  Symbol_table_list2), !,

put(Tail, Symbol_table_list2,
  [(Nid1, Level, Label, Eq, Deq, matched, Nid2)|TempList], FinalList).
put([[(Nid1, Level, Label, _, _, matched, Nid2)|Tail],
  Symbol_table_list1,
  TempList, FinalList):-
get_E_and_D((Nid1, _, _), Nid2,
  Symbol_table_list1, Eq, Deq,
  Symbol_table_list2),

put(Tail, Symbol_table_list2,
  [(Nid1, Level, Label, Eq, Deq, matched, Nid2)|TempList], FinalList).
put([[(Nid1, Level, Label, E, D, notmatched, 0)|Tail],
  Symbol_table_list,
  TempList, FinalList):-
absent in_symbol_table((Nid1, _, _, _, _,
  Symbol_table_list, _), !),
put(Tail, Symbol_table_list,
  [(Nid1, Level, Label, E, D, notmatched, 0)|TempList], FinalList).
put([[(Nid1, Level, Label, _, _, notmatched, 0)|Tail],
  Symbol_table_list1,
  TempList, FinalList):-
get_E_and_D((Nid1, _, _), _,
  Symbol_table_list1, Eq, Deq, _),
put(Tail, Symbol_table_list1,
  [(Nid1, Level, Label, Eq, Deq,
  notmatched, 0)|TempList], FinalList).
put([[(Nid1, Level, Label, E, D, dummy, 0)|Tail],
  Symbol_table_list,
  TempList, FinalList):-
put(Tail, Symbol_table_list,
  [(Nid1, Level, Label, E, D, dummy, 0)|TempList], FinalList).

**************************************************************************
The following rule check if Nid1 is absent in the
list called Symbol_table_list.
********************************************
absent_in_symbol_table(Nid1, __, Nid2, Eeq, Deq, 
Symbol_table_list, 
[(Nid2, Eeq, Deq) | Symbol_table_list]):- 
nonempty([Nid1, __], 
Symbol_table_list).
********************************************

The following rules extract the equivalent 
on-set and the don't-care-set from the 
list called Symbol_table_list for Nid1 
which is already present in the Symbol_table_list. 
********************************************
get_E_and_D(Nid1, __, Nid2, 
:=[(Nid1, Eeq, Deq) | Tail], Eeq, Deq, 
[(Nid2, Eeq, Deq) | Tail]):- !.
get_E_and_D(Nid1, __, Nid2, [H | Tail], Eeq, Deq, 
[H | New_Tail]):- 
get_E_and_D(Nid1, __, Nid2, Tail, Eeq, Deq, 
New_Tail).
********************************************
The following rules convert all the node-pairs 
having X*Y form into single nodes Y and X 
and retain the children of the form without * 
(i.e. single child). 
********************************************
make_list_of_single_nodes([], []):- !.
make_list_of_single_nodes([X*Y|Tail], 
[Y,X | New_Tail]):- !,
make_list_of_single_nodes(Tail, 
New_Tail).
make_list_of_single_nodes([X|Tail], [X | New_Tail]):- 
make_list_of_single_nodes(Tail, 
New_Tail).
********************************************
The following rule check if sharing is possible 
in localpass. If possible, then it is carried out. 
********************************************
check_for_sharing(_, [], CStruct, X, X, CStruct, 
localpass).
check_for_sharing(Wt, [Head_children | Rest_children], 
Input_CStruct, Templist, 
Final_list, Final_CStruct, 
localpass):- 
extract_child(Head_children, 
Child, L_or_R_flag), 
find_match(Wt, Head_children, 
Child, Rest_children, 
New_rest_children, Input_CStruct, 
Semifinal_CStruct, localpass), !,
build_head_children_list(Child, 
    Head_children, 
    L_or_R_flag, 
    Modified_head_children),
find_match_for_right_child_if_exists_and_possible( 
    Wt, Modified_head_children, 
    L_or_R_flag, New_rest_children1, 
    New_rest_children2, Final_head_children, 
    Semifinal1_CStruct, Semifinal2_CStruct, 
    localpass),
check_for_sharing(Wt, New_rest_children2, 
    Semifinal2_CStruct, 
    [Final_head_children | Templist], 
    Final_list, Final_CStruct, 
    localpass).
check_for_sharing(Wt, [Head_children | Rest_children], 
    Input_CStruct, Templist, 
    Final_list, Final_CStruct, 
    localpass):-
select_alternative_for_head_children( 
    Head_children, 
    Head_children_alternative_deleted), 
check_for_sharing(Wt, Rest_children, 
    Input_CStruct, 
    [Head_children_alternative_deleted | 
    Templist], 
    Final_list, Final_CStruct, localpass).
/* *************************************************/
The following rule extract each child from a pair or 
non-pair having alternate or non-alternate form only 
if Match boolean of that child is 'notmatched'
this extracted child is going to be the Child2 among 
Child and Child2 and hence it is to the right of 
Child in the list of node-structure. So a match 
between Child and Child2 is possible only if the 
Child2 is not already matched to a node which is 
to the right of both Child and Child2.
*******************************************************************************
extract_child(X1 or * or X1, 1):-
    check_for_matched_flag_in_child(X1).
extract_child(X2 * or X2, 2):-
    check_for_matched_flag_in_child(X2).
extract_child(X3 * or X3, 3):-
    check_for_matched_flag_in_child(X3).
extract_child(X4 * or X4, 4):-
    check_for_matched_flag_in_child(X4), !.
extract_child(X1 * or X1, 5):-
    check_for_matched_flag_in_child(X1).
extract_child(* X2, X2, 6):-
    check_for_matched_flag_in_child(X2), !.
extract_child(X1, Xl, 7):-
    check_for_matched_flag_in_child(Xl).

******************************************************************************
The following rule checks if the term is equal to 'notmatched'. (if yes then only the child is extracted
to test if it can be matched to its left child or not.
******************************************************************************
check_for_matched_flag_in_child(_,
    Match_Boolean, _):-
    Match_Boolean \== dummy.

******************************************************************************
The following rule takes each child of a pair or
non-pair (having alternate or non-alternate form) and tries to find match(es) for it with the first child
found in the rest of the list and then stops. This is recursively continued for every node in the list
******************************************************************************
find_match(_, _, _, [false, false], CStruct,
        CStruct, _) :- !.

find_match(Wt, Children_having_Child, Child,
    [Head_children | Tail_children],
    [New_head_children | Tail_children],
    Input_CStruct, Final_CStruct, Pass):-
    match_found(Wt, Children_having_Child,
        Child, Head_children, New_head_children,
        Input_CStruct, Final_CStruct, Pass), !.

find_match(Wt, Children_having_Child, Child,
    [Head_children | Tail_children],
    [Head_children | New_Tail_children],
    Input_CStruct, Final_CStruct, Pass):-
    find_match(Wt, Children_having_Child,
        Child, Tail_children,
        New_tail_children, Input_CStruct,
        Final_CStruct, Pass).

******************************************************************************
The following rule succeeds if a match is found, and
generates a modified child a match is found if the
test for equivalence is correct. If the test for
equivalence is correct then get the modified child
then get one of the alternate forms.
******************************************************************************
match_found(Wt, Children_having_Child1, Child1,
    Children_having_Child2,
    New_children_with_Child2,
    Input_CStruct,
    Final_CStruct, localpass):-
    extract_child(Children_having_Child2,
        Child2, L_or_R_flag),
    test_for_equivalence(Child1, Child2, E_new,
        D_new),
    try_local_merge(Child1, Child2, Input_CStruct,
        Final_CStruct, local),
test_for_sneak_path(Wt, Children having Child1, Child1, Child2),

get_one_alternate_form(Child1, Child2, E_new, D_new, Children having Child2, L_or_R_flag, New_children_with_Child2).

The following rule check if two nodes are equivalent (i.e. compatible) or not.

/** **************************************************************/
test_for_equivalence( _, _, _, E0, D0, Matched_boolean1, _),
( _, _, _, E1, D1, Matched_boolean2, _),
E_new, D_new):-
Matched_boolean1 \== dummy,
Matched_boolean2 \== dummy,
equivalent(E0, D0, E1, D1, E_new, D_new).

/*****************************************************************
The following rules extract one of the alternate forms based of the Left_or_Right_flag (i.e. 1, 2, 3, 4, 5, 6 and 7)
***************************************************************************/

get_one_alternate_form((Nid0, _, _, _, _, _,_), (Nid1, Level1, Label1, _, _, _, _), E_new, D_new, _ or _ * X3 or _, 1, (Nid1, Level1, Label1, E_new, D_new, matched, Nid0) * X3).

get_one_alternate_form((Nid0, _, _, _, _, _,_), (Nid1, Level1, Label1, _, _, _, _), E_new, D_new, _ or _ * or X4, 2, (Nid1, Level1, Label1, E_new, D_new, matched, Nid0) * X4).

get_one_alternate_form((Nid0, _, _, _, _, _,_), (Nid1, Level1, Label1, _, _, _, _), E_new, D_new, X1 or _ * or _, 3, X1 * (Nid1, Level1, Label1, E_new, D_new, matched, Nid0)).

get_one_alternate_form((Nid0, _, _, _, _, _,_), (Nid1, Level1, Label1, _, _, _, _), E_new, D_new, _ or X2 * or _, 4, X2 * (Nid1, Level1, Label1, E_new, D_new, matched, Nid0)).

get_one_alternate_form((Nid0, _, _, _, _, _,_), (Nid1, Level1, Label1, _, _, _, _), E_new, D_new, _ * X2, 5, (Nid1, Level1, Label1, E_new, D_new, matched, Nid0) * X2).

get_one_alternate_form((Nid0, _, _, _, _, _,_), (Nid1, Level1, Label1, _, _, _, _), E_new, D_new, X1 * _, 6, X1 * (Nid1, Level1, Label1, E_new, D_new, matched, Nid0)).
get_one_alternate_form({Nid0, _, _, _, _, _, _},
    {Nid1, Level1, Label1, _, _, _, _, _}, E_new, D_new,
    _, 7, {Nid1, Level1, Label1, E_new, D_new,
            matched, Nid0}).

/*******************************************************************************/

The following rules remove one alternate-form from the head children
*******************************************************************************/

build_head_children_list(New_child, _ or _ * X3 or _,
                          1, New_child * X3).

build_head_children_list(New_child, _ or _ * _ or X4,
                          2, New_child * X4).

build_head_children_list(New_child, X1 or _ * _ or _,
                          3, X1 * New_child).

build_head_children_list(New_child, _ or X2 * _ or _,
                          4, X2 * New_child).

build_head_children_list(New_child, _ * X2, 5,
                          New_child * X2).

build_head_children_list(New_child, X1 * _, 6,
                          X1 * New_child).


/*******************************************************************************/

The following rules check if more matches are possible or not: if possible, it is done.
More matches are possible for those pairs whose right child has not yet been matched.
More matches are not considered for non-pairs (i.e. the case of single node) and those
pairs child whose right child has already been matched. The subgoal first_child_nid, in the
rules more_match_possible below is needed in order to retain the Nid of first child which
has been explained before for the fact first_child_nid.
*******************************************************************************/

find_match_for_right_child_if_exists_and_possible(Wt, X1 * Right Child, 1, New rest children,
                                               Final rest children, X1 * Right_child,
                                               Input_CStruct, Final CStruct, Pass):- !,
                                               find_match(Wt, X1 * Right_child,
                                                         Right_child, New rest children,
                                                         Final rest children, Input_CStruct,
                                                         Final_CStruct, Pass).

find_match_for_right_child_if_exists_and_possible(Wt, X2 * Right_child, 2, New rest children,
                                               Final rest children, X2 * Right_child,
                                               Input_CStruct, Final CStruct, Pass):- !,
                                               find_match(Wt, X2 * Right_child,
find_match_for_right_child if_exists_and_possible(Wt, X1 * Right_child, 5, New_rest_children, Final_rest_children, X1 \^ Right_child, Input_CStruct, Final_CStruct, Pass):- !.
find_match_for_right_child if_exists_and_possible(_, Right_child, _, New_rest_children, New_rest_children, Right_child, CStruct, CStruct, _).
/* ***********************************************/
The rules below take the following action:
If no match is found with any child(ren) of the
Head_Children then the left alternative is
arbitrarily selected. For the forms X*Y and for
a single child X, the forms are retained.
/* ***********************************************/
select_alternative_for_head_children(X1 or X3 or _, X1 \^ X3):- !.
select_alternative_for_head_children(X, X).
/* ***********************************************/
Among the following facts, the first two has been
implemented the third fact corresponding to the
nonlocal merge has not been implemented in the
present version.
try_local_merge(Child, Child2, Input_CStruct, Final_CStruct, local).
test_nonlocal_merge(Child, Child2, Input_CStruct, Final_CStruct, nonlocal).
/* ***********************************************/
The following rules check if a sneak-path gets
created or not when any two nodes are merged.
/* ***********************************************/
/* ***********************************************/
This rule ensures that there is no chance of creation
of a sneak-path if the nodes getting merged are not
label-compatible. In the present case, their labels
are -1 and 1 and therefore they are label-
incompatible.
/* ***********************************************/
test_for_sneak_path(_, _, {_, _, -1, _, _, _}, _) :- !.
This rule ensures that there is no chance of creation of a sneak path if the nodes getting merged are not label-compatible. In the present case, their labels are 1 and -1 and therefore they are label-incompatible.

```
test_for_sneak_path(_, _, (_, _, l, _ _, _ _)),
    (_, _, -1, _ _, _ _, _)) :- !.
```

The following rule includes the depth-path which starts from the leaf node and ends in its brother if it has a label-compatible brother (e.g. in XorX'*YorY' X' is a label-compatible brother of Y' if label of X' is 0 while label of Y' is 1).

```
test_for_sneak_path(Wt,
    (Nid0, _, _, _, _, _) or (Nid0, Level0, 0, _, _, _),
    (Nid1, Level1, Label1, _, _, _)) :-
    get_depth_paths(Nid0, Level0, 0),
    (Nid1, Level1, Label1),
    get_depth_paths(Wt, List_of_Depth_Paths), !,
    get_ancestor_paths(Nid1, Level0, Label1),
    List_of_Ancestor_Paths),

    (test_compatible1([[(Nid2, Level2, 1),
                      (Nid0, Level0, 0)]], List_of_Depth_Paths),
    List_of_Ancestor_Paths)).
```

The following is an alternative of the third rule. In case, there is no depth path other then the depth-path related to the label-compatible brother as mention in case of the third rule, then the list of depth-paths has to contain only the depth-path related to Label-compatible brother.

```
test_for_sneak_path(_,
    (Nid0, _, _, _, _, _) or (Nid0, Level0, 0, _, _, _),
    (Nid1, _, Label1, _, _, _)) :-
    get_ancestor_paths(Nid1, Level0, Label1),
    List_of_Ancestor_Paths), !,
    (test_compatible1([[Nid2, Level2, 1],
                      (Nid0, Level0, 0)]], List_of_Ancestor_Paths)).
```

The following rule includes the depth path from the starting leaf node and ends in its brother if it has a label-compatible brother (e.g. in XorX'*YorY' X' is a label-compatible brother of Y' if label of X' is -1 while label of Y' is 0).

```
test_for_sneak_path(Wt, 
  (Nid0, _, _, _, _, _, _ ) or (Nid0, Level0, -1, _, _, _, _ ),
  (Nid2, _, _, _, _, _, _ ) or (Nid2, Level2, 0, _, _, _, _ ),
  (Nid0, Level0, -1, _, _, _, _ ),
  (Nid1, Level1, Label1, _, _, _, _ )) :-
  get_depth_paths((Nid0, Level0, -1),
  (Nid1, Level1, Label1), Wt, List_of_Depth_Paths), !,
  get_ancestor_paths((Nid1, Level0, Label1),
  List_of_Ancestor_Paths), 
  \((test_compatible1(List_of_Depth_Paths, 
  List_of_Ancestor_Paths)).

The following is an alternative of the fifth rule. 
In case, there is no depth path other then the depth path related to the label-compatible brother 
as mentioned in case of fifth rule, then the list of depth paths has to contain only the depth-path related 
to label-compatible brother.

test_for_sneak_path(_, 
  (Nid0, _, _, _, _, _, _ ) or (Nid0, Level0, -1, _, _, _, _ ),
  (Nid2, _, _, _, _, _, _ ) or (Nid2, Level2, 0, _, _, _, _ ),
  (Nid0, Level0, -1, _, _, _, _ ),
  (Nid1, _, Label1, _, _, _, _ )) :-
  get_ancestor_paths((Nid1, Level0, Label1),
  List_of_Ancestor_Paths), !,
  \((test_compatible1(List_of_Depth_Paths, 
  List_of_Ancestor_Paths)).

The following rule is invoked if all the rules 
related test_for_sneak_path above, fails.

test_for_sneak_path(Wt, 
  (Nid0, Level0, Label0, _, _, _, _ ),
  (Nid1, Level1, Label1, _, _, _, _ )) :-
  get_depth_paths((Nid0, Level0, Label0),
  (Nid1, Level1, Label1), Wt, List_of_Depth_Paths), !,
  get_ancestor_paths((Nid1, Level0, Label1),
  List_of_Ancestor_Paths), 
  \((test_compatible1(List_of_Depth_Paths, 
  List_of_Ancestor_Paths)).

The following rule is invoked when there is no chance 
of any creation of sneak-paths when the two nodes 
being considered for merge, are actually merged.
test_for_sneak_path(_, _, _):- true.

The following rule takes two nodes, in the same level, as input and checks if they can be made adjacent in the flippable-nonflippable structure at that level. If possible, the nodes are made adjacent, otherwise this rule fails.

try_local_merge(Child, Child2, Input_CStruct, Final_CStruct, local):-
    Child = (NodeNo1, _, _, _, _, _),
    Child2 = (NodeNo2, _, _, _, _, _),
    check_for_merge( Input_CStruct, Final_CStruct,
                     NodeNo1, NodeNo2, local).

Rule for generating the depth-paths from a node is given below. The concept of depth-path is associated with a node which is currently being considered as one of the possible candidates for merge. A depth-path for such a node is a path that starts from the node, goes in upward direction as far as it can go in the switching-graph generated so far and then come down to the same level as the starting node. Care has been taken so that the path does not end up in the starting node itself. The path is represented by a list of only those nodes (i) which, at the same level, are label-compatible to each other (i.e. if any two nodes in the path are in the same level, then their labels should be either (1,1) or (-1,-1) or (0,1) or (1,0) or (0,1) or (-1,0) or (0,-1) or (0,0) pair) (ii) which exist in the path.

The following rule get_depth_paths generates all the possible depth_paths for the node involved.

get_depth_paths((Nid0, Level0, Label0),
    (Nid1, Level1, Label1), Wt,
    List_of_Depth_Paths):-
    setof(Find_path, Find_path(Level10, Wt, up,
        [(Nid0, Level0, Label0)],
        (Nid1, Level1, Label1), Find_path),
    List_of_Depth_Paths).

Rule for generating the ancestor-paths from a node is given below. The concept of ancestor-path is associated with a node which is currently being considered as one of the possible candidates for merge. An ancestor-path for such a node is a path that starts from the node and goes in upward direction as far as it can go in the switching-graph.
(up to the level = 1 if possible) generated so far.
The ancestor path is represented by a list of nodes
that exist in the path.
The following rule get_ancestor_paths generates
all the possible ancestor_paths for the node
involved.
*******************************************************************************/
get_ancestor_paths((Nid1, Level0, Label1),
   List_of_Anccestor_Paths):-
   setof(Final_list_of_nodes_in_path,
       visit_ancestor((Nid1, Level0, Label1),
         [],
         Final_list_of_nodes_in_path),
       List_of_Anccestor_Paths).
*******************************************************************************/

It is to be noted that while describing the rules
we will refer the node-id of a node as node itself.
*******************************************************************************/
The following rules take an existing flippable-
nonflippable structure, the two nodes which are
being considered for local merge and the type
of merge desired which is, in the present case,
"local" and outputs the modified flippable-
nonflippable structure in the variable
New_flip_structure. If local merge is not possible
then the nonlocal merge should be carried out
(which has not been implemented yet).
*******************************************************************************/
check_for_merge(Old_flip_structure,
   New_flip_structure, NodeNo1, NodeNo2,
   Type_of_merge_desired):-

   check_for_merge1(Old_flip_structure,
                   New_flip_structure, NodeNo1,
                   NodeNo2, Type_of_merge_desired),!.

check_for_merge(Old_flip_structure,
               New_flip_structure, _ , _, nonlocal).
*******************************************************************************/
The following rules are called by the rule
check_for_merge. Each of them in sequence from top
to bottom corresponds to the 4 options below:
1. when the node NodeNo1 is in the left
   sub-structure and node NodeNo2 is in the right
   sub-structure.
2. when the node NodeNo1 is in the right
   sub-structure and node NodeNo2 is in the left
   sub-structure.
3. when both the nodes NodeNo1 and node NodeNo2
   are in the left sub-structure.
4. when both the nodes NodeNo1 and node NodeNo2
   are in the right sub-structure.
*******************************************************************************/
check_for_merge(Old_flip_structure, New_flip_structure, NodeNo1, NodeNo2, Type_of_merge_desired):-
    arg(1, Old_flip_structure, X),
    arg(2, Old_flip_structure, Y),
    appears(NodeNo1, X),
    appears(NodeNo2, Y),
    m(NodeNo1, X, Xnew, Right_flag),
    m(NodeNo2, Y, Ynew, Left_flag), !,
    functor(Old_flip_structure, Functor, 2),
    functor(New_flip_structure, Functor, 2),
    arg(1, New_flip_structure, Xnew),
    arg(2, New_flip_structure, Ynew),
    check_if_type_of_merge_is_acceptable(Right_flag, Left_flag, NodeNo1, NodeNo2, New_flip_structure, Type_of_merge_desired).

check_for_merge(Old_flip_structure, New_flip_structure, NodeNo1, NodeNo2, Type_of_merge_desired):-
    arg(1, Old_flip_structure, X),
    arg(2, Old_flip_structure, Y),
    appears(NodeNo1, Y),
    appears(NodeNo2, X),
    m(NodeNo2, X, Xnew, Right_flag),
    m(NodeNo1, Y, Ynew, Left_flag), !,
    functor(Old_flip_structure, Functor, 2),
    functor(New_flip_structure, Functor, 2),
    arg(1, New_flip_structure, Xnew),
    arg(2, New_flip_structure, Ynew),
    check_if_type_of_merge_is_acceptable(Right_flag, Left_flag, NodeNo1, NodeNo2, New_flip_structure, Type_of_merge_desired).

check_for_merge(Old_flip_structure, New_flip_structure, NodeNo1, NodeNo2, Type_of_merge_desired):-
    arg(1, Old_flip_structure, X),
    arg(2, Old_flip_structure, Y),
    appears(NodeNo1, X),
    appears(NodeNo2, X), !,
    check_for_merge(X, Xnew, NodeNo1, NodeNo2, Type_of_merge_desired),
    functor(Old_flip_structure, Functor, 2),
    functor(New_flip_structure, Functor, 2),
    arg(1, New_flip_structure, Xnew),
    arg(2, New_flip_structure, Y).
check_for_merge(Old_flip_structure, New_flip_structure, NodeNo1, NodeNo2, Type_of_merge_desired):-
arg(1, Old_flip_structure, X),
arg(2, Old_flip_structure, Y),
appears(NodeNo1, Y),
appears(NodeNo2, Y),
check_for_merge(Y, Ynew, NodeNo1, NodeNo2,
    Type_of_merge_desired),
functor(Old_flip_structure, Functor, 2),
functor(New_flip_structure, Functor, 2),
arg(1, New_flip_structure, X),
arg(2, New_flip_structure, Ynew).

************************************************************************
The following rules tries to move a node in the left sub-structure as much to the right as possible.
The format followed by the rules is
mr(Node_id, Structure, New_Structure, Right_flag).
where
1. mr denotes the relation move-right
2. Node_id denote the node which the rules are trying to move to the right
3. Structure is a variable containing the input flippable-nonflippable structure where the node is present
4. New_Structure is a variable that will contain the modified flippable-nonflippable structure after the node has been moved to as much right as possible. If the node has not at all been moved from its initial position in the original flippable-nonflippable structure (which is in the variable Structure), then New_Structure is assigned the flippable-nonflippable structure present in the variable Structure

************************************************************************

mr(Nid, Nid, Nid, true):- !.
% when Nid is in L of f(L,R)
mr(Nid, f(L,R), n(New_R, New_L1), true):-
    appears(Nid, L),
    ml(Nid, L, New_L, true), !,
    flip(R, New_R),
    flip(New_L, New_L1).
% when Nid is in R of f(L,R)
mr(Nid, f(L,R), n(L,New_R), true):-
    appears(Nid, R),
    mr(Nid, R, New_R, true), !.

************************************************************************
The following rule is called when Nid is in L but cannot be moved to extreme right.
In this case the Nid is once moved to the left and then once moved to the right. The two modified flippable-nonflippable structures obtained from the initial flippable-nonflippable structure because of these operations, are compared. Since the initial
structure is flippable, hence out of these two
modified structures, whichever is found to contain
the Nid at the maximum right position, is retained
and the other modified structure is rejected.
********************************************************************
\mr(Nid, f(L,R), New_structure, false):-
appears(Nid, L),
ml(Nid, L, New_L1, _),
  mr(Nid, L, New_L2, _),
  flip(R, New_R),
  flip(New_L1, New_L11),
check_which_is_better_right(Nid, n(New_R, New_L11),
n(New_L2, R),
New_structure), !.
********************************************************************
The following rule is called when Nid is in R but
cannot be moved to extreme right.
In this case the Nid is once moved to the left
and then once moved to the right. The two modified
flippable-nonflippable structures obtained from
the initial flippable-nonflippable structure because
of these operations, are compared. Since the initial
structure is flippable, hence out of these two
modified structures, whichever is found to contain
the Nid at the maximum right position, is retained
and the other modified structure is rejected.
********************************************************************
\mr(Nid, f(L,R), New_structure, false):-
appears(Nid, R),
mr(Nid, R, New_R1, _),
ml(Nid, R, New_R2, _),
  flip(New_R2, New_R22),
flip(L, New_L),
check_which_is_better_right(Nid, n(L, New_R1),
\n(New_R22, New_L),
\n(New_structure), !.

% when Nid is in R of structure n(L,R) then first
% it is tried to move it to the extreme right of R
\mr(Nid, n(L,R), n(L,New_R), true):-
appears(Nid, R),
mr(Nid, R, New_R, true), !.

% when Nid is in R of structure n(L,R) but cannot be
% moved to the extreme right of R
\mr(Nid, n(L,R), n(L,New_R), false):-
appears(Nid, R),
mr(Nid, R, New_R, _), !.

% when Nid is in L of n(L,R) then it is tried
% to move Nid as much as possible to the right in L
\mr(Nid, n(L,R), n(New_L,R), false):-
appears(Nid, L),
mr(Nid, L, New_L, _).
The following rules try to move a node in a flippable-nonflippable structure as much to the left as possible.

The format followed by the rules is
ml(Node_id, Structure, New_Structure, Right_flag).

where
2: Node_id denote the node which the rules are trying to move to the left.
3: Structure is a variable containing the input flippable-nonflippable structure where the node is present.
4: New_Structure is a variable that will contain the modified flippable-nonflippable structure after the node has been moved to as much left as possible. If the node has not at all been moved from its initial position in the original flippable-nonflippable structure (which is in the variable Structure), then New_Structure is assigned the flippable-nonflippable structure present in the variable Structure.

ml(Node_id, Structure, New_Structure, Right_flag).

The following rule is called when Node_id is in R but cannot be moved to extreme left.

In this case the Node_id is once moved to the left and then once moved to the right. The two modified flippable-nonflippable structures obtained from the initial flippable-nonflippable structure because of these operations, are compared. Since the initial structure is flippable, hence out of these two modified structures, whichever is found to contain the Node_id at the maximum left position, is retained and the other modified structure is rejected.

ml(Node_id, Structure, New_Structure, false):=
appears(Node_id, Structure),
mr(Node_id, Structure, New_Structure),
ml(Node_id, Structure, New_Structure, false).
flip(New_R1, New_R1l),
check_which_is_better_left(Nid, n(New_R1l, New_L),
   n(L, New_R2),
   New_structure), !.

/******************************************************************************/
The following rule is called when Nid is in L but
cannot be moved to extreme left.
In this case the Nid is once moved to the left
and then once moved to the right. The two modified
flippable-nonflippable structures obtained from
the initial flippable-nonflippable structure because
of these operations, are compared. Since the initial
structure is flippable, hence out of these two
modified structures, whichever is found to contain
the Nid at the maximum left position, is retained
and the other modified structure is rejected.
/******************************************************************************/
ml(Nid, f(L,R), New_structure, false):-
   appears(Nid, L),
   ml(Nid, L, New_L1, _),
   mr(Nid, L, New_L2, _),
   flip(New_L2, New_L2l),
   flip(R, New_R),
   check_which_is_better_left(Nid,
      n(New_L1, R),
      n(New_R, New_L2l),
      New_structure), !.
% when Nid is in L of structure n(L,R) then at first
% it is tried to move Nid to the extreme left of L
% ml(Nid, n(L,R), n(New_L,R), true):-
%    appears(Nid, L),
%    ml(Nid, L, New_L, true), !.
% when Nid is in structure n(L,R) and is present
% in L but cannot be moved to the extreme left of L
ml(Nid, n(L,R), n(New_L,R), false):-
   appears(Nid, L),
   ml(Nid, L, New_L, _), !.
% when Nid is in R of n(L,R), then it is tried to
% move Nid to the left of R as much as possible.
% Since the structure is non-flippable, so there is
% no question of being able to move Nid to the extreme
% left of the structure and hence Left_flag remains
% "false"
ml(Nid, n(L,R), n(L,New_R), false):-
   appears(Nid, R),
   ml(Nid, R, New_R, _).
%--------------------------------------------------------
check_which_is_better_right(Nid, Struct1, Struct2,
   New_struct):-
   traverse_structure(Struct1, List1/[]),
   traverse_structure(Struct2, List2/[]),
   find_position_of_Nid(Nid, List1, Position1),
   find_position_of_Nid(Nid, List2, Position2),
select_the_better_right(Struct1, Struct2,  
   Position1,  
   Position2, New_struct).
traverse_structure(Integer, [Integer | X]/X):-  
   integer(Integer), !.
traverse_structure(Struct, List/X):-  
   arg1(1, Struct, Arg1),  
   arg2(2, Struct, Arg2),  
   traverse_structure(Arg1, List/X1),  
   traverse_structure(Arg2, X1/X).
% position of Nid in List counting the first  
% element of List as 1.  
find_position_of_Nid(Nid, List, Position):- !,  
   nth1(Position, List, Nid).
% select one of the two structures as new structure  
% depending on the positions  
% Position1 corresponds to Nid's position in Struct1  
% Position2 corresponds to Nid's position in Struct2  
select_the_better_right(Struct1, _, Position1,  
   Position2,Struct1):-  
   Position1 >= Position2, !.
select_the_better_right(_,Struct2,_,_,Struct2).  
%-----------------------------------------------  
check_which_is_better_left(Nid, Struct1, Struct2,  
   New_struct):-  
   traverse_structure(Struct1, List1/[]),  
   traverse_structure(Struct2, List2/[]),  
   find_position_of_Nid(Nid, List1, Position1),  
   find_position_of_Nid(Nid, List2, Position2),  
   select_the_better_left(Struct1, Struct2,  
   Position1, Position2, New_struct).
% select one of the two structures as new structure  
% depending on the positions  
% Position1 corresponds to Nid's position in Struct1  
% Position2 corresponds to Nid's position in Struct2  
select_the_better_left(Struct1, _, Position1,  
   Position2,Struct1):-  
   Position1 =< Position2, !.
select_the_better_left(_,Struct2,_,_,Struct2).  
%-----------------------------------------------  
appears(Nid, Nid):- !.
appears(Nid, Struct):-  
   arg1(1, Struct, Arg1),  
   appears(Nid, Arg1), !.
appears(Nid, Struct):-  
   arg2(2, Struct, Arg2),  
   appears(Nid, Arg2).
/****************************************************************************  
The following rule interchanges the position of the  
sub-structure in a flippable-nonflippable structure  
recursively.  
******************************************************************************/  
flip(Integer, Integer):- integer(Integer), !.

109
flip(Struct, New_Struct):-
    arg(1, Struct, Arg1),
    arg(2, Struct, Arg2),
    flip(Arg1, New_Arg1),
    flip(Arg2, New_Arg2),
    functor(Struct, Functor, 2),
    functor(New_Struct, Functor, 2),
    arg(1, New_Struct, New_Arg2),
    arg(2, New_Struct, New_Arg1).

*******************************************************************************
The following rules checks if the flippable-nonflippable structure (that has been obtained after modifying the left and right sub-structures in order to bring as close as possible the two nodes which are considered for being merged) contains the nodes (which are being considered for local merge are adjacent.
*******************************************************************************

The rule below is invoked if there is no dummy node in between the two nodes to be merged.
*******************************************************************************
check_if_type_of_merge_is_acceptable(true, true, _, _, _, local):- !.
*******************************************************************************
The two rules below are invoked if there is one or more dummy nodes in between the two nodes to be merged. The first rule, in the sequence, is invoked if the position of the Nid1 is to the left of Nid2. The second rule, in the sequence, is invoked if the position of the Nid1 is to the right of Nid2. In the following two rules only '<' and '>' are checked because the '=' case is possible when both left and right flags are true which is already taken care by the above rule.
*******************************************************************************
check_if_type_of_merge_is_acceptable(_, _, Nid1, Nid2, New_structure, local):-
    traverse_structure(New_structure, List/[]),
    find_position_of_Nid(Nid1, List, Position1),
    find_position_of_Nid(Nid2, List, Position2), !,
    Position1 < Position2,
    check_if_all_nids_between_Nid1_and_Nid2_are_dummy(Position1, Position2, List).
check_if_type_of_merge_is_acceptable(_, _, Nid1, Nid2, New_structure, local):-
    traverse_structure(New_structure, List/[]),
    find_position_of_Nid(Nid1, List, Position1),
    find_position_of_Nid(Nid2, List, Position2), !,
    Position1 > Position2,
    check_if_all_nids_between_Nid1_and_Nid2_are_dummy(Position1, Position2,
/* Position1, List. 
 
The following rule is invoked if the nodes cannot be brought adjacent. In that case the merge will be a nonlocal merge. */

check_if_type_of_merge_is_acceptable(_, _, _, _nonlocal).

/* The following check is very important because the compound structure does not tell us whether any node(s), if present between any two merging nodes, is (are) dummy or not. If it (they) is (are) found dummy then the merge would be local merge. In order to keep the flipping-related rules in this file simpler and easy to understand, such a check has been incorporated here separately instead of putting them somewhere inside the other rules in this file. */

check_if_all_nids_between_Nid1_and_Nid2_are_dummy(
    Smaller_Position_No,
    Bigger_Position_No, List):-
    SNo is Smaller_Position_No + 1,
    BNo is Bigger_Position_No - 1,
    test_dummy(SNo, BNo, List).

/* The following rules tests if all the nodes in the list, List, from position SNo to BNo are dummy or not. */

test_dummy(XNo, XNo, List):-
    nth1(XNo, List, Nid), !,
    nid_is_dummy(Nid).

test_dummy(SNo, BNo, List):-
    nth1(SNo, List, Nid), !,
    nid_is_dummy(Nid),
    One_more_than_SNo is SNo + 1,
    test_dummy(One_more_than_SNo, BNo, List).

/* The following rule tests if Nid is the id of a dummy node or not. A node will be dummy if its father is matched to another node to its left (in the list of nodes) which in turn is matched to a node such that Left_node_id1 is not equal to 0. */

nid_is_dummy(Nid):-
    Father_of_nid is Nid // 2,
    node(Father_of_nid, Level, Label, _, _matched, Left_node_id),
    node(Left_node_id, _, _, _, _matched, _, Left_node_id1),
    Left_node_id1 =\= 0.
The rule find_path is invoked when any two nodes become possible candidate for a merge. It checks for the existence of any sneak-path that would get created if those nodes are merged.
find_path takes in the starting level (that the level of the nodes which are possible candidates for a merge), the weight of the previous level, the starting direction (i.e. up), a list containing one of the two nodes (which we shall henceforth call the starting-node. The other node will be called the ending-node. As the execution of the program proceeds, the list will gradually build up containing all the nodes in one depth-path (the term depth-path has been explained before) beginning with the starting-node, and ending with the ending-node.
A sample input to the find_path rule would be as follows:
```
find_path(5, 4, up, [(42, 5, -1)], (42, 5, -1),
(60, 5, -1), Find_path).
```
where a node is represented by (Nid, Level, Label)
Before describing the rules below, we need to assume the followings.
Let us assume that a point-charge is traversing down a path in the switching-graph and it has just crossed one node, A, which has already been merged with another node, B. Also we assume that A and B has two children C and D. The point-charge has now some possible alternative paths to traverse:
1. It can go up and cross its matched node B depending on B's label.
2. It can go down and cross its child C depending on C's label.
3. It can go down and cross its child D depending on D's label.
Let us now assume that the point-charge is traversing up a path in the same switching-graph mentioned above. Suppose it has crossed the node C. The point-charge has now some possible alternative paths to traverse:
1. It can go down and cross its brother D depending on D's label.
2. It can go up and cross its father A depending on A's label.
3. It can go up and cross its father B depending on B's label.
find_path:-
find_next(down, (Nid1, Level, Label),
Next, up),
new_when_direction_is_down_up(Next, -
[[(Nid1, Level, Label) | Tail]],
compatible(Next, [(Nid1, Level, Label) | Tail]),
find_path(Start_Level, Wt, up,
[Next, (Nid1, Level, Label) | Tail],
(Nid2, Level of Nid2, Label of Nid2),
(Nid3, Level3, Label3),
Find_path).

/********************************************************************
The following rule is the boundary condition. It succeeds when the a
node at the same level as the starting_node has been reached such that the
new node that has been reached does not have the same
father as the starting_node i.e. either the new node
reached is not the starting node itself or it does
not have a surrogate father.
*********************************************************************/
find_path(Start_Level, Wt, down,
[(Nid1, Level1, Label1) | Tail],
(Nid2, __, Label2), (_, __, _),
[(last nid, Start_Level, Label of last nid)]
[(Nid1, Level1, Label1) | Tail])):-
Level1 is Start_Level - 1,
Parent of Nid2 is Nid2 // 2,
Nid1 =\= Parent of Nid2,
setof(Matched_node,
find_matched_node(
  Parent of Nid2, 
  Matched_node),
  Matched_nodes_to_Parent_of_Nid2),
nonmember(Nid1, 
  Matched_nodes_to_Parent_of_Nid2),
node(Nid1, __, El, __, _),
deduce_label_of_last nid(El, Wt, 
  Label2, 
  Label of last nid), !.

/********************************************************************
The following rule is invoked when setof() in above
rule fails.
*********************************************************************/
find_path(Start_Level, Wt, down,
[(Nid1, Level1, Label1) | Tail],
(Nid2, __, Label2), (_, __, _),

113
[(lastnid, Start_Level, Label_of_lastnid),
  [(Nid1, Level1, Label1), Tail])]] :-
  Level1 is Start_Level - 1,
  Parent_of_Nid2 Is Nid2 // 2,
  Nid1 =\= Parent_of_Nid2,
  node(Nid1, __, El, __, __),
  deduce_label_of_lastnid(El, Wt,
    Label2,
    Label_of_lastnid), !.

/***********************************************************************/
find_path(Start_Level, Wt, Direction, [Head | Tail],
          (Nid2, Level_of_Nid2, Label_of_Nid2),
          (Nid3, Level3, Label3), Find_path):-
  find_next(Direction, Head, Next,
    New_Direction),
  new(Next, [Head | Tail]),
  compatible(Next, [Head | Tail]),
  find_path(Start_Level, Wt, New_Direction,
            [Next, Head | Tail], (Nid2,
                 Level_of_Nid2, Label_of_Nid2),
            (Nid3, Level3, Label3), Find_path).

/***********************************************************************/
The following rules find out the label of the last
node reached in the depth-path, whose level is same
as the starting node’s level. The level is
determined by the binary weight, Wt, corresponding
to the level.

**************************************************************************
**deduce_label_of_lastnid(El, Wt, -1, -1) :- **
**  reduced_function(El, Wt, El_new_left,
**              El_new_right),
**  El_new_left =\= [],
**  El_new_right =\= [], !.
**deduce_label_of_lastnid(El, Wt, 0, -1) :-
**  reduced_function(El, Wt, El_new_left,
**              El_new_right),
**  El_new_left =\= [],
**  El_new_right =\= [], !.
**deduce_label_of_lastnid(El, Wt, 1, 1) :-
**  reduced_function(El, Wt, El_new_left,
**              El_new_right),
**  El_new_left =\= [],
**  El_new_right =\= [], !.
**deduce_label_of_lastnid(El, Wt, 0, 1) :-
**  reduced_function(El, Wt, El_new_left,
**              El_new_right),
**  El_new_left =\= [],
**  El_new_right =\= [], !.

114
El_new_left == [],
El_new_right \== [], !.

deduce_label_of_lastnid(El, Wt, _, 0):-
    reduced_function(El, Wt, El_new_left, El_new_right).
    El_new_left \== [],
    El_new_right \== [].

/****************************
The following rule tries to find the next node in
the path while
1. coming upwards and then going to downward
direction.
2. coming downwards and then going to downward
direction.
3. coming upwards and then going to upward
direction.
4. coming downwards and then going to upward
direction.

find_next(up, (Nid, Level, _),
    (Brother_of_Nid, Level, Label_of_brother),
    down):-
    brother_of(Nid, Brother_of_Nid),
    find_label(Brother_of_Nid, Label_of_brother).
find_next(down, (Nid, Level, _),
    (Child_of_Nid, Level_of_Child,
    Label_of_Child), down):-
    child_of(Nid, Child_of_Nid),
    Level_of_Child is Level + 1,
find_next(up, (Nid, Level, _),
    (Father_of_Nid, Level_of_Father,
    Label_of_Father), up):-
    father_of(Nid, Father_of_Nid),
    Level_of_Father is Level - 1,
    find_label(Father_of_Nid, Label_of_Father).
find_next(down, (Nid, Level, _),
    (Matched_node_to_Nid, Level,
    Label_of_Matched_node_to_Nid), up):-
    match_of(Nid, Matched_node_to_Nid),
    find_label(Matched_node_to_Nid,
    Label_of_Matched_node_to_Nid).

/****************************
The following rule tries to find the id of a node
which is matched to the id of another node termed
as Nid in the rule

match_of(Nid, Matched_node_to_Nid):-

115
find_matched_node(Nid, Matched_node_to_Nid).

The following rules try to find the id of a node which is matched to the id of another node termed as Nid.
1. If Nid is the leftmost node then its tried to find out if Nid is connected to any node at its own level and if such a matched node is found then it is verified that the matched node is not dummy.
2. If Nid is not the leftmost node then its tried to find out the leftmost node matched to Nid at Nid’s own level and if such a matched node exists then it is verified that the matched node is not dummy.
3. If Nid is not the leftmost node then its tried to find out the leftmost node matched to Nid at Nid’s own level. Next any other node which is not equal to Nid but matched to the leftmost node is found out if it exists and then it is verified that the new node is not dummy.

find_matched_node(Nid, Matched_node_to_Nid):-
    leftmost_node(Nid),
    connected(Nid, Matched_node_to_Nid),
    node_exists_and_not_dummy(Matched_node_to_Nid).

find_matched_node(Nid, Matched_node_to_Nid):-
    get_leftmost_node(Nid, Matched_node_to_Nid),
    node_exists_and_not_dummy(Matched_node_to_Nid).

find_matched_node(Nid, Matched_node_to_Nid):-
    get_leftmost_node(Nid, Leftmost_Nid),
    connected(Leftmost_Nid, Matched_node_to_Nid),
    Nid \= \~ Matched_node_to_Nid,
    node_exists_and_not_dummy(Matched_node_to_Nid).

The following rules try to find the father of a node, say X

father_of(X, Father_of_X):-
    find_father(X, Father_of_X).

father_of(X, Father_of_X):-
    find_surrogate_father(X, Father_of_X).

find_father(X, Father_of_X):-
    even(X),
    Father_of_X is X // 2.

find_father(X, Father_of_X):-
    odd(X),
    Father_of_X is (X-1) // 2.

The following rules try to find the surrogate father of a node, say X

find_surrogate_father(X, Surrogate_Father_of_X):-
    find_father(X, Father_of_X),
    leftmost_node(Father_of_X),
    connected(Father_of_X, Surrogate_Father_of_X).
find_surrogate_father(X, Surrogate_Father_of_X) :-
    find_father(X, Father_of_X),
    get_leftmost_node(Father_of_X, Surrogate_Father_of_X).
find_surrogate_father(X, Surrogate_Father_of_X) :-
    find_father(X, Father_of_X),
    get_leftmost_node(Father_of_X,
                     Leftmost_Father_of_X),
    connected(Leftmost_Father_of_X, Surrogate_Father_of_X),
    Surrogate_Father_of_X =\_ Father_of_X.

/********************
Following rules take care of nodes to the right of
the real father where the real father is \_  
********************/

connected(Z,Y) :- node(Y, _, _, _, _, matched, Z).
connected(Z,Y) :- node(Y, _, _, _, _, matched, X),
                 connected(Z,X).

/********************
The following rules try to find the brother of a 
node, say Nid  
********************/

brother_of(Nid, Brother_of_Nid) :-
    find_brother(Nid, Brother_of_Nid).
brother_of(Nid, Brother_of_Nid) :-
    find_matched_brother(Nid, Brother_of_Nid).

/********************
The following rules try to find the brother of a 
node, say X, when the siblings have been generated 
from one father only  
********************/

find_brother(X, Brother_of_X) :- odd(X),
                                Brother_of_X is X - 1,
                                node_exists_and_not_dummy(Brother_of_X).
find_brother(X, Brother_of_X) :- even(X),
                                Brother_of_X is X + 1,
                                node_exists_and_not_dummy(Brother_of_X).

/********************
The following rules try to find the brother of a 
node, say X, when the siblings have been generated 
from more than one merged father  
********************/

find_matched_brother(X, Brother_of_X) :-
    find_surrogate_father(X, Father_of_X),
    Brother_of_X is Father_of_X * 2,
    Brother_of_X =\_ X,
    node_exists_and_not_dummy(Brother_of_X).
find_matched_brother(X, Brother_of_X) :-
    find_surrogate_father(X, Father_of_X),
    Brother_of_X is (Father_of_X * 2) + 1,
    Brother_of_X =\_ X,
    node_exists_and_not_dummy(Brother_of_X).
The following rules try to find the child of a node, say Nid.

child_of(Nid, Child_of_Nid):-
    find_child(Nid, Child_of_Nid).
child_of(Nid, Child_of_Nid):-
    find_matched_child(Nid, Child_of_Nid).

The following rules try to find the child of a node, say X where the child has been generated by the single father.

find_child(X, Child_of_X):-
    Child_of_X is X * 2,
    node_exists_and_not_dummy(Child_of_X).
find_child(X, Child_of_X):-
    Child_of_X is (X * 2) + 1,
    node_exists_and_not_dummy(Child_of_X).

The following rules try to find the child of a node, say X where the child has been generated by the more than one father.

find_matched_child(Nid, Matched_Child_of_Nid):-
    leftmost_node(Nid),
    connected(Nid, Node_connected_to_Nid),
    Matched_Child_of_Nid is Node_connected_to_Nid * 2,
    node_exists_and_not_dummy(Matched_Child_of_Nid).
find_matched_child(Nid, Matched_Child_of_Nid):-
    leftmost_node(Nid),
    connected(Nid, Node_connected_to_Nid),
    Matched_Child_of_Nid is (Node_connected_to_Nid * 2) + 1,
    node_exists_and_not_dummy(Matched_Child_of_Nid).
find_matched_child(Nid, Child_of_Nid):-
    get_leftmost_node(Nid, Leftmost_Nid),
    Child_of_Nid is Leftmost_Nid * 2,
    node_exists_and_not_dummy(Child_of_Nid).
find_matched_child(Nid, Child_of_Nid):-
    get_leftmost_node(Nid, Leftmost_Nid),
    Child_of_Nid is (Leftmost_Nid * 2) + 1,
    node_exists_and_not_dummy(Child_of_Nid).
find_matched_child(Nid, Matched_Child_of_Nid):-
    get_leftmost_node(Nid, Leftmost_Nid),
    connected(Leftmost_Nid, Matched_Nid),
    Nid =\= Matched_Nid,
    Matched_Child_of_Nid is Matched_Nid * 2,
    node_exists_and_not_dummy(Matched_Child_of_Nid).
find_matched_child(Nid, Matched_Child_of_Nid):-
get_leftmost_node(Nid, Leftmost_Nid),
connected(Leftmost_Nid, Matched_Nid),
Nid \neq Matched_Nid,
Matched_Child_of_Nid is
(Matched_Nid * 2) + 1,
node_exists_and_not_dummy(Matched_Child_of_Nid).

The following rule tries to find that the new node,
which is being considered for inclusion in the path (that is being traversed when the direction of
traversal is first downwards and then upwards), is a node which exists in the node-database, is
not a dummy node and has not yet been included in the depth-path.

new_when_direction_is_down_up((Nid_of_Next, Level, _),
[(Nid_of_Head, Level, Label) | Tail]):-

node_exists_and_not_dummy(Nid_of_Next),
nomember(Nid_of_Next, [(Nid_of_Head,
Level, Label) | Tail]).

The following rule tries to find a new node in the
path being traversed when the direction of
traversal is any combination of up and down
directions except the combination discussed for
the previous rule.

new(_, []).

new((Nid_of_Next, Level, _),
[(Nid_of_Head, Level, _) | Tail]):- !,

node_exists_and_not_dummy(Nid_of_Next),
find_matched_nodes(Nid_of_Head,
List_of_Matched_Nids_to_Head_including_Head),
nomember(Nid_of_Next,
List_of_Matched_Nids_to_Head_including_Head),
new((Nid_of_Next, Level, _), Tail).

new((Nid_of_Next, Level_of_Next, _),
[(_, _, _) | Tail]):-
new((Nid_of_Next, Level_of_Next, _), Tail).

The following rule tries to find if the node being
considered for inclusion in the depth-path, is
label-compatible with any node in the path
whose level is equal to the level of the node being
considered.
Any two nodes A and B are label-compatible if
1. label of A is 1, label of B is 1
2. label of A is 1, label of B is 0
3. label of A is 0, label of B is 1
4. label of A is -1, label of B is 0
5. label of A is 0, label of B is -1
6. label of A is -1, label of B is -1

compatible([], []). compatible([Nid_of_Next, Level, Label_of_Next], _). compatible([Nid_of_Next, Level, Label_of_Next], [T]).


check_compatibility((Nid_of_Next, Level, Label_of_Next), [T]).

The following rule tries carries out the compatibility checking.

check_compatibility(-1, -1).
check_compatibility(1, 1).
check_compatibility(0, _):- !.
check_compatibility(_0, 0).

The following rules verify that a node, say Nid, exists and it is not a dummy node.

node_exists_and_not_dummy(Nid):- node(Nid, _, _, _, matched, _).
node_exists_and_not_dummy(Nid):- node(Nid, _, _, _, notmatched, _).

The following rule finds the label of a node, say Nid.

find_label(Nid, Label_of_Nid):- node(Nid, _, Label_of_Nid, _, _, _).

The following rule tries to find all the nodes which are matched to the node Nid Head and puts them in a list together with the Nid Head.

find_matched_nodes(Nid_Head, _[Nid_Head | List_of_Matched_Nids]):- leftmost_node(Nid_Head), !,-
get_matched_nodes(Nid_Head, List_of_Matched_Nids).
find_matched_nodes(Nid_Head, _[Leftmost_Nid | List_of_Matched_Nids]):- get_leftmost_node(Nid_Head, Leftmost_Nid),
get_matched_nodes(Leftmost_Nid, List_of_Matched_Nids).

The following rule finds the node that is matched to Nid Head and is in the left of Nid Head.

leftmost_node(Nid_Head):-

120
node(Node_Head, __/_/_, _, notmatched, 0).

The following rule tries to find the leftmost node in a group of matched nodes if the node matched to left of Node_Head is not to the immediate left of Node_Head.

******************************************************************************
get_leftmost_node(Node_Head, Leftmost_Nid):-
    node(Node_Head, __/_/_, _, matched, Leftmost_Nid),
    node(Leftmost_Nid, __/_/_, _, notmatched, 0), !.
get_leftmost_node(Node_Head, Leftmost_Nid):-
    node(Node_Head, __/_/_, _, matched, X),
    get_leftmost_node(X, Leftmost_Nid).
******************************************************************************

The following rule tries to find all the nodes which are matched to the node Last_Left_Nid and puts them in a list.

******************************************************************************
get_matched_nodes(Last_Left_Nid, [H | T]):-
    node(H, __/_/_, _, matched, Last_Left_Nid),
    get_matched_nodes(H, T), T.
******************************************************************************

get_matched_nodes(_, []).  
******************************************************************************

The following rule checks if X is odd or even
******************************************************************************
odd(X) :- Y is X mod 2, Y =\= 0.
even(X) :- Y is X mod 2, Y =:: 0.
******************************************************************************

The following rules check if any of the ancestor-path from one of two nodes which are being considered for the local merge is label-compatible with any of the depth-paths starting from the other node. Here the label compatibility of the paths means that all the nodes, belonging to both ancestor-path and depth-path, at every level in the switching-graph, have labels either all 1's or all -1's or a combination of 0 and 1 or a combination of 0 and -1.
The format of the rule is

******************************************************************************
test_compatible1(List_of_Depth_Paths, List_of_Anccestor_Paths)
******************************************************************************
test_compatible1(List_of_Depth_Paths, [H | _]):-
    test_compatible2(
        List_of_Depth_Paths, H), !.
test_compatible1(List_of_Depth_Paths, [ _ | T]):-
    test_compatible1(List_of_Depth_Paths, T).
******************************************************************************

The format of the following rule is

******************************************************************************
test_compatible2(List_of_Depth_Paths, Ancestor_Path)
******************************************************************************
test_compatible2([H | _], One_Ancestor_Path):-
    test_compatible3(H, One_Ancestor_Path).

The following rule takes just one ancestor-path and one depth-path and tests their compatibility.
The format of the rule is:
    test_compatible3(One_Depth_Path, One_Ancestor_Path).

 ******************************************************/
    test_compatible3([], _).
    test_compatible3([( _, Level, Label ) | T], One_Ancestor_Path):-
        member1(_, Level, Label),
        One_Ancestor_Path,
        test_compatible3(T, One_Ancestor_Path).

******************************************************/
    member1(_, Level, 1), [( _, Level, 1 ) | _]):- !.
    member1(_, Level, -1), [( _, Level, -1 ) | _]):- !.
    member1(_, Level, _), [( _, Level, 0 ) | _]):- !.
    member1(_, Level, _), [( _, Level, _ ) | _]):- !.

    member1(Node, [ _ | T ]):-
        member1(Node, T).

The following rules create an ancestor-path.
It takes one of the nodes considered for being merged and an empty list as inputs and generates a list of nodes that forms one ancestor-path.
******************************************************/
    visit_ancestor([1,0,0], X, X):- !.
    visit_ancestor(Current_Head, List_of_nodes_in_path, Final_list_of_nodes_in_path):-
        Current_Head = (NId1, _, _),
        father_of(NId1, Father_of_NId1),
        node(Father_of_NId1, Level_of_father, Label_of_father,
             List_of_nodes_in_path, Final_list_of_nodes_in_path, [Current_Head] | List_of_nodes_in_path),
        % Visit the next ancestor
        visit_ancestor([Father_of_NId1, _'-'_'-'], Level_of_father, Label_of_father),
        [Current_Head | List_of_nodes_in_path],
        Final_list_of_nodes_in_path).
Appendix D
PROLOG CODE FOR TOPOLOGICAL LAYOUT GENERATOR

:- use_module(library(basics)).
:- use_module(library(lists)).

The following rules are used to reverse a list.

//rev([], X/X).
d_rev([X|Y], Z/2Z):- d_rev(Y, Z/2Z).

:- dynamic blank/3, wire/3, red/3, green/3, cr/3.

The following rule traverses the flippable-nonflippable structure from left to right and collects the node-ids present in the structure in a list in the same sequence in which they appear in the structure.

traverse(Struct, ListIn, [Struct|ListIn]) :-
    integer(Struct), !.

traverse(Struct, ListIn, ListOut) :-
    arg(2, Struct, Arg2),
    traverse(Arg2, ListIn, Templist),
    arg(1, Struct, Arg1),
    traverse(Arg1, Templist, ListOut).

The following rule generates say, x number of lists where x is equal to the number of node-ids present in the flippable-nonflippable structure mentioned above. Each such list corresponds to a particular column of the topological matrix to be formed and each element of a list belongs to a particular row of that matrix. All x number of lists are generated in a list as a list (FinalList) of lists.

gen_row_col([], ListIn, _ListIn) :- !.
gen_row_col([Head|Tail], ListIn, NodeList, FinalList) :-
    process_col(Head, [], NodeList, CurrentList, NewNodeList),
    gen_row_col(Tail, [CurrentList|ListIn], NewNodeList, FinalList).

process_col(Node, X, NodeList, [Node|NodeList]) :-
    FatherNode is Node/2,
    FatherNode = 1, !.
    process_col(Node, X, NodeList, Final_col_list, [Node|NodeList]) :-

123
FatherNode is Node//2,
member(FatherNode, NodeList), !,
fill_with_blanks(FatherNode, [Node|X],
                Final_column_list).
process_col(Node, Current_col, NodeList,
            CurrentList, NewNodeList) :-
    FatherNode is Node//2,
    process_col(FatherNode, [Node|Current_col],
                [Node|NodeList], CurrentList, NewNodeList).
fill_with_blanks(1, Node_list, Node_list) :- !.
fill_with_blanks(Node, NodeList, New_node_list) :-
    FatherNode is Node//2,
    fill_with_blanks(FatherNode, [b|NodeList],
                     New_node_list).
/* ***********************************************
The following rules take the list of lists
generated by gen_row_col rule in the reversed
order and generates the elements (i.e. operators)
of the matrix
************************************************/
determine_elements(Col_count, []) :-
    num_rows(Row_count),
    generate_cr(Col_count, Row_count),
    Last_col is Col_count -1,
    gen_blanks_odd_rows(Last_col,
                        Row_count, 1,1).

determine_elements(Col_count,
                   [Head_a_list_of_nodes|
                    Remaining_list_of_lists_of_node]) :-
    assert_elements(Col_count, 2,
                    Head_a_list_of_nodes),
    New_col_count is Col_count+1,
    determine_elements(New_col_count,
                       Remaining_list_of_lists_of_node).
/* ***********************************************
The following rules generates a carriage return
for the end of every row.
************************************************/
generate_cr(Col, Row) :-
generate_cr(Col, 1, Row).
generate_cr(Col, Current_row, Final_row) :-
    Current_row =< Final_row, !,
    assert(Cr(cr, Current_row, Col)),
    New_row is Current_row+1,
    generate_cr(Col, New_row, Final_row).

generate_cr(_, _, _).
/* ***********************************************
The following rules generates a carriage return
for the end of each row
************************************************/
gen_blanks_odd_rows(Col_max, Row_max,
                    Col_Current, Row_current) :-
Col_current <= Col_max, !,
assert(blank(b, Row_current, Col_current)),
Col_new is Col_current+1,
gen_blanks_odd_rows(Col_max, Row_max,
        Col_new, Row_current).
gen_blanks_odd_rows(Col_max, Row_max,
        _, Row_current) :-
Row_current < Row_max, !,
Row_new is Row_current+2,
gen_blanks_odd_rows(Col_max, Row_max,
        _, Row_new).
gen_blanks_odd_rows(_, _, _, _).
/**-------------------------------------------------------------------------------------**/
The following rules generates operators for every even row of the matrix.
/**-------------------------------------------------------------------------------------**/
assert_elements(_, _, []).
assert_elements(Col_count, Row_count,
        [Head_node| Remaining_list_of_nodes]) :-
process_a_node(Col_count, Row_count, Head_node),
        New_row_count is Row_count+2,
        assert_elements(Col_count, New_row_count,
        Remaining_list_of_nodes).

process_a_node(Col, Row, b) :-
assert(blank(b, Row, Col)), !.
process_a_node(Col, Row, X) :-
        node(X, _, 'R', 'D', dummy, _),
        assert(Blank(b, Row, Col)).
process_a_node(Col, Row, X) :-
        node(X, _, 0, 'R', 'D', _),
        assert(wire(X, Row, Col)).
process_a_node(Col, Row, X) :-
        node(X, _, -1, 'R', 'D', _),
        assert(red(X, Row, Col)).
process_a_node(Col, Row, X) :-
        node(X, _, 1, 'R', 'D', _),
        assert(green(X, Row, Col)).

find_position(N, Row, Col) :-
        red(N, Row, Col), !.
find_position(N, Row, Col) :-
        green(N, Row, Col), !.
find_position(N, Row, Col) :-
        wire(N, Row, Col).
find_bigger(Colx, Coly, Colx, Col2) :-
        Colx < Coly,
        Col2 is Coly - 1, !.
find_bigger(Colx, Coly, Coly, Col2) :-
        Col2 is Colx - 1.
matched(Row1, Col1, Col2) :-
        find_list_of_matched_nodes(Row, List_of_node_cols),
        find_ends(List_of_node_cols, Col1, Col1),
        Row1 is Row + 1,
        Col2 is Col1 - 1,
retract_blank_assert_wire(Row, Coll, Col2).
find_list_of_matched_nodes(Row, List_of_node_cols) :-
  node(N1, _, _, _, _, matched, _),
  \+ node(N1, _, _, _, _, notmatched, _),
  find_list(N1, [Row, List_of_node_cols]).
find_list(N1, List, Row, [[N1, Col] | List]) :-
  node(N1, _, _, _, _, notmatched, 0), !,
  find_position(N1, Row, Col).
find_list(N1, List, Row, Final) :-
  node(N1, _, _, _, _, matched, N2),
  find_position(N1, Row, Col),
  find_list(N2, [[N1, Col] | List], Row, Final).
brother(Row1, Colbegin, Colend) :-
  node(N1, _, _, _, _, X, _),
  X \= dummy,
  even(N1),
  N2 is N1 + 1,
  node(N2, _, _, _, _, _, _),
  find_position(N1, Row, Colx),
  find_position(N2, Row, Coly),
  find_bigger(Colx, Coly, Colbegin, Colend),
  Row1 is Row - 1,
  retract_blank_assert_wire(Row1, Colbegin, Colend).

even(X) :- X1 is X//2, X2 is X1*2, X \= X2.
retract_blank_assert_wire(Row, Coll, Col2) :-
  Coll \= Col2,
  retract(blank(b, Row, Coll)),
  assert(wire(oddrow, Row, Coll)),
  Newcoll is Coll + 1,
  retract_blank_assert_wire(Row, Newcoll, Col2), !.

retract_blank_assert_wire(_, _, _).
find_ends({[Node, Col] | Tail}, Min_col, Max_Col) :-
  find_ends({[Node, Col] | Tail}, Col, Col, Min_col, Max_Col).
find_ends([], Colmin, Colmax, Colmin, Colmax) :- !.
find_ends([(_, Col) | Tail], Colmin_so_far, Colmax_so_far, Colmin, Colmax) :-
  Col < Colmin_so_far, !,
  find_ends(Tail, Col, Colmax_so_far, Colmin, Colmax).
find_ends([(_, Col) | Tail], Colmin_so_far, Colmax_so_far, Colmin, Colmax) :-
  Col > Colmax_so_far, !,
  find_ends(Tail, Colmin_so_far, Col, Colmin, Colmax).
find_ends([(_, _) | Tail], Colmin_so_far, Colmax_so_far, Colmin, Colmax) :-
  find_ends(Tail, Colmin_so_far, Colmax_so_far, Colmin, Colmax).
d_q_sort([], X/X).

d_q_sort([Pivot|Rest], X1/X2) :-
    separate(Pivot, Rest, [], [], Bigger, Smaller),
    d_q_sort(Smaller, X1/[Pivot|X4]),
    d_q_sort(Bigger, X4/X2).

separate([], [], X,Y,X,Y).

d_q_sort([Head|Tail], Bigin, Smallin, Bigout, Smallout) :-
    less(Head, Pivot), !,
    separate(Pivot, Tail, Bigin, [Head|Smallin], Bigout, Smallout).

separate(Pivot, [Head|Tail], Bigin, Smallin, Bigout, Smallout) :-
    separate(Pivot, Tail, [Head|Bigin], Bigout, Smallin, Smallout).

less([X1,Y1], [X2,Y2]) :- X1 =< X2, Y1 =< Y2.

less([X1, _], [X2, _]) :- X1 < X2.

output(Sorted_list) :- findall((X,Y,Z),
    retrieve_all(X,Y,Z), S),
    d_q_sort(S, Sorted_list/[]).

******************************************************************************

The following rules take the sorted list of triplets
(w,Row,Column), (b,Row,Column), (g,Row,Column) and (c,Row,Column) and outputs
the matrix (in a file called topological_matrix) in
such a way that each line in the output file
contains maximum 40 characters. The end of a row of
the matrix is determined by writing two characters
"W#" after the last column of the matrix for every
row.

******************************************************************************

process_sorted_list([(_,Row,_) | _, _]) :-
    num_rows(Row_count), Row is Row_count + 1, !.

process_sorted_list([w,_,_] | Rest], Count) :-
    NewCount is Count + 1,
    NewCount > 40, !,
    write('W#'),
    nl,
    process_sorted_list(Rest, 1).

process_sorted_list([w,_,_] | Rest], Count) :-
    NewCount is Count + 1,
    process_sorted_list(Rest, NewCount), !.

process_sorted_list([b,_,_] | Rest], Count) :-
    NewCount is Count + 1,
    NewCount > 40, !,
    write('b#'),
    nl,
    process_sorted_list(Rest, 1).

process_sorted_list([b,_,_] | Rest], Count) :-
    NewCount is Count + 1,
    process_sorted_list(Rest, NewCount), !.

process_sorted_list([r,_,_] | Rest], Count) :-
    NewCount is Count + 1,
NewCount is Count + 1,
NewCount > 40, !,
write('R#'),
nl,
process_sorted_list(Rest, 1).

process_sorted_list([r,_,_ | Rest], Count) :-
write('R#'),
NewCount is Count + 1,
process_sorted_list(Rest, NewCount), !.

process_sorted_list([g,_,_ | Rest], Count) :-
NewCount is Count + 1,
NewCount > 40, !,
write('G#'),
nl,
process_sorted_list(Rest, 1).

process_sorted_list([g,_,_ | Rest], Count) :-
write('G#'),
NewCount is Count + 1,
process_sorted_list(Rest, NewCount), !.

process_sorted_list([c,_,_ | Rest], Count) :-
NewCount is Count + 1,
NewCount > 40, !,
write('E#'), /* after every last column put E instead of nl */
nl,
process_sorted_list(Rest, 1).

process_sorted_list([c,_,_ | Rest], Count) :-
write('E#'), /* after every last column put E instead of nl */

NewCount is Count + 1,
process_sorted_list(Rest, NewCount), !.
retrieve_all(w, Row, Col) :- wire(_, Row, Col).
retrieve_all(c, Row, Col) :- cr(_, Row, Col).
retrieve_all(r, Row, Col) :- red(_, Row, Col).
retrieve_all(g, Row, Col) :- green(_, Row, Col).
retrieve_all(b, Row, Col) :- blank(_, Row, Col).

The following rule extracts the flippable-
nonflippable structure corresponding to the last
level of the switching-graph from the file
graphNew (which is generated by the switching-graph
synthesizer) and assigns it to the variable Struct.
Then it traverses that flippable-nonflippable
structure from left to right and collects the
node-ids present in the structure in a list Y in
the same sequence in which they appear in the
structure. Next, it calls the rule gen_row_col which
generates say, x number of lists where x is equal to
the number of node-ids present in the flippable-
nonflippable structure mentioned above. Each such
list corresponds to a particular column of the
topological matrix to be formed and each element
of a list belongs to a particular row. All x number
of lists are generated in a list as a list Z of
lists. The Z is then reversed and the reversed
list is assigned to the variable X. Then the rule
determine_elements is called.

query(X) :- flip(Struct),
  traverse(Struct, [], Y),
  gen_row_col(Y, [1,1,1], Z),
  d_rev(Z, X/[]),
  determine_elements(1, X).

The following rule compiles the file graphNew
containing the output of switching-graph
synthesizer, the node database, opens a file called
topological_matrix and outputs the topological
matrix in that file.

make_matrix :- [graphNew],
  query(_,
        findall(_, brother(_,_,_),_),
        findall(_, matched(_,_,_),_),
        output(Sorted_list),
        tell(topological_matrix),
        process_sorted_list(Sorted_list, l),
        nl,
        get_cols_of_r_g_w_in_last_row(Sorted_list, RGW_Col_list),
        get_max_col_no_in_sorted_list(Sorted_list, Max_Col_No_in_Sorted_list),
        get_max_col_no_in_RGW_Col_list(RGW_Col_list, Max_Col_No_in_RGW_Col_list),
        make_last_two_rows(RGW_Col_list, Max_Col_No_in_RGW_Col_list, Max_Col_No_in_Sorted_list),
        nl,
        told,
        get_cols_of_r_g_w_in_last_row([], []):- !.

Following rule generates topological matrix with
time statistics.

gen_mat :- statistics(runtime, [T0|_]),
  make_matrix,
  statistics(runtime, [T1|_]),
  T is T1 - T0,
  format('make_matrix/mod17 took `3d sec.`n', [T]).
get_cols_of_r_g_w_in_last_row([], []):- !.

generategrammar(X) :- [grammar],
  query(_,
        findall(_, brother(_,_,_),_),
        findall(_, matched(_,_,_),_),
        output(Sorted_list),
        tell(topological_matrix),
        process_sorted_list(Sorted_list, l),
        nl,
        get_cols_of_r_g_w_in_last_row(Sorted_list, RGW_Col_list),
        get_max_col_no_in_sorted_list(Sorted_list, Max_Col_No_in_Sorted_list),
        get_max_col_no_in_RGW_Col_list(RGW_Col_list, Max_Col_No_in_RGW_Col_list),
        make_last_two_rows(RGW_Col_list, Max_Col_No_in_RGW_Col_list, Max_Col_No_in_Sorted_list),
        nl,
        told,
        get_cols_of_r_g_w_in_last_row([], []):- !.

num_rows(Row_count), !,
The following rule reverses the sorted list and extracts the Column number from the first element of the reversed list. Since the list is already sorted by column numbers in increasing order, so the first element of the reversed list will have the maximum column number.

get_max_col_no_in_sorted_list(Sorted_list, Max_Col_No):

d_rev(Sorted_list, [( ENTRY, Max_Col_No) ]/[]).
write_to_file(LLNewList, LNewList).

% when the number of cols, having W, G, R in a row,
% is a multiple of 4, then the wire is put till last
% but one col. After that, a blank "b" has to be put
% in the last col. The first of the rules
% fill_last_but_one_row takes care of this.
fill_last_but_one_row([], [], L, ['b#'] | L):- !.
fill_last_but_one_row([H1, H2, H3, H4, H5], Start, ILList, FList):- !,
    fill_with_wire_and_blank([H1, H2, H3],
        Start, ILList, ILList),
    fill_with_wire_and_blank([H4, H5], H3,
        ILList, ILList2),
    FList = ['b#'] | ILList2.
% this is required because
% wire is filled from H4 to
% H5 - 1. So the last col,
% H5, should be blank.
fill_last_but_one_row([H1, H2, H3, H4 | T], Start, ILList, FList):- !,
    fill_with_wire_and_blank([H1, H2, H3, H4],
        Start, ILList, ILList),
    fill_last_but_one_row(T, H4, ILList, ILList,
        FList).
fill_last_but_one_row([H | T], Start, ILList, FList):- !,
    fill_with_wire_and_blank([H | T],
        Start, ILList, ILList),
    FList = ['b#'] | ILList.
fill_with_wire_and_blank([H1, H2, H3, H4], Start, ILList, FList):- !,
    fill_with_blanks(Start, H1, ILList, ILList),
    fill_with_wire([H1, H2, H3, H4], ILList, FList).
% fill with wire from
% H1 to H4 - 1
fill_with_wire_and_blank([H1, H2, H3], Start, ILList, FList):- !,
    fill_with_blanks(Start, H1, ILList, ILList),
    fill_with_blanks(H1, H2, H3), ILList, FList).
% fill with wire
% from H1 to H3 - 1
fill_with_wire_and_blank([H1, H2], Start, ILList, FList):- !,
    fill_with_blanks(Start, H1, ILList, ILList),
    fill_with_blanks(H1, H2), ILList, FList).
% fill with wire
% from H1 to H2 - 1

fill_with_wire([H1, __, H4], IList, FList):-
  OneLessThanH4 is H4 + 1,
  put_wire(H1, OneLessThanH4, 
            IList, FList).

fill_with_wire([H1, __, H3], IList, FList):-
  OneLessThanH3 is H3 + 1,
  put_wire(H1, OneLessThanH3, 
            IList, FList).

fill_with_wire([H1, H2], IList, FList):-
  OneLessThanH2 is H2 + 1,
  put_wire(H1, OneLessThanH2, 
            IList, FList).

put_wire(X, X, List, [\'W#\'|List]):- !.

put_wire(H1, One_Less_than_Last_Element, IList, FList):-
  OneMoreThanH1 is H1 + 1,
  put_wire(OneMoreThanH1, 
            One_Less_than_Last_Element, 
            [\'W#\'|IList], FList).

fill_last_row([], _, List, List):- !.

fill_last_row([H1, H2, H3, H4, H5], Start, 
               IList, FList):- !,
    fill_with_blank_and_red([H1, H2, H3], 
                            Start, IList, IList1),
    One_more_than_H3 is H3 + 1,
    fill_with_blank_and_red( 
                             [H4, H5],
                             One_more_than_H3, 
                             IList1, FList).

fill_last_row([H1, H2, H3, H4 | T], Start, 
               IList, FList):- !,
    fill_with_blank_and_red( 
                             [H1, H2, H3, H4], 
                             Start, IList, IList1),
    One_more_than_H4 is H4 + 1,
    fill_last_row(T, One_more_than_H4, 
                  IList1, FList).

fill_last_row([H|T], Start, IList, FList):- !,
  fill_with_blank_and_red([H|T], 
                         Start, IList, FList).

fill_with_blank_and_red([H1, H2, H3, H4], Start, 
                        IList, FList):-
  fill_with_blanks(Start, H4, 
                   IList, IList1), 
  fill_with_blanks(Start, H4, 
                   FList = [\'R#\'|IList1].

fill_with_blank_and_red([H1, H2, H3], Start, 
                        IList, FList):-
  fill_with_blanks(Start, H3, 
                   IList, IList1),
fill_with_blanks(Start, H1, H2, IList, FList):-
fill_with_blanks(Start, H2, IList, IList),
fill_with_blanks(Start, H, List, FList),
fill_with_blanks(Start, H, IList, FList),
fill_with_blanks(H, H, List, List):- !.

% fill with blanks
% from "Start" to H3 - 1
FList = ['R#'| IList1].

% fill with blanks
% from "Start" to H2 - 1
FList = ['R#'| IList1].

% fill with blanks
% from "Start" to H - 1
Start < H,
NewStart is Start + 1,
fill_with_blanks(NewStart, H, [b#| IList], FList).

% extra blanks and E(Max_RWG_Col No, Max_Col No,
LList, L1NewList, LList, LNewList):-
No_of_Extra_Blanks is
Max_Col No - Max_RWG_Col No,
put_blanks_and_E(No_of_Extra_Blanks, LList, L1NewList, LList,
LNewList).

% extra blanks and E(0, LList, ['E#'| LList1], LList,
L1NewList, LNewList):- !.

% extra blanks and E(No_of_Extra_Blanks, LList, L1NewList, L1NewList,
LNewList, LNewList):-
Remaining_Extra_Blanks is
No_of_Extra_Blanks - 1,
put_blanks_and_E("
Remaining_Extra_Blanks,
['b#'| LList1], L1NewList,
['b#'| LList], LNewList).

% write to file(L1NewList, LNewList):-
d_rev(L1NewList, R_L1NewList/[]),
d_rev(LNewList, R_LNewList/[]),
write_to_file(R_L1NewList, R_LNewList).

write_to_file1(R_L1NewList, R_LNewList):-
write_to_file2(R_L1NewList, 0, Count),
Count =< 40,
write_to_file2(R_LNewList, Count, _).

write_to_file1(R_L1NewList, R_LNewList):-
write_to_file2(R_LNewList, 1, _).

write_to_file2([], Count, Count):- !.
write_to_file2([|T], ICount, FCount):-
NewCount is ICount + 1,
NewCount > 40, nl,
write(T).
write_to_file2(T, 1, FCount).

write_to_file2([H|T], ICount, FCount):-
    write(H),
    NewCount is ICount + 1,
    write_to_file2(T, NewCount, FCount).
VITA AUCTORIS

Arindam Das was born in 1970 in Ichapur, West Bengal, India. He completed his secondary school education at Kendriya Vidyalaya Ichapur and his higher secondary education at St. Xavier’s College, Calcutta, India in 1987. From there he went on to the Indian Institute of Technology, Kharagpur, India where he obtained a Bachelor of Technology (Honours) in Computer Science and Engineering in 1991. In August 1993, he finished his Master of Science in Computer Science at the University of Windsor.