Incremental Methods for Formal Verification and Logic Synthesis

by

Gitanjali Meher Swamy

B.Tech (Indian Institute of Technology, Kanpur) 1991
M.S. (University of California, Berkeley) 1993

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering-Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA, BERKELEY

Fall 1996

The dissertation of Gitanjali Meher Swamy is approved:

Chair Date

Date

Date

University of California, Berkeley

Fall 1996

Incremental Methods for Formal Verification and Logic Synthesis

Abstract

Incremental Methods for Formal Verification and Logic Synthesis

by

Gitanjali Meher Swamy
Doctor of Philosophy in Engineering-Electrical Engineering and Computer Sciences
University of California, Berkeley
Professor Robert K. Brayton, Chair

IC design is an iterative process; the initial specification of a design is rarely complete and correct. The designer begins with a preliminary and usually incorrect sketch (possibly from a previous generation design), and iteratively refines and corrects it. Usually, refinements are small, and there is much common information between successive design iterations. The current genre of CAD tools do not take into account this iterative nature of design. For each change made to the design, the design is re-verified and re-optimized without taking advantage of information from previous iterations. This leads to inefficient performance. In this thesis, we propose the paradigm of incremental algorithms for CAD. Incremental algorithms use information from a previous design iteration, as well as information about changes to the design to evaluate the design efficiently. In particular, we examine incremental algorithms for two different classes of CAD problems: *formal design verification* and *logic synthesis*.

Design verification is the process of checking if the design satisfies all the initial specifications. Most existing techniques for verification evaluate the entire design in a single pass. In practice design verification is never called just once; the designer tends to modify the system both iteratively and incrementally, and would like to incrementally call the verifier at each stage. Current techniques ignore this common information. This redundancy is particularly costly while dealing with large systems that take a lot of time and effort to verify.

This thesis proposes incremental formal design verification as a solution to this problem. Incremental verification runs the entire verification process only once, and propagates successive changes or increments thereafter. We have developed incremental algorithms for the two most commonly used methods for formal design verification: *language containment and model checking.*

Logic synthesis refers to the process of optimizing a logic description of a circuit, specified as a netlist of logic gates. This representation can be optimized for area, delay, and power. Most problems in logic synthesis are computationally hard, and are solved using heuristics. This often makes algorithms unstable; if the input is changed slightly, the new result of synthesis can be significantly different. Since a designer can spend much effort hand-optimizing circuits, it is desirable to retain as much of this human insight as possible. In addition, the network may have already been implemented in silicon at a lower level of the design hierarchy, and it can be inconvenient to change. We propose the paradigm of incremental synthesis, whose underlying motivation is to preserve the old design implementation, while keeping the objective (power,area,delay) reasonable.

In incremental verification, it is imperative to get exactly the same answer as by running non-incremental verification; incrementalization saves the designer computation effort and time by utilizing information from previous iterations. However, an incremental synthesis algorithm is concerned more with preserving similarity to the earlier design, and hence is not guaranteed to have the same result as the corresponding non-incremental algorithm.

The paradigm of incremental analysis, in both synthesis and verification, raises issues of detecting change from a new high-level specification of the design. We present methods for detecting changes made to the system from a high-level specification of the design.

The final overall goal of this thesis is create incremental algorithms for CAD, and to demonstrate their effectiveness to the user.

Professor Robert K. Brayton
Dissertation Committee Chair

To Amma and Appa

Contents

List of Figures

List of Tables

Acknowledgements

I started writing my acknowledgments long before I even knew what my thesis would be. It has been a constant source of joy to avoid writing the main body of my thesis by concentrating on my acknowledgments. After this process one comes away with a warm fuzzy feeling much like the feeling one gets while sitting on ones desk in school browsing the web, happy and fulfilled that you've been "doing something". Theoretically speaking, acknowledgment writing satisfies all the perquisites of being classified as working on your thesis: it is contained within your thesis, and you have to type it. Writing the acknowledgments has been fun: Let me begin by acknowledging the acknowledgment.... Obviously you can see where this is headed; I'm going to thank everyone from the janitor to the president, so please fasten your seat-belts for the ride.

Statutory Warning: These acknowledgments are full of effusive thanks and must only be read if not allergic to sentiment.

Here are my acknowledgments in no particular order:

To Professor Robert Brayton, a most wonderful and patient advisor. I think the following story illustrates this point. The first time I wrote a report and handed it to Bob, it came back neatly corrected and annotated with more red than black. It shook me; I had not realized until that point what a bad writer I was (operative word "was"). So I went to a friend of mine to ask for his advice. This friend's exact words were quote " Don't worry; all advisors do this. Just send him back the same paper." end-quote. Famous last words; I sent another copy of exactly the same document back to Bob, and it came back one day later with exactly the same set of annotations and a note: " Next time also give me a Xerox of the old draft with your new submission". Thats Bob for you, exceedingly patient and exceedingly thorough. He has put in much effort into teaching me the three r's: research, riting and rithmatic; I hope I have imbibed some fraction of it. Thank you to his wife Ruth, for accepting all of us in CAD into her family.

I want to thank Professor Alberto Sangiovanni-Vincentelli for being on my dissertation committee, and for his honest advice. He never minced his words whether they were criticism or praise. I fondly treasure both. Thank you to Professor Alice Agogino for being on my dissertation and qualifying exam committees. I really appreciate your time and interest in my work. I want to thank Professor Richard Newton for frank discussions, inspirational speeches, and good advice. I have always found your matter of fact brand of advice fun to listen to. Thanks for being on both my prelim and qualifying exam committees.

Without Brad Krebs and Judd Reiffin, our wonderful, kind, and understanding system administrators, nothing could have been achieved. Thanks for fixing all the stuff that seemed to break only when I used them. Thanks for patiently listening to my conspiracy theories on the CAD group computers. Thanks for not laughing your head off when I suggested a possible cause of a system administration problem.

I owe being in this field to my close friend and colleague Adnan Aziz. Thank you for mentoring my projects, for arguing and explaining the finer points of verification, and for recommending me to Bob (I guess he's not going to forgive you for that). I shall always remember that it is always the bottom of the ninth in life.

I owe much to the wonderful, and patient teachers that I have had over the years: Prof. R. N. Biswas, Prof H. S. Mani, Prof A. K. Mazumdar, Prof. S. Kar, Prof. J.K. Bhattacharya and my high school physics teacher D D. Mulherker. I think the word "patient" will debut many times in this thesis.

Thank you Sheila Humphreys, for encouraging me to believe in myself. Without you, there would have been no Ph.D.

I learned that things can work, and that large groups can function well together from the VIS group. A big thanks to all the members of the group: Adnan Aziz, Tom Shiple, Stephen Edwards, Sunil Khatri, Yuji Kukimoto, Shaz Qadeer, Rajeev Ranjan, Sriram Rajamani, Abelardo Pardo, Tiziano Villa, Shaker Sarwary, Serdar Tasiran. I have learned so much from your examples and I would love to work with you all again.

The person whose smart first principle based thinking I've always benefited from is Vigyan Singhal. Its always been a pleasure listening to his clear and brilliant dissection of ideas. I must also thank his wife Sonali for the numerous dinners.

I have the learned a lot from Tom Shiple. Thank you for answering all those BDD questions in the first year, and for showing me how much can be achieved by clear thinking. Also, a thank you to his wife Suzanne for that superb apple pie.

To Yuji Kukimoto, a thank you for letting me bounce ideas off you. I don't know if you realize how much you have contributed to this thesis. You have a precise and logical thinking that never failed to clear my thought processes.

Thank you to Tiziano Villa and Tim Kam, I had fun working with you on the multi-valued prime generator. Also, thank you for all those quietly sarcastic remarks, which I will not repeat.

I owe Sriram Rajamani a thank you all the fun time spent on the Isyn project. It was a great learning experience and your enthusiasm, and complete lack of cynicism was refreshing. What was also impressive were the imaginative ideas you came up with. Also, I must thank Shaz Qadeer, for his good mathematical insights, and off key singing.

I have to thank Stephen Edwards for all the sarcastic comments that have helped rid me of chips that may have been on my shoulder. As a collaborator, I found our discussions fun, even though we tried to kill each other over differences (thank you to Nina for changing that). Thank you for teaching me rock climbing and better coding. I hate to admit this but I had fun working and talking to you.

Thanks to Rick Mcgeer for being a great mentor to a callow first year graduate student. Its only now that I can appreciate how much effort you put into me as a student. Thank you for opening a lot of doors for me. To Felice Balarin, a thanks for mentoring the 290h project that started my thesis.

I've known Rajeev Ranjan and Renu Mehra for ages. Our friendship has evolved so dramatically. Thank you for all the research discussions, thank you for giving me shelter in your home, and most of all thank you for being such nice people. Thank you for the great jokes and the home cooked meals.

To my friends Jagesh and Alpa Sanghavi, thank you for all the glorious dinner parties. I still can't tell which I enjoyed more, the food or the company (both were sublime). I really enjoyed our discussions on politics, life and prop 209.

Eric and Sandy Felt, thank you for the fun occasions that we managed to get together. I wish there had been more. I shall never forget the Christmas carolling of 95.

Wendell and Mary Baker. Thank you for the sharp witticisms. How could I have survived my first year without them, or your answers to my technical questions ? Thanks for the apartment; it was a great place and you even threw in Louise (your grandmother) into the deal.

Desmond Kirkpatrick, thank you for all the linux/PC advice. Also, Sanjay asked me to add this: thank you for the tennis. Chris Lennard, thank you for the really crazy hot wings barbecues and the Sens code. Those and your loud raucous laughter on the other side of my cubicle have been some of the icings of life here. Premal Buch, thank you for the power_simplify code.

Thank you to all the 290ls students, who soon became accepted denizens of the CAD floor. It was a pleasure teaching you guys; you learned so fast. Amit Narayan, please call me when you start your mutual fund; I don't want to miss that investment. Sunil Khatri, thanks for teaching me the value of regression tests. Adrian Isles, Thanks for all the discussions on society, uninterpreted functions, and proof reading this acknowledgement.

Thank you to Jagesh Sanghavi, Eric Felt, Desmond Kirkpatrick, Lisa Guerra for weathering prelims together. I could not have understood all of that material without our study sessions.

Thank you to Rajeev Murgai for being such a happy fun loving and un-complicated friend. Thank you for listening to all my woes, explaining how to fix Tex, and always being around at 3:00 am. Thank you to KJ Singh for patiently explaining how nodes may be added to a network without mentioning the word imbecile at any point. Thank you to Jaijeet Roychowdhary for the gourmet cooking. Might I dare to add, no thanks for the harassment. Thank you to Sriram Krishnan, for his brutal frankness. He has informed me that no one will read this acknowledgment anyway, so I can write what I want. Thanks to Yoshinori Watanabe, the kindest TA in the world.

I'd like to thank Heather for patiently weathering the 290h project with me, Suzan Szollar for lending me her perl book, Carol Wawrukawiecz for the pumpkins, Dominique Borrione for all our discussions on verification and Ellen Sentovich for occasionally giving me some fun papers and advice. Please keep doling out both. It was inspirational to have such smart successful women around. Thank you to Subarnarekha Sinha for proofing this acknowledgment, and its nice to leave the mantle of the only woman in the CAD group to you.

Thanks to all other members and visitors of the group, who were around and smiled at me once in a while: ST Cheng, Iason Vassilou, Harry Hsieh, Ken Yamaguchi, Edoardo Charbon, Paul Stephan, Narendra Shenoy, Ramin Hojati, Luciano Lavagno, Alex Saldanha, Alexander, Michael, Ernst, Huey Wang, Mark Beardslee, the friendly young faces (Mukul, Gurmeet, Alok, Marco, Luca...) and everyone else who is going to kill me for leaving them out of this lengthy acknowledgment.

Thanks to Flora Oviedo, for making all of us in CAD a family (we're all her babies). Thank you to Kia Cooper, Justin Adler-Swanberg, Gwen Horn, and Elise Mills for being very supportive staff. Thank you to Sam the janitor for being there at 2:00 am every night. A thanks to the great guys in the grad office: Heather, Mary, Pearl, Ruth and Pat, without your superior knowledge of the convoluted workings of the grad division, all would have been lost. And another thanks to the ERL staff, Tito, Jeff, and of course, Teresita (I have to thank payroll).

A big thank you to the National Science Foundation and the University of California for funding me through these graduate years, and making this Ph.D. possible.

To all those friends from IITK: Thank you to Barathi and Rajeev Jayaraman. Life without you guys will seem strange. Thank you for shelter, fabulous stock picking advice, and showing me how to get rid of those irritating paragraph indents in ucthesis.sty.

Thanks to Bratati Ghosh and Satyajit Rangnathan. Your erudite and witty conversation alleviated Ph.D. blues and boredom. Plus, you were always ready for coffee at any time.

Thank you to Vandana Tandon and Rajat Mittal, your company always gave me a sense of perspective, i.e. hiking is more fun than work.

Thank you to Vivek Bhatt and Ashutosh Joglekar, my adopted brothers. Thanks for all the advice, it always paid off and you never charged for it.

I'd also like to thank someone who has had a profound effect on my life from age 14: Supriya Krishnamurthy, my childhood buddy. Thanks for a friendship that has not been dulled by either time or distance.

Thank you to Dan and Dri Engels for sharing the pain of moving from warm sunny Berkeley to cold Boston, and for turning the psycho cat on me. At least, we can suffer together.

Thanks to all those great guys at DEC: Antun Domic, Kolar Kodandapani, Nate Phillips, Heather Harkness, Channery, and Eric Lehman. The one summer I spent there was a great learning experience.

Thank you to Praveen for those deceptively gentle remarks on life. You were my kindest apartment mate. Also thank you to his wife Jyothi, for giving me shelter whenever I returned to work on my thesis.

Thanks to my first critics, Egmont, Winnie I, Winnie II, Winnie III, Esme, Emily,Penny, Goldie, Elsie and Callie. They expressed disapproval by eating my papers. Life has come a long way since then; reviewers are more civilized nowadays.....

Thank you to my wonderful parents, Dr. Swamy and Dr. Swamy, who told me I could achieve anything I wanted. Sometimes I think a strong motivation for completing my thesis came from visualizing the following telephonic scenario in my house:
Ring Ring....
Hello, Can I speak to Dr. Swamy ?
Sorry, he is in Madras.
Can I speak to Dr. Swamy ?
Sorry, she is in court.
Can I speak to Dr. Swamy ?
Speaking.
Thank you to my sister Chu for not beating me to being the third Dr. Swamy. Thank you to my great in-laws, Rani and Bab. Did you know what you were getting into when you welcomed me into your family ? Thank you to my wonderful aunt Sushiela for all the philosophy. Thank you to Anna and Sandy for giving me a home away from home.

The person I owe the most to is my husband Sanjay. He has been my friend, philosopher, mentor and guide. I appreciate his patient guidance and his humorous perspective. Sanjay truly believes in equality of the sexes: he has informed me that in the event that we are ever attacked by a grizzly bear, its every man/woman for themselves. Thank you for managing to be both my Pygmalion and treat me as an equal.

The End (about time).

Chapter 1

Introduction

1.1 Design Flow for Digital IC's

Digital IC's are designed using a *top-down design methodology* as shown in Figure 1.1. The design is optimized at three key levels of abstraction: behavioral, logical and physical. Each stage of optimization is followed by a translation to a more detailed level of representation. The final result of the top-down design process is a silicon mask, which is sent to the IC manufacturing facility to use in fabricating the chip.

A typical design flow begins with a description of the system in some high-level language. The three most common high-level *hardware description languages (HDL)* are Verilog [1], C [2], and VHDL [3]. Most high-level languages describe the design as combination of behavior (specified as a program) and structure (specified as hardware). *High-level synthesis* [4] is the process of optimizing and further translating this behavioral description of the design. It concludes by compiling this high-level description into a low-level HDL, which describes the same design as a *hierarchical* system of interacting circuits or *networks* of logic gates.

A network [5] is a graph whose nodes correspond to hardware elements such as logic gates and latches, and edges correspond to interconnections between them. The network is a logic gate circuit implementation for the design. We describe this hierarchy of networks using a low-level format called Blif-Mv [6], which supports a multi-valued, hierarchical system of networks. The hierarchical description of the design can be re-structured and optimized

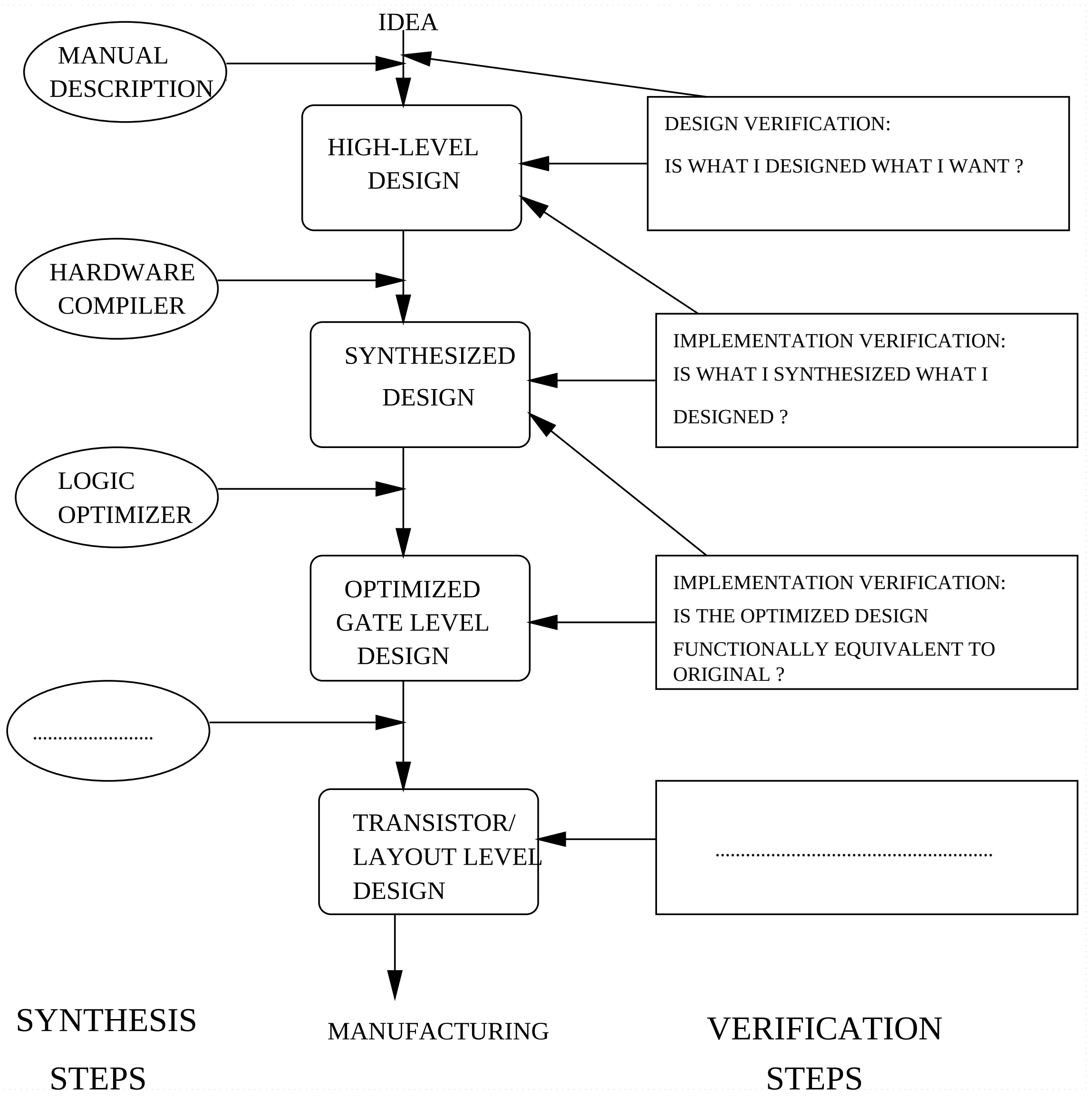

IDEA
MANUAL DESCRIPTION
HIGH-LEVEL DESIGN
DESIGN VERIFICATION:
IS WHAT I DESIGNED WHAT I WANT ?
HARDWARE COMPILER
SYNTHESIZED DESIGN
IMPLEMENTATION VERIFICATION:
IS WHAT I SYNTHESIZED WHAT I DESIGNED ?
LOGIC OPTIMIZER
OPTIMIZED GATE LEVEL DESIGN
IMPLEMENTATION VERIFICATION:
IS THE OPTIMIZED DESIGN FUNCTIONALLY EQUIVALENT TO ORIGINAL ?
....
TRANSISTOR/ LAYOUT LEVEL DESIGN
....
MANUFACTURING
SYNTHESIS STEPS
VERIFICATION STEPS

Figure 1.1: Design flow

and this process is called *hierarchical synthesis*

A critical part of the design process is identifying whether the design is correct. This process of checking if what was designed was in fact wanted is called *design verification.* Design verification of digital systems is traditionally accomplished by *simulation* [7] and more recently by *formal design verification* methods [8] [9].

To verify the design behavior, a finite state machine or FSM representation must be extracted from the network. Some methodologies may directly translate the high-level HDL into an FSM representation by bypassing the network. The FSM for a design includes a set of "states" representing the design at different times. The states correspond to the values on the latch (or timed) elements in the network. The FSM must also contain functional information that relates the next state of the design to the current state and inputs to the design. It represents the sequential and functional behavior of the design without any references to the actual hardware realizing it.

Verification usually involves traversing the state space of the FSM. If verification fails, the designer must modify his/her design, or possibly the statements that are being checked, and run verification again. This happens many times in the design process. The network description can also be sent to a synthesis engine in the event that the designer desires to optimize the network logic to simplify it for easier design verification.

Once the behavioral model of the design has been verified, the next stage consists of the logic-level optimization (or synthesis). The eventual goal of logic synthesis consists of creating an optimized network implementation for the design. This optimized representation must also be represented in terms of logic gates that are available in the set of basic gates prescribed by the manufacturing process. This set of basic gates is called the *technology* of choice. The design network is first optimized independent of the technology for objectives like area, delay and power etc [10]. This is called *technology independent logic synthesis.* After this, the netlist is mapped [11] into the technology of choice, and this is referred to as *technology mapping.* Next, the mapped circuit may also be further re-optimized; a process that is called *technology dependent logic synthesis.*

It is important to verify that this optimized design is indeed functionally the same as the design input to the synthesis process. This is accomplished via *implementation verification*;

a process that compares and verifies that these two designs are the same. Implementation verification can be performed between the behavioral and synthesized versions, as well as the synthesized and optimized versions of the design.

After logic optimization is complete, the next stage in the design process consists of *layout* [12], where the net-list of logic gates is translated into the actual "layout" on the silicon surface. The layout representation is in the form of polygons that represent transistors on the silicon surface. It patterns how portions of the silicon surface must be treated during manufacture. Common objectives in layout optimization may be minimizing area, reducing crosstalk etc. Finally, the silicon map obtained from the layout stage is sent to a silicon fabrication facility to make the IC chip.

The process of designing an IC chip is therefore complex and time consuming. Time to market is a very critical factor for the success or failure of any IC chip, and it becomes imperative to use all means to save time and effort in this process. Thus, the primary aim of any CAD method is to reduce time to market, and optimize performance.

1.2 Motivation: Design is Incremental and Iterative

It is a well established fact that IC design is an iterative process. The designer rarely has a complete and correct specification for a design to begin with. Usually he/she begins with a preliminary and usually incorrect sketch (possibly from a previous generation design) and incrementally refines and corrects it. There also many instances of intrinsically iterative [13] algorithms in both formal design verification and logic synthesis. There is much common information between successive iterations. Unfortunately, the current genre of CAD tools are monolithic in their approach; information is rarely used between stages. For each change made to the design, the design is re-verified and re-optimized and this leads to inefficiency as well as poor quality design.

We propose the new paradigm of *incremental algorithms* for CAD that use information from a previous design iteration, as well as information about changes to the design to efficiently re-compute the required quantities. In particular, we examine incremental algorithms for two different types of design problems: *formal design verification* and *logic synthesis*. By proposing this new class of algorithms we reduce the design time in two ways:

- By taking advantage of pre-computed information in *Incremental Formal Design Verification*, we reduce the time taken to compute information at each stage in the top-down design process,
- We pipeline the top-down design process by re-using earlier implementations, and hence allowing parallel development at a different level of the hierarchy in *Incremental Synthesis*. Pipelining reduces the design time.

In this thesis, we begin by identifying how to detect increments or changes to the system in Chapter 3. First, we describe design verification and demonstrate how this may be incrementalized in Chapters 4, 5 and 6. The objective of incrementalization in this context is to reduce the time at each stage in the design process by re-using information. In Chapter 7. we examine incremental logic synthesis, with the goal of reducing time by allowing pipelining of design process.

1.3 Detecting Commonalities between Designs

The first step toward creating incremental tools consists of detecting what information has changed in a design, and what information may be re-utilized. The problem of detecting differences between two designs reduces to the problem of identifying a "matching" between the two design networks. A matching is a function that maps each gate or "node" in the new circuit into one in the old circuit. If a matching does not exist, the node is mapped to a "null" node. The matching problem does not require any correspondences in inputs to the design; the purpose is to identify *structurally identical* regions in the networks. Verification and Synthesis information is computed from the network structure, and we hope to preserve some of the previously computed information by identifying commonalities of structure.

In Chapter 3 we propose a two stage algorithm that first identifies nodes that match functionally and next iteratively prunes this set to identify nodes such that their network substructures match. We examine all potentially matching substructures to find the set of matchings that are maximal, i.e. match the largest number of nodes. Once common substructures are identified, functional information for each substructure need only be computed once. Hence, we save in the computation of FSM information from the network.

1.4 Incremental Verification

1.4.1 Introduction: Formal Design Verification

Design verification is the process of checking if what the designer created was indeed what was intended. In the context of this thesis we will be concerned with the verification of sequential logic circuits [14] [15]. These circuits can be represented as finite state machines or finite automata, often using the process of abstraction [16].

Design verification of systems that can be modeled as FSM's may be performed in many ways. Two important paradigms used to verify finite state structures are: *Language Containment* and *Model Checking*. The behavior of a sequential circuit can be represented as a ω *regular* language [17], which is simply a regular language on infinite strings.

In Language Containment [18] the property is specified by a property automaton, which is an FSM which monitors the system, and accepts or rejects the behavior of the system based on some acceptance criterion. The property is satisfied if all behavior is acceptable. If the language of the system is a subset of the language of the property, then it indicates that the behaviors exhibited by the system satisfy the requirements on system behavior. This is referred to as *language containment*, and indicates that the design satisfies requirements. Language containment is checked by creating a product machine, which is the composition of the system automaton, and the complement of the property automaton. For the language containment check to be satisfied, the product machine must have an empty language. This language emptiness check is performed by traversing the product automaton to find the presence of states that are involved in valid (or bad) behavior. These states are also called *fair* states, and if the language emptiness check fails, the set of these states is returned as witnesses to the failure.

In Model Checking [19] the required behavior is specified by a formula in a temporal logic. If the system is a model of the formula representing required behavior, then the system is said to pass model checking. In this thesis, we examine a temporal logic called Computational

Tree Logic (CTL) [19]. In CTL model checking, the system automaton is traversed to ascertain states that satisfy the CTL property. This is done successively, by first marking states that satisfy sub-formulae, and then using them to compute the states that satisfy the entire formula. The CTL property is satisfied if and only if the specified initial states of the system FSM are contained within the states that satisfy the property.

The smallest basic CTL formulas are a labels on the states in the FSM that are called atomic propositions. Apart from atomic propositions, there are 4 four CTL formulae: $p + q, EXp, EGp, E(pUq)$, where p, q are CTL sub-formulae.

In general, an incremental approach will first identify the changes to the system FSM or properties and other constraints. Note that all methods of verification return some set of states that are witness to the failure of verification. The incremental method tries to use change information to update this witness set. Thus, our incremental algorithms are three step processes that consists of identifying changes, using them to recompute the witness set, and finally using the new witness set to compute a new answer to the verification question. Reachability analysis is also done incrementally, by using the change information to modify the reachable/unreachable set.

1.4.2 Incremental FSM Traversal

Both model checking and language containment involve some form of traversal (or reachability) of the state space of the FSM; the complexity of these algorithms is polynomial in the number of states. At the end of this traversal the states in the FSM are marked either *reachable* or *unreachable* from some specified initial state. This can be computationally expensive due to the "explosion" in the number of states in real design. The representation of the reachable states currently used in synthesis and verification is inherently non-updatable [20] [14]. In addition it tends to have a large representation, even if the finite state machine itself has a compact representation.

The incremental algorithms use the reachable set representation computed at a previous iteration, and information about the changes to the system to update it, rather than computing the reachable set from the beginning. We have also examined alternate ways to represent the reachable set that ease the incremental update of traversal information.

Remark: Instead of dealing with a single finite state machine, we will be dealing with systems of interacting finite state machines. In order to extend the analysis to such systems, we will be relying on the techniques of Aziz et al [21].

1.4.3 Key Idea for Incremental Verification

All methods for verification that are described above, involve the computation of "fixed point" algorithms [22]. Fixed point algorithms begin with some initial set of states in the FSM, and consistently add or remove states from this set, until a quiescent point also called fixed point, is reached. If the set of states increases monotonically, the fixed point is called the least fixed point or LFP, and if it is monotonically decreasing, it is called the greatest fixed point or GFP. An important observation about fixed point algorithms is that an LFP algorithm returns the correct final set of states (of the FSM), when supplied with any subset of the final set of states , and a GFP algorithm returns the correct final set of states (of the FSM), when supplied with any superset of the final set of states. We exploit this fact in order to generate an incremental algorithm. We use information about changes made to the system, and the original answer to compute the best starting set of states for a fixed point algorithm.

1.5 Incremental Synthesis

The aim of incremental logic synthesis differs considerably from that of incremental design verification. In incremental verification, we are concerned with getting exactly the same answer as by running ordinary verification. However, we hope to save time and effort of computation by utilizing information from previous iterations. In incremental synthesis, we are concerned less with performance. A designer can spend much effort hand-optimizing a circuit for speed or layout area, so it is desirable to retain as much of this human insight as possible. The aim of incremental synthesis is to retain as much of the old implementation as possible.

Unfortunately, most problems in logic synthesis are computationally hard, and are solved using heuristics. This often makes algorithms unstable; if the input is changed slightly, the new result of synthesis can be significantly different.

Preserving information will reduce time to market in two ways: (1) it avoids the effort of re-optimizing the unchanged portions of the design, and (2) it enables the pipelining of the design process by making synthesis algorithms “stable” (small input changes do not lead to drastic output changes). Thus, the high-level, logic-level and physical-level descriptions of the design can be simultaneously operated on, because the propagation of information between levels is relatively ”stable”.

Traditional methods for logic synthesis [23] represent the design as a graph, where the nodes represent logic gates that compute some Boolean function, and the edges represent interconnections between them. This representation is also called a *network*. Logic synthesis is accomplished by re-factoring the the Boolean functions computed by this network, so as to improve the area , delay, and power numbers. Traditional re-synthesis restructures the entire network.

Our approach will consist of finding subregions of the old network implementation, whose re-synthesis alone can realize the new functionality. The sub-regions alone are re-synthesized using traditional synthesis algorithms. There are conflicting objectives: we need to minimize the change to the implementation, while getting the best performance. To address these conflicting objectives, we introduce the notion of a *sensitivity measure* to determine which sub-regions give better performance with re-synthesis.

Given the old implementation of the design, and the new required specification (functionality) of the design; first we answer the following subproblem: given a region (by hitherto unknown means), we show how to check whether re-synthesis of the given region is adequate to effect the given change in functionality (sufficiency check). Next, we examine two procedures that use the above sub-procedure many times; an exact one and a heuristic one. The exact procedure implicitly represents all possible subregions and searches through them to find the exact minimum answer. In the heuristic procedure, we identify a means to evaluate the viability of the node for re-synthesis in terms of sensitivity. We combine this with the sufficiency check to get an incremental synthesis algorithm that iteratively adds nodes (in order of their sensitivities) to a region until it is sufficient for re-synthesis.

1.6 Background: Incremental Algorithms

There has been a body of work in incremental algorithms in related areas. The purpose of this section is to place these in perspective from the point of view of CAD. As stated in the previous section, design is an iterative process. In the traditional design paradigm, (Figure 1.2), each time the design is changed, the entire verification, synthesis and layout process is run from the beginning.

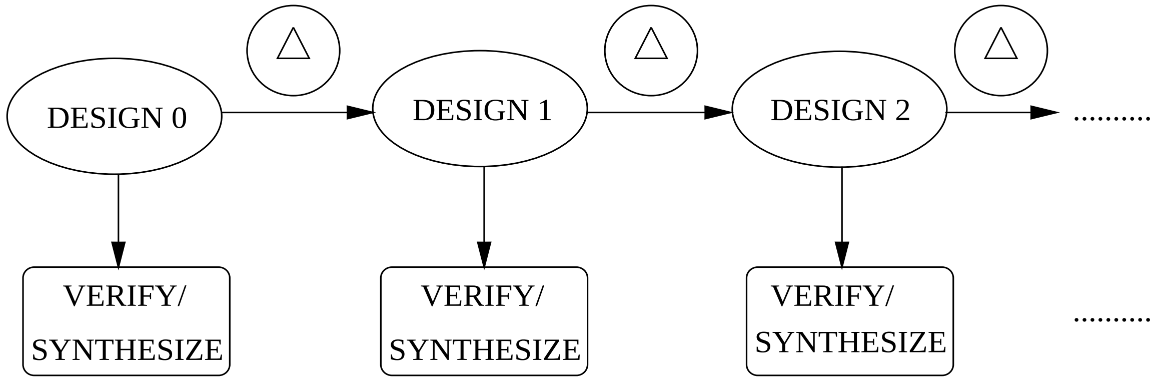


Figure 1.2: Traditional CAD

A superior paradigm is the incremental one (Figure 1.3), where the answer to each successive iteration is obtained by updating the previous answer.

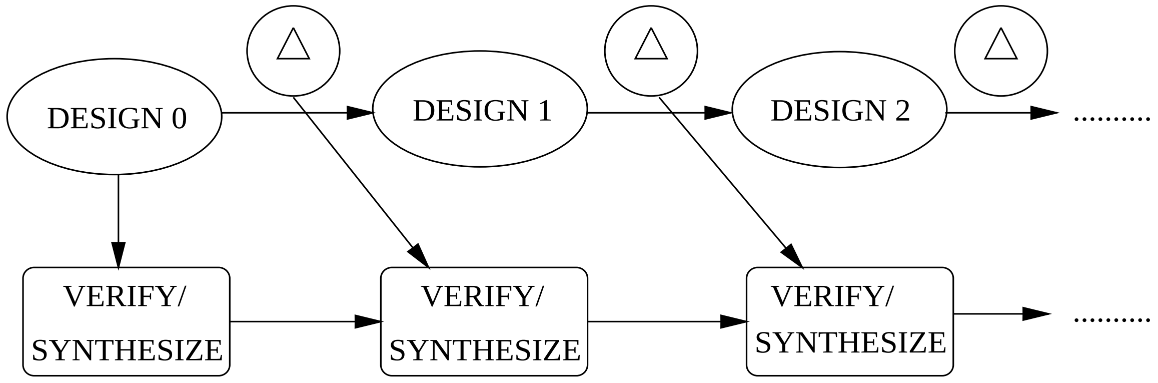


Figure 1.3: Incremental CAD

We define incremental algorithms as:

Incremental Algorithm: *An incremental algorithm is one that has the ability to re-use* *information about one input instance for another (close) input instance.*

There are no restrictions placed on the time, complexity or nature of update. This is less stringent than the criterion imposed by Ramalingam and Reps et al [24], who studied incremental algorithms for certain graph problems. This work defined incremental algorithms as those algorithms, whose time complexity could be written as a function of the change to the system alone, where the change or $\triangle$ could be written as the sum of the change in the input and output of the algorithms. Thus, $\triangle = \triangle_{input} + \triangle_{output}$, and the complexity of the algorithm $= O(f(\triangle))$. They show that some problems were intrinsically non-incremental; i.e. there was no locally persistent (storing only local information) algorithm that could be written for updating the information, which had a complexity only dependent on the size of the change.

Unfortunately, one problem which has no locally persistent algorithm and hence is not incrementalizable according to these criteria, is the problem of reachability, which is essentially the heart of all verification algorithms. Hence, for our purposes, we will impose a less rigid criterion for incrementality: we require that the incremental algorithm re-use more information than the corresponding non-incremental algorithm.

Incremental Algorithms have been proposed in many fields. They have been relatively common in the field of compilers and languages [25] [26] [27]. Incremental algorithms for layout are already present in the field of digital IC design. Some examples of this are in the work on incremental design rule checking by Taylor and Ousterhout [28], which are incorporated in the *Magic* system. At the layout level, the circuit is laid out as a map of silicon polygons. This map is used by the manufacturing facility to determine how to process different regions on the silicon surface. It is important to verify that the silicon polygons created may actually be manufactured, i.e. they are not too close together etc. This process of checking is called *design rule checking* (DRC). In [28], when the layout was modified, instead of running the design rule checker on the entire circuit, the tool recorded areas that were modified and only re-checked the modified areas. In order to perform DRC incrementally, Magic had to store additional information on the status of a region. Since DRC is also a form of verification, this technique is sometimes referred to as incremental verification; however, we are not interested in verification at this level in the design process. There is no commonality between this work and ours, other than the incremental paradigm.

Swartz et al [29] also proposed a similar methodology for incremental layout.

Closer to our work is the work done in [24] on the computational complexity of graph problems. Similar problems had been examined in [30]. They proposed incremental algorithms for three classes of graph theoretic problems: the single sink shortest path, the all paths shortest path, and the circuit value problem. We shall illustrate the single sink shortest path. The incremental algorithm modifies Dijkstra's algorithm [31] for the sssp problem. They store the graph of all shortest paths annotated with the distance to each node. The incremental update consists of two procedures:

Delete Edge: If an edge is deleted from the system, and it is not a member of the shortest path graph then no further update is necessary. If it is a member, then successor states to the edge, which have no other predecessor must be re-assessed. These vertices are identified as the affected set.

Add Edge: If an edge is added to the system, the predecessor to this edge must be reassessed. It must be identified as an affected state. Both Add and Delete Edges update the distances from the affected states by checking to see if the distance to their immediate successors may change. If the distance does change (due to the addition or subtraction of edges), then the algorithm propagates the affected state frontier to the successor states of the current affected set. This propagation is very similar to the algorithm of Dijkstra original algorithm for shortest paths [31]. Figure 1.4 illustrates the functioning of this algorithm on one example. The complexity of this algorithm is shown to be $O(\triangle log(\triangle))$, where $\triangle$ is the size of the affected set. However, this work was restricted to a limited set of graph theoretic algorithms, and there was no experimental justification offered.

In the field of verification, relevant work done concurrently with our research is on incremental model checking in modal mu-calculus [32]. They presented an incremental algorithm for model checking in the alternation free fragment of modal mu-calculus. They used as starting point the linear time algorithm of Cleveland and Steffen. One of the flaws of this work lay in the relative lack of experimental validation. Our work was done concurrently, with this work and subsumes it.

Also relevant is the work done concurrently by Kim et al [33] on efficient prototyping based on incremental design and module by module verification. They describe a system for

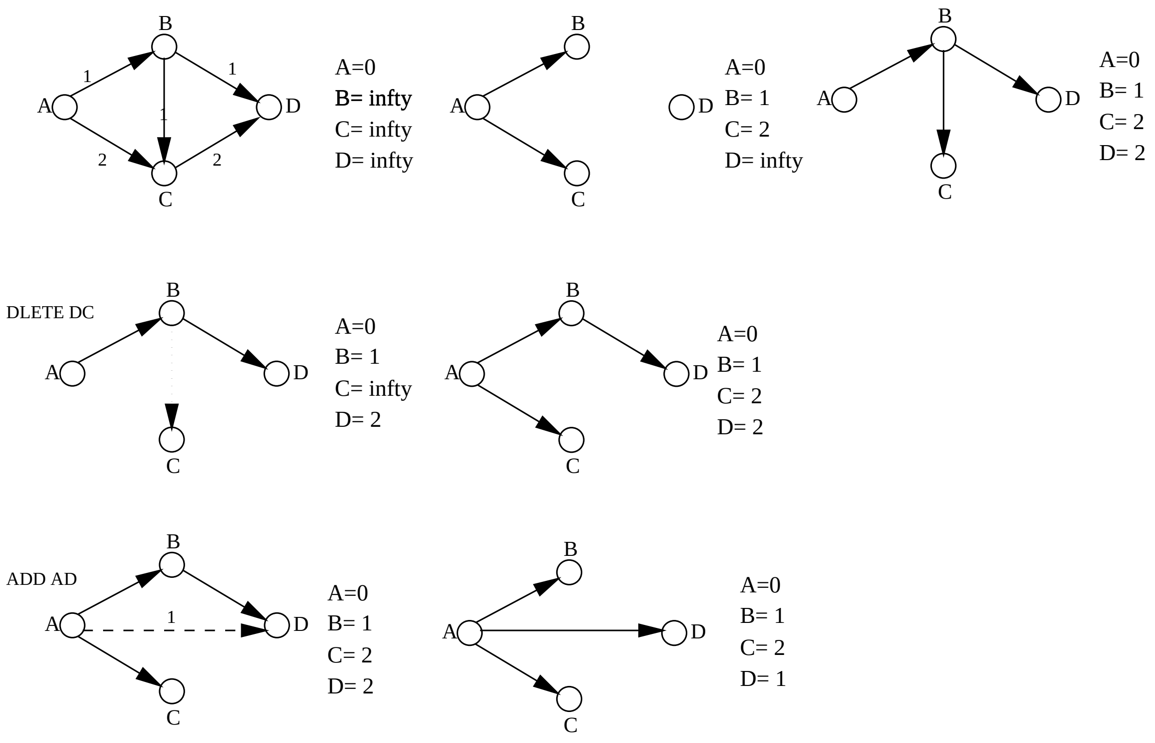


Figure 1.4: Single sink shortest path

incremental prototyping, and the main focus of their work is on methods for iteratively refining the design. Their work would primarily be an application for our algorithms rather than an alternate approach.

In the field of incremental logic synthesis, there is related work by Watanabe et al [34], which represented the design as a network. If on a successive design iteration the functionality of the network is changed to some new functionality, then the old implementation was updated by adding post and pre-rectifying logic to it. This method had the intrinsic disadvantage that at each iteration the network only increased in size. Brand et al [35] approached this problem by identifying cones of un-affected logic. If the function of a node in the network was changed, the transitive fanins (cone) was affected. Only affected logic was re-synthesized. The flaw in this work lay in the fact that if the output function was changed, then the entire circuit was affected. An alternative approach was proposed first by Kukimoto et al [36] [37] and then Lin et al [38] that attempted to obtain a region within the network, whose re-synthesis alone was sufficient to realize the new functionality. These works proposed identifying whether or not the re-synthesis of a region was sufficient to realize the functionality of the new circuit by computing the observability relation of the region that was consistent with the overall functionality (new) and compatible with the remainder of the circuit. However, a major limitation of these works is that none of these methods suggested a strategy for identifying a good region for re-synthesis.

1.7 Summary

In this thesis we have provided a theoretical framework for the incremental CAD, as well as experimental evidence to show that considerable gains may be obtained by the use of such a paradigm. We address the following goals:

1. We show in Chapter 3 how incremental modifications to the system can be made, and recognized from a network (HDL) description of an FSM. In particular, we find a solution to the problem of finding a maximal set of common subregions between two successive versions of a design. We give two algorithms to detect common substructures; an exact one and a heuristic one. The exact solution may only be run on small examples. For these, we experimentally show that the quality of the solution of the heuristic algorithm is comparable with the exact answer. We prove the correctness of both algorithms. Next, we experimentally illustrate incremental preservation of information by recomputing functional information for only the subregions that are not common.

2. We address the following four aspects of the incremental design verification problem:

   (a) Most computations in FSM based verification are of a fixed point nature. We prove certain properties about general fixed point algorithms in Chapter 2, and show how these properties of fixed points can be exploited to get incremental algorithms.

   (b) We give incremental approaches for FSM traversal, and evaluate their performance in Chapter 4. The reachability computation (or traversal) is an integral part of verification and synthesis. We make this incremental by storing different variants of traversal information, which are used to reconstruct new traversal information in the event of a change to the FSM representing the design. We give three different strategies for incremental FSM traversal based on different amounts of stored traversal information. We prove the correctness of all these procedures and experimentally show how gains in time may be obtained by their use.

(c) We also propose an incremental strategy for language containment; one method of design verification. We construct an incremental algorithm to use the information about changes to the system, and the answer to the previous computation of language containment. We prove the correctness of this algorithm and compare its performance with the current methods. This is described in Chapter 5.

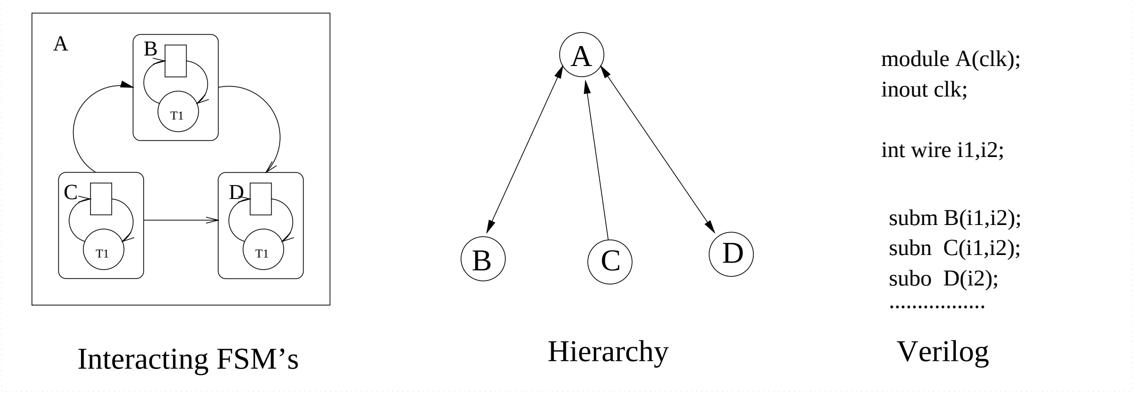
(d) In Chapter 6, we extend the aforementioned ideas to get an incremental algorithm for model checking, another method for design verification. We prove the correctness of this algorithm and prove some theoretical results about further improvements to incremental model checking.

3. We propose an incremental algorithm to the synthesis problem. This algorithm attempts to preserve the previous implementation of a design, while realizing a new functionality. We prove the correctness of such an algorithm. We also examine an exact algorithm for realizing the smallest possible change to an implementation to get a new functionality. We experimentally evaluate the performance of these algorithms. Chapter 7 describes this incremental approach to synthesis.

Chapter 2

Some Terminology in Digital Design

As explained in Section 1.1, IC's are designed using a *top down design methodology.* In this section, we relate the various representations of the design at the high-level and logic level in the top-down design methodology. This methodology begins with a high-level description of the design. This high–level description represents the design as a hierarchy of interacting modules.



```
module A(clk);
inout clk;

int wire i1,i2;

 subm B(i1,i2);
 subn  C(i1,i2);
 subo  D(i2);
 .................
```

Figure 2.1: Hierarchy

Definition 1 Hierarchical Netlist*[6]: A Hierarchical Netlist (or Hierarchical Node) is defined recursively as an n-tuple* N, I, O, H*, where*

- *N is a circuit or interconnection of logic gates.*
- *Σ is a finite set of input values.*
- *Γ is a finite set of output values.*
- *H is a set of hierarchical netlists (or children).*

A hierarchy node that is not the child of any other node is called a root node. Each node interacts with other nodes via its inputs and outputs. We show a system of interacting modules, the corresponding hierarchy and Verilog description in Figure 2.1.

In this system of interacting modules, each module has a "state". It reads in inputs from other modules and the environment, and depending on these moves to another state, while producing outputs for the other modules to read. Our designs are represented as modules with a finite number of states, and hence we refer to them as Finite State Machines (FSM).

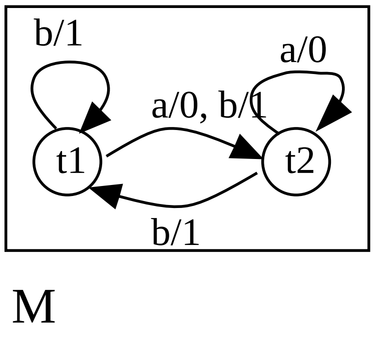


Figure 2.2: Finite state machine

Definition 2 Finite State Machine*: A finite state machine or* finite automaton $\mathcal{M}$ *is a 5-tuple* $(Q, \Sigma, \Gamma, T, I)$ *where*

- *Q is a finite set of states*
- *Σ is a finite set of input values*
- *Γ is a finite set of output values*
- *$T \subseteq Q \times \Sigma \times \Gamma \times Q$ is the transition relation*

- *I is a set of initial or starting states of the machine.*

Figure 2.2 describes a finite state machine with 2 states, 1 input, 1 output, and 4 edges in the state transition graph, i.e transition relation.

$T(q, \sigma, \gamma, t) = 1$ means that from state $q \in Q$ on input $\sigma \in \Sigma$, there is a transition to some state $t \in Q$, while the output is $\gamma \in \Gamma$. Thus an FSM can be represented by a *state transition graph*, whose vertices are states, and edges are labeled with elements of $(\Sigma \times \Gamma)$.

Since our entire system is a collection of interacting modules, we sometimes refer to the entirety as a *Product Machine.*

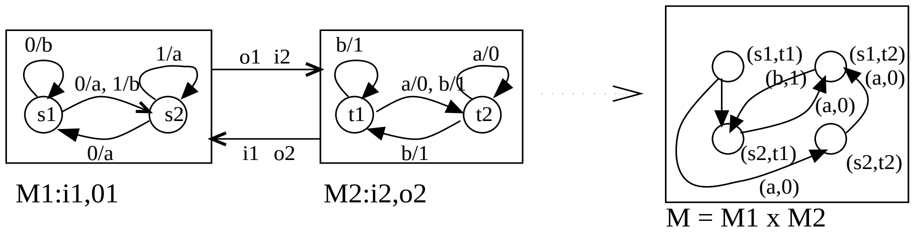


Figure 2.3: Forming the product machine

Definition 3 Product Machine*: Given a collection of interconnected finite state machines* $\{M_1, M_2, \ldots, M_n\}$*, their* product *is the finite state machine on the product state space. The transition relation is the Cartesian product of the component transition relations.*

In Figure 2.3 we describe a system of interacting finite state machines M_1, M_2, and the corresponding product machine M.

A *closed system* of interacting FSM's is a system with no external inputs. Any open system can be closed by adding machines that simulate the environment.

Each module, or collections of modules may be "flattened" into a single circuit or network. A network represents the design as digital hardware, which is an implementation of the design.

Definition 4 Boolean network *[5]: A* **network** $N = \{n_i\}$ *is a set of nodes with three associated functions:* func(n) *is the* function *of the node,* fanins$(n) \in \{0, 1, \ldots\}$ *is the number* *of* fanins *of the node, and* $\text{fanin}(n, k) \in N, k = \{1, \ldots, \text{fanins}(n)\}$ *is the kth fanin of the node.*

$\mathcal{N}$, can also be thought of as a directed acyclic graph (DAG) such that each node in $\mathcal{N}$ has a Boolean function $(func(n)(n_1 \ldots n_m))$. There is a directed edge from node n_i to node n if the function $func(n)$ is dependent on node n_i. We say that node n_i is a **fanin** of a node n, and node n is a **fanout** of node n_i.

A node n_i is a **transitive fanin** of a node n if there is a directed path from n_i to n, and we denote n as the **transitive fanout** of n_i.

Some nodes of a Boolean network are denoted as its **primary inputs** and **primary outputs**. Primary inputs are read from the environment outside the network, and primary outputs are written to the environment.

Nodes with at least one fanin and one fanout are called **internal**.

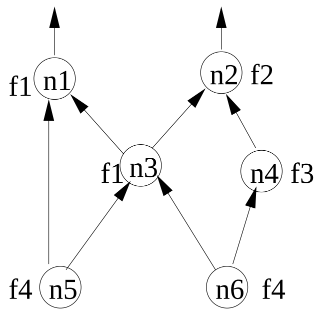
Figure 2.4 describes a small network with 6 nodes.

Let y_i denote the output variable of node n_i, unless n_i is a primary input or output. Let x_i denote a primary input and z_i denote a primary output. If $func(n_3) = +$, then $y_3 = x_5 + x_6$.

In a **sequential** network, some nodes are designated as **latches**. The output of a latch is a function of its inputs at the previous time step.

Notice that the output of each node is a Boolean function of its inputs. Both logic synthesis and formal verification consist of algorithms that manipulate Boolean functions and relations on the *network* and *FSM* domains.

Definition 5 *A* **completely specified** *Boolean function F with n inputs and m outputs is a mapping $F : B^n \longrightarrow B^m$, where $B = \{0, 1\}$. If $m = 1$ the* **onset** *and* **offset** *are the set of points satisfying $F(x) = 1$ and $F(x) = 0$ respectively. A* **minterm** *v of a function F is a vertex (i.e. a point in B^n) such that $F(v) = 1$. The* **cofactor** *F_{x_i} of F (completely specified) with respect to variable x_i is the function F evaluated at $x_i = 1$. The* **Shannon** *form of $F = x_i \cdot F_{x_i} + \overline{x_i} \cdot F_{\overline{x_i}}$ (i.e. in terms of its cofactors). The* **size** *of $F(x)$, $(|F(x)|)$ denotes the number of minterms (onset points) in $F(x)$. An* **incompletely specified** *Boolean function is a mapping $F : B^n \longrightarrow Y^m$ where $Y = \{0, 1, \star\}$ ($\star \Rightarrow F$ can be 0 or 1).*



$$N = \{n_1, \ldots, n_6\}$$
$$\text{func}(n_1) = \text{func}(n_3) = f_1$$
$$\text{func}(n_2) = f_2$$
$$\text{fanins}(n_2) = 3$$
$$\text{fanins}(n_5) = \text{fanins}(n_6) = 0$$
$$\text{fanin}(n_2, 1) = n_3$$
$$\text{fanin}(n_2, 2) = n_5$$
$$\text{fanin}(n_2, 3) = n_4$$
$$\text{tf}(n_1) = \{n_3, n_5, n_6\}$$

Figure 2.4: A network

If $m = 1$ the **onset**, **offset**, *and* **don't care set (dcset)** *are the set of points such that $F(x) = 1$, $F(x) = 0$, and $F(x) = \star$ respectively.*

A relation is a more general form than a function. In order to represent a relation as a function, it must be *deterministic* and *completely specified*. Determinism requires that for each input assignment, there is exactly one output assignment. Complete specification implies that there is no input assignment for which there does not exist an output assignment.

Definition 6 *A* **Relation** *is a mapping $O^F(x,z) : B^n \times B^m \longrightarrow B$, where x ($|x| = n$) are inputs and y ($|y| = m$) are outputs. Given inputs x and outputs z, a relation is characterized as $O^F(x,z) = 1$ if input assignment x leads to output assignment z, and $O^F(x,z) = 0$ otherwise.*

Any Boolean function with inputs $x = (x_1, \ldots, x_n)$ and outputs $z = (z_1, \ldots, z_m)$, $(z = F(x))$ may also be represented as its **observability relation** $O^F(x,z) := z \overline{\oplus} F(x)$. Given inputs x and outputs z, an observability relation is characterized as $O^F(x,z) = 1$ if $z = F(x)$ and $O^F(x,z) = 0$ if $z \neq F(x)$.

Logic Synthesis is concerned with obtaining an optimal network description of the design. This is usually accomplished by finding a better function for each node, or groups of node in the network.

Notice that the latch elements in hardware have a "time" aspect. Thus, the output of a latch at the next time step is a function of the inputs of the latch at the current time step. Thus, the circuit has a corresponding finite state machine, whose transition relation relates the latch outputs and primary outputs to latch inputs and primary inputs at the previous time step.

In most FSM based formal design verification algorithms, we define functions that map the set of states of the FSM representing the design to itself. The algorithms are concerned with computing the fixed points of these functions.

Definition 7 **Fixed point***[22]: Let $f : Q \rightarrow Q$ be a monotone (increasing or decreasing) function, the fixed point FP of f given I is given by the set $f^i(I)$, where $f(f^i(I) = f^i(I))$.*

If f is monotonically increasing, then the fixed point is called the *least fixed point* or LFP, and if f is monotonically decreasing, it is called the *greatest fixed point* or GFP.

```
$FP(f(Q), I$
  $R = f(I)$
if $(R = I)$
  return $R$
else
  return $FP(f(Q), R)$
```

Consider $S = \{$*The set of all subsets of states of the FSM*$\}$. We can define a linear order of elements of this set:

Rules 2.1 $\forall_{a,b \in S} a \leq b \Leftrightarrow a \subseteq b$

Every subset if bounded and below trivially as the set of states is a finite and discrete set. Hence, we can apply the Knaster Fixed Point theorem [39] to guarantee that there exists at least one LFP for given a function f that is monotone increasing. Similarly, if I define the linear order:

Rules 2.2 $\forall_{a,b \in S} a \leq b \Leftrightarrow a \supseteq b$

By a similar logic, I can guarantee that there exists at least one GFP for given a function f that is monotone decreasing.
The key ideas in this thesis are based on the following theorems:

Theorem 2.1 *If* $f_I, f_S : Q \rightarrow Q$ *are monotonically increasing and* $f_S(Q) \subseteq f_I(Q)$*, then* $\forall_S I \supseteq S$*,* $LFP(f_S(Q), S) \supseteq LFP(f_I(Q), I) = L$*.*

Proof. $S \supseteq I \Rightarrow f_S(S) \supseteq f_S(I)$ (monotonicity).
$\Rightarrow f_S(S) \supseteq f_S(I) \supseteq f_I(I)$ (by definition).
$\Rightarrow \forall_k {f^k}_S(S) \supseteq {f^k}_I(I)$.
$\Rightarrow LFP(f_S(Q), S) \supseteq LFP(f_I(Q), I)$.
□

Theorem 2.2 *If f is monotonically increasing $Q \subseteq f(Q)$, and $L = LFP(f(Q), I)$, then $\forall_S I \subseteq S \subseteq L$, $LFP(f(Q), S) = L$.*

Proof. Assume the converse. Let $L' = LFP(f(Q), S)$, $L = LFP(f(Q), I)$.

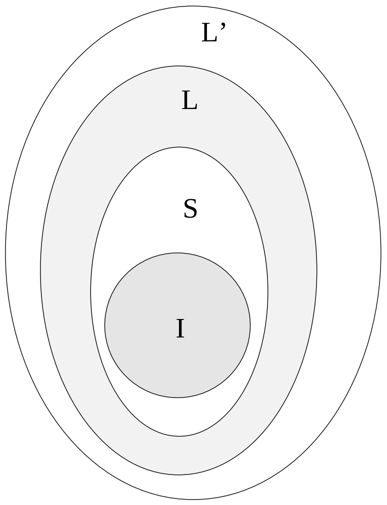


Figure 2.5: Proof: LFP theorem

From Theorem 2.1 $L' \supseteq L$
Consider $L' \bigcap \overline{L}$.
Let H_1 be the set such that $f(H_1) = L' \bigcap \overline{L}$

1. $H_1 \bigcap L = \emptyset$.
   If not, $\exists s \in L$ such that $f(s) \notin L$.
   $\Rightarrow L$ is not a fixed point.
   A contradiction, hence proved.

   Repeating this argument, there is a sequence H_k.
   $f^k(H_k) = L' \bigcap \overline{L}, H_k \bigcap L = \emptyset$.
   Let I reach fixedpoint in n steps.
   Hence, $\exists H_n \supseteq I$ such that $f^n(H_n) = L' \bigcap \overline{L}$.

2. $H_n \bigcap S = \emptyset$.

   If not $\exists x \in H_n \bigcap S$.

   $\forall_k f^k(x) \notin L$ (else a contradiction).

   i.e. $\exists x \in S, \forall f^k(x) \notin L$.

   L is not a fixedpoint.

   A contradiction, hence proved.

3. $H_n \bigcap S = \emptyset$ and $H_n \subseteq L' \Rightarrow L' \not\supseteq S$. A contradiction, hence $L' = L$.

□

Theorem 2.3 *If $f_I, f_S : Q \rightarrow Q$ are monotonically decreasing and $f_S(Q) \subseteq f_I(Q)$, then $\forall_S S \subseteq I$, $GFP(f_S(Q), S) \subseteq GFP(f_I(Q), I) = G$.*

Proof. $S \subseteq I \Rightarrow f_S(S) \subseteq f_S(I)$ (monotonicity).
$\Rightarrow f_S(S) \subseteq f_S(I) \subseteq f_I(I)$ (by definition).
$\Rightarrow \forall_k {f^k}_S(S) \subseteq {f^k}_I(I)$.
$\Rightarrow GFP(f_S(Q), S) \subseteq GFP(f_I(Q), I)$.
□

Theorem 2.4 *If f is monotonically decreasing $f(Q) \subseteq Q$, and $G = GFP(f(Q), I)$, then $\forall_S G \subseteq S \subseteq I$, $GFP(f(Q), S) = G$.*

Proof. Assume the converse. Let $G' = GFP(f(Q), S)$, $G = GFP(f(Q), I)$.
From Theorem 2.3 $G' \subseteq G$
Consider $G \bigcap G'$.
Let H_1 be the set such that$f(H_1) = G \bigcap \overline{G'}$

1. $H_1 \bigcap G' = \emptyset$.

   If not, $\exists s \in G'$ such that $f(s) \notin G'$.

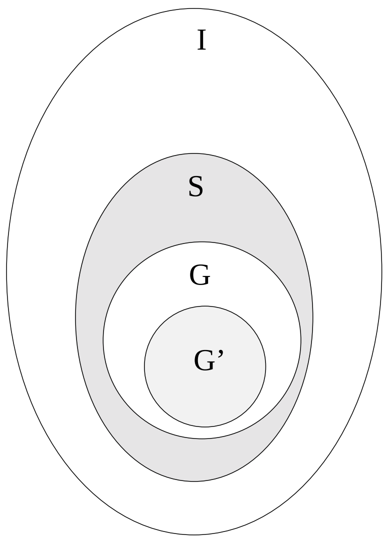


Figure 2.6: Proof: GFP theorem

$\Rightarrow G'$ is not a fixed point.

A contradiction, hence proved.

Repeating this argument, there is a sequence of H_k's.

$f^k(H_k) = G \bigcap \overline{G'}, H_k \bigcap G' = \emptyset$.

Let I reach fixedpoint in n steps.

Hence, $\exists H_n \subseteq I$ such that $f^n(H_n) = G \bigcap \overline{G'}$.

2. $H_n \bigcap S = \emptyset$.

   If not $\exists x \in H_n \bigcap S$.

   $\forall_k f^k(x) \notin G'$ (else a contradiction).

   i.e. $\exists x \in S, \forall f^k(x) \notin G'$.

   G' is not a fixedpoint.

   A contradiction, hence proved.

3. $H_n \bigcap S = \emptyset$ and $H_n \supseteq G \Rightarrow G \not\supseteq S$. A contradiction, hence proved.

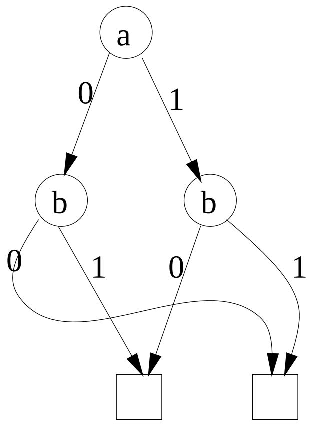


Figure 2.7: BDD for $f(a,b) = \overline{a} \cdot b + a \cdot \overline{b}$

□

Theorem 2.5 *The GFP and LFP satisfy the following rules.*

1. $GFP(f(Q),(A+B)) \subseteq GFP(f(Q),A) + GFP(f(Q),B)$.
2. $LFP(f(Q),(A+B)) \subseteq LFP(f(Q),A) + LFP(f(Q),B)$.

Proof. From Theorems 2.1 and 2.3. □

One data structure which is commonly used to represent all these Boolean functions and relations is a binary decision diagram.

Definition 8 Binary Decision Diagram*[40] [41]: A binary decision diagram or BDD, (Figure 2.7) is a compact representations of recursive Shannon decompositions for the function. A function graph is a rooted directed graph with a vertex set containing two types of vertices: a non terminal vertex which has a variable identifier and two children (a left child and a right child), and a terminal vertex which has as attribute 0 or 1. This representation is canonical for given variable ordering, and is used to implicitly represent sets of points.*

The BDD can be used to implicitly represent sets of points. The complexity of operations on a BDD are proportional to the number of nodes, however the number of vertices that a BDD may represent may be as much as exponential in the number of nodes. Hence, this representations allows us to operate on sets of states, instead than on individual states, and this is referred to as *implicit* computation.

A typical example is the computation of the set of reachable states from the initial states. Using Binary Decision Diagrams (BDDs) as a data structure, Boolean formulas and operations on them can be performed efficiently even on instances of very huge size, where explicit enumeration would fail. The reason is that the size of BDD representations is not linear in the size of the represented sets. Issues of BDD ordering [42] [43], partition of the system representation [44], and new types of decision diagrams have been explored to represent compactly important sets of Boolean and discrete functions.

Though not explicitly mentioned throughout this thesis we will be representing and manipulating quantities via their BDDs.

Chapter 3

Detecting Commonalities Between Designs

3.1 Introduction

The first step towards an incremental paradigm consists of detecting change, or alternately identifying what information may be preserved.

We address the problem of finding a high quality matching between two networks. We compare pairs of networks—combinational logic designs represented as directed acyclic graphs whose nodes are generalized (multi-valued, non-deterministic) gates and whose edges are generalized (multi-valued) connecting wires. We look for matchings, functions $M : N \to N' \cup \{\emptyset\}$ from each node in a new network N to a node in an old network N' or to "unmatched" ($\emptyset$) such that if $M(n) = n'$, then the gates at nodes n and n' are identical (when their inputs are permuted) and their fanins match ($M(n_k) = n'_k$ for corresponding fanins n_k and n'_k). The quality of a matching is the number of matched nodes $q(M) = |\{n \in N | M(n) \neq \emptyset\}|$. We solve the problem of finding the maximum quality matching.

The ability to reuse information is the primary motivation for solving this problem. One application, incremental design analysis, stems from the iterative nature of design. A designer usually wants to analyze each version of a design (with, e.g, a formal verification check). Analysis can be done more efficiently by identifying unchanged portions of a design and reusing the information computed for them. If we have a matching M, we can reuse information for each node n where $M(n) \neq \emptyset$. Our techniques may also be used to identify common areas within a single design, allowing common information to be computed efficiently. Another application is incremental synthesis, where the aim is to preserve as much of the old design as possible.

A matching corresponds to structurally identical transitive fanin cones of the design that start at a node and contain all the nodes and wires in its transitive fanin. We choose to identify these because the global function at a node is a function only of its transitive fanins. An example is the transition function [45], used frequently in formal verification and usually computed using BDDs [40]. Identifying matching nodes allows us to compute the new BDD by substituting variables, which can be done efficiently.

The approach we propose does not require any additional matching information (e.g., correspondences between the primary inputs). We expect most designs we compare will be the *output of a compiler* that does not usually supply any correspondence information. An alternative would be to use names to guess correspondences, but this is insufficient when names are automatically generated—they are often very sensitive to small changes in a design. Finally, by not assuming input correspondences, our algorithms can be applied to more general problems such as identifying identical portions within the same design.

We propose a greedy three-phase algorithm to find a good matching. First, nodes with identical functions are identified. Next, this information is combined with connectivity information to find nodes that have identical structures in their transitive fanins. Finally, the matchings implied by these nodes are combined into a high-quality matching. We use both a greedy heuristic, as well as an exact formulation.

Other approaches to incremental synthesis rely on knowing input correspondences. Brand et al's [46] work on incremental synthesis, which identifies regions of commonality similar to our own, requires knowledge of input correspondences and can only detect regions that start at the inputs. We assume that any two primary inputs may match if they can take the same set of values.

Burch et al [47] solve a functional matching problem that does not require input correspondence information. However, they are only comparing Boolean functions, and their approach does not generalize to circuit designs. Our main objective is to get a quick matching rather than the exact node function matching. We adopt a similar notion of a semi-canonical form, but our form is simpler (and hence faster) at the expense of some precision. Also, we deal with more general multi-valued functions [48], rather than just binary.

The techniques presented here can be used to drive the incremental verification algorithms of Swamy et al [49] [50] and Sokolosky et al [32]. These use information about the similarities between two designs to speed up the verification process.

This chapter is organized as follows. Section 3.2 contains exact and heuristic solutions to the network (structural) matching problem. We present both an exact formulation (Section 3.4) and a greedy algorithm that works well in practice (Section 3.5). Section 3.6 describes our approach to the gate function (node) matching problem. Section 3.7 describes the results of some experiments on the algorithms and presents our conclusions.

3.2 Network Matching

Our aim is to find a node in the old network for each node in the new network, with information we can use for its analysis. This information, by assumption, is only a function of the node and its transitive fanin. Thus, the matching node in the old network must have an identical transitive fanin.

We cannot use the technique of using the simulation signatures of nodes to distinguish them, because we do not have an input correspondence. We identify the set of all potentially matching nodes (called candidate pairs) and combine a compatible subset of these to form the matching. In Section 3.4, we show that the problem of finding the best subset can be reduced to finding a maximal prime compatible. In Section 3.5, we present a greedy algorithm for finding a good subset.

A network (defined in Chapter 2) is characterized by a set of nodes with three associated functions: $\mathrm{func}(n)$ is the *function* of the node, $\mathrm{fanins}(n) \in \{0, 1, \ldots\}$ is the number of *fanins* of the node, and $\mathrm{fanin}(n, k) \in N, k = \{1, \ldots, \mathrm{fanins}(n)\}$ is the kth fanin of the node.

We only consider acyclic networks. Formally, $n \notin \mathrm{tf}(n)$, where $\mathrm{tf}(n)$ denotes the set of nodes in the transitive fanin of n.

Definition 9 *The* **transitive fanin** *of a node n is the set of nodes* $\mathrm{tf}(n) = \cup_{k=1}^{\mathrm{fanins}(n)}(\mathrm{fanin}(n,k) \cup \mathrm{tf}(\mathrm{fanin}(n,k)))$.

The following definition characterizes which nodes we might consider matching. Informally, two nodes could match if their functions are identical and their respective fanins could match.

Definition 10 *A pair of nodes n_1, n_2 is a* **candidate pair** *(denoted $n_1 \sim n_2$) if* $\mathrm{func}(n_1) = \mathrm{func}(n_2)$*,* $\mathrm{fanins}(n_1) = \mathrm{fanins}(n_2)$*, and* $\forall_{k=1,\ldots,\mathrm{fanins}(n_1)}\mathrm{fanin}(n_1,k) \sim \mathrm{fanin}(n_2,k)$*. Note that the correspondence between the fanins is determined by reducing the node function representation to some semi-canonical form, and noting that in that form, the ith variable for (canonical) node function for n* must *correspond with the ith variable for the (canonical) node function for n'.*

This is of course an approximation, since there may be several permutations of fanins where $func(n_1) = func(n_2)$. Note that this definition implies that all primary inputs may match with each other. We add the caveat that the primary inputs may match provided they can take the same set of values, i.e. a primary input that can take values $0, 1, 2$ cannot match with a primary input that takes values $0, 1, 2, 3, 4, 5$.

Not all candidate pairs lead to consistent matchings. Specifically, it may be necessary to match a node in the new network to two or more nodes in the old network simultaneously. This is particularly nonsensical in the case of zero-fanin nodes, which represent inputs to the network. Figure 3.1 depicts a contradictory situation.

Formally, the consistency constraint requires a matching to be a function mapping each node in the new network either to a matched node in the old network, or to "unmatched," represented as $\emptyset$.

Definition 11 *Given two networks N (the new network) and N' (the old network), a* **matching** *is a function $M : N \rightarrow N' \cup \{\emptyset\}$ such that $M(n) \neq \emptyset$ implies ($n \sim M(n)$ and $\forall k = 1, \ldots, \mathrm{fanins}(n)\ .\ M(\mathrm{fanin}(n,k)) = \mathrm{fanin}(M(n),k)$).*

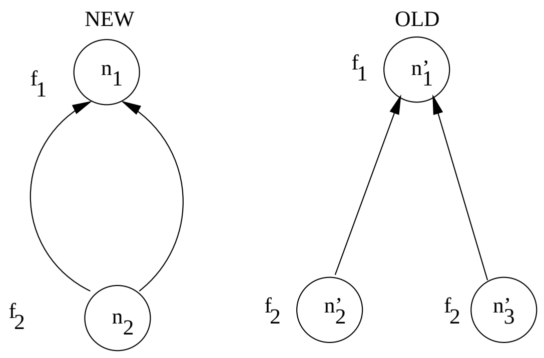


Figure 3.1: A candidate pair $(n_1 \sim n'_1)$ with no consistent matching.

Note: This definition implies that if $M(n) \not= \phi$, then $\forall n_a \in \text{tf}(n), M(n_a) \not= \phi$.

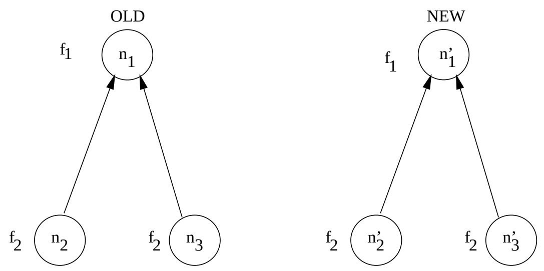


Figure 3.2: A matching with $\text{q}(M) = 3$

Our objective is to find a matching that maximizes the number of matched nodes (called the quality of the match), i.e. those for which $M(n) \not= \phi$.

Definition 12 *The* **quality** *of a matching M is the number of matched nodes, i.e.,* $\text{q}(M) = |\{n \mid M(n) \neq \emptyset\}|$.

Definition 13 *If it exists, the* **implied matching** *of a candidate pair $n_1 \sim n_2$ is*

$$\begin{aligned} M(n_1) &= n_2 \\ \forall_k M(\text{fanin}(n_a, k)) &= \text{fanin}(M(n_a), k),\ \ n_a \in \text{tf}(n_1) \cup \{n_1\} \\ M(n) &= \emptyset,\ \ n \notin \text{tf}(n_1) \end{aligned}$$

Theorem 3.1 *An implied matching is a matching.*

Proof.

1. $\forall k = 1, \ldots, \text{fanins}(n), M(\text{fanin}(n,k)) = \text{fanin}(M(n), k)$.
2. M is a function.

□

We will be combining implied matchings to form bigger matchings, but some pairs of implied matchings—those that map a node in the new network to two different nodes in the old—cannot be combined. We need a formal definition of which matchings can be merged:

Definition 14 *A pair of matchings M_1 and M_2 are* **compatible** *(written $M_1 \rightleftharpoons M_2$) if $(M_1(n) \neq \emptyset) \wedge (M_2(n) \neq \emptyset) \Rightarrow M_1(n) = M_2(n)$.*

Note that compatibility is not transitive; i.e. $M_1 \rightleftharpoons M_2$, and $M_2 \rightleftharpoons M_3$, does not imply that $M_1 \rightleftharpoons M_3$. Let N and N' be the networks in Figure 3.2. M_1, M_2, M_3 are defined as follows.

$$\begin{array}{lll} M_1(n_1) = n'_1 & M_2(n_1) = \phi & M_3(n_1) = n'_1 \\ M_1(n_2) = n'_2 & M_2(n_2) = \phi & M_3(n_2) = n'_3 \\ M_1(n_3) = n'_3 & M_2(n_3) = \phi & M_3(n_3) = n'_2 \end{array}$$

. $M_1 \rightleftharpoons M_2$, and $M_2 \rightleftharpoons M_3$, but $M_1 \not\rightleftharpoons M_3$.

Definition 15 *The* **merge** *of two matchings M_1 and M_2, written $M_1 + M_2$, is the function*

$$(M_1 + M_2)(n) = \begin{cases} M_2(n) & \text{if } M_1(n) = \emptyset \\ M_1(n) & \text{otherwise} \end{cases}$$

Lemma 3.2 *If $M_1 \rightleftharpoons M_2$, then $M_1 + M_2$ is a matching and $M_1 + M_2 = M_2 + M_1$, i.e. merging is commutative. Moreover, if in addition $M_2 \rightleftharpoons M_3$ and $M_1 \rightleftharpoons M_3$, then $(M_1 + M_2) + M_3 = M_1 + (M_2 + M_3)$, i.e. merging is associative.*

Proof. $M_1 \rightleftharpoons M_2 \Leftrightarrow \forall_n (M_1(n) \neq \emptyset) \cdot (M_2(n) \neq \emptyset) \Rightarrow M_1(n) = M_2(n)$.

$$(M_1 + M_2)(n) = \begin{cases} M_2(n) & \text{if } M_1(n) = \emptyset \\ M_1(n) & \text{if } M_1(n) \neq \emptyset \end{cases}$$

$$(M_2 + M_1)(n) = \begin{cases} M_1(n) & \text{if } M_2(n) = \emptyset \\ M_2(n) & \text{if } M_2(n) \neq \emptyset \end{cases}$$

1. if $M_1 \neq \emptyset$, $M_2 \neq \emptyset$.
   $\Rightarrow M_1 + M_2 = M_1 = M_2 = M_2 + M_1$.
2. if $M_1 = \emptyset$, $M_2 \neq \emptyset$.
   $\Rightarrow M_1 + M_2 = M_2 = M_2 + M_1$.
3. if $M_1 \neq \emptyset$, $M_2 = \emptyset$.
   $\Rightarrow M_1 + M_2 = M_1 = M_2 + M_1$.
4. if $M_1 = \emptyset$, $M_2 = \emptyset$.
   $\Rightarrow M_1 + M_2 = \emptyset = M_2 + M_1$.

$\Rightarrow M_1 + M_2 = M_2 + M_1$. Associativity proved in a similar manner, i.e. by enumerating all possibilities. □

Lemma 3.3 *Merging only improves quality, i.e., if $M_1 \rightleftharpoons M_2$, then* $\mathrm{q}(M_1), q(M_2) \leq q(M_1 + M_2)$.

Proof. Assume not.
$\Rightarrow \exists n \ st \ (M_1(n) \neq \emptyset) \cdot (M_1 + M_2)(n) = \emptyset$.
$M_1(n) \neq \emptyset \Rightarrow (M_1 + M_2)(n) = M_1(n) \neq \emptyset$.
$\Rightarrow (M_1 + M_2)(n) \neq \emptyset$.
A contradiction, hence $\nexists n \ st \ (M_1(n) \neq \emptyset) \cdot ((M_1 + M_2)(n) = \emptyset$.
$\Rightarrow q(M_1) \leq q(M_1 + M_2)$. □

```
Partition nodes in both networks by function
Refine this partition s.t. all nodes in a bucket have fanins in the same buckets
Form all candidate pairs by considering all pairs of nodes in each bucket
Sort the candidate pairs by the number of nodes in their transitive fanin
```

Figure 3.3: Identifying compatible nodes.

3.3 Determining Matchings: A Refinement Algorithm

In order to determine the entire set of implied matchings, we use the following iterative algorithm. We begin by assuming all nodes whose node functions are matched to be matched. We implement this algorithm with a hash table. Nodes with the same node function are put into the same initial "bucket" in the hash table. The canonical form of the node function imposes a certain order on the fanins of the node. If two node functions in canonical form are equal, then the fanins node corresponding to ith variable of the node function, must correspond. We refine the node matchings iteratively, by "un-matching" two nodes, if some of their corresponding fanins are un-matched. We accomplish this by re-bucketing each node in the hash table. At each iteration, the new bucket signature of a node consists of its table signature (canonical form) and the bucket numbers of its fanins (in the order imposed by their node function tables). Thus, if at some iteration, any nodes in the same bucket have corresponding fanins in different buckets, then after that iteration, these nodes get put into different buckets.

This algorithm is similar to the algorithm for the computation of equivalent states in an FSM [51], [45]. After this refinement, all pairs of nodes in a bucket are candidates. The algorithm is shown in Figure 3.3.

Though we have described a procedure that matches entire cones, this procedure can be modified to match sub-regions by restricting the number of iterations of the refinement procedure, or keeping track of all buckets seen during the refinement process.

3.4 An Exact Formulation

Once we have a set of consistent matchings (Section 3.3), we address the problem of finding a maximum compatible matching exactly.

Lemma 3.3 indicates that merging compatible matchings gives higher quality matchings. In this section, we use this idea to exactly characterize the problem of finding the maximal quality matching. We show that the maximal matching is a "prime" matching—one for which merging in other matchings is either impossible or unproductive.

Lemma 3.4 *If M is the sum of a finite number of compatible implied matchings then it is a matching, i.e., $\forall_{i,j} M_i \rightleftharpoons M_j$ and $M = M_1 + M_2 + \cdots + M_k \Rightarrow M$ is a matching .*

Proof. Follows from the definition of matching, implied matching, and Lemmas 3.2. □

We can define a dominance relation [52] [53] [54] as follows:

Definition 16 *A matching M_1* **dominates** *a matching M_2 (written $M_1 \geq M_2$) if $M_1 \rightleftharpoons M_2$ and $M_1 + M_2 = M_1$.*

Definition 17 *A* **prime** *matching is one that is not dominated by any other matching.*

Lemma 3.5 *If M_1 is a prime matching, and $M_1 \geq M_2$, then $\mathrm{q}(M_1) \geq \mathrm{q}(M_2))$.*

Proof. Since $M_1 \geq M_2$, $M_1 \rightleftharpoons M_2 \Rightarrow M_1 = M_1 + M_2$.
Lemma 3.3 implies $\mathrm{q}(M_2) \leq \mathrm{q}(M_1 + M_2)$. Since $M_1 + M_2 = M_1$, it follows that $\mathrm{q}(M_2) \leq \mathrm{q}(M_1)$. □

We can reduce maximal or prime matching to a prime generation problem in the following manner.

1. Associate a Boolean variable u_i with each matching M_i. $u_i = 1$ implies M_i is part of the given matching.

2. For each pair of matchings M_i and M_j that are not compatible $M_i \not\Leftarrow M_j$, construct a clause $(\overline{u_i} + \overline{u_j})$. This means either M_i must not be in the partition or M_j must not be in the partition.

3. logically AND all such clauses to get a function $f(u)$.

4. A prime of function $f(u)$ corresponds to a compatible set of matchings. The maximal prime corresponds to a maximal matching.

Theorem 3.6 *A maximum matching is a prime matching and can be built from a set of compatible implied matchings.*

Proof. Follows from Lemmas 3.4 and 3.5. □

Thus, from the above the problem of finding the maximum matching is one of finding the maximum quality prime. We can do this naively by enumerating each prime matching and calculating its quality (in actuality, we implement a slightly more efficient procedure). However, since the number of primes of a set of n elements is $O(3^n/n)$ [55] and n can be $O(N^2)$, where N is the number of nodes in each network, it is often impractical to explicitly search the entire set of primes. This worst case comes when the network consists of a set of zero-fanin nodes with identical functions.

3.5 A Greedy Algorithm

The exact method cannot handle large examples; we extend the scope of the examples by using the following heuristic algorithm. Our heuristic algorithm finds the set of all candidate pairs with implied matchings and merges them greedily, trying the highest quality ones first.

First we used the refinement procedure of Section 3.3 to identify candidate pairs. Once the candidate pairs are identified, we build a matching by merging together compatible implied matchings. We consider candidate pairs one at a time, starting with those with the largest number of nodes in their transitive fanins, and "grow" a matching by merging each compatible implied matching.

```
Partition nodes in both networks by function
Refine this partition s.t. all nodes in a bucket have fanins in the same buckets
Form all candidate pairs by considering all pairs of nodes in each bucket
Sort the candidate pairs by the number of nodes in their transitive fanin
$M(n) = \emptyset$, the empty matching
for $M_i$ largest to $M_i$ smallest
    if $M \rightleftharpoons M_i$
        $M = M + M_i$
RETURN$M$
```

Figure 3.4: The greedy matching algorithm.

The entire algorithm is shown in Figure 3.4. The performance of our implementation of this algorithm on example circuits is discussed in Section 3.7.

3.6 Table Matching: Matching Node Functions

In this section, we discuss how to identify whether two node functions are identical if we do not have an input correspondence. This is known as *Boolean matching*, and is a well studied problem. For our experiment, we are looking for a quick estimator of whether two node functions, represented as *node function tables* match.

The nodes in our networks have discrete-valued functions (a generalization of Boolean functions) associated with them. These are represented in BLIF-MV-style tables [48], such as that in Figure 3.5. Each column on the left represents an input variable, and each row is a pattern that, when the inputs match it, produces the output in the rightmost column. Each entry is either a single value (e.g., 3), a set of values (e.g., 1, 2, 5), or the set of all values (i.e., "–"). Note that BLIF-MV permits symbolic values of the form *red*, *blue*, *greeen*, which are represented as the values $0, 1, 2$.

Figure 3.5 represents a function $f(x_1, x_2, x_3)$ that is 3 when $x_1 = 0$ and $x_2 = 2$ or 3, or when $x_2 = 1$; is 0 when $x_1 = 1$, $x_2 = 0$, and $x_3 = 1$; and is 1 default.

We want to be able to quickly identify tables that compute the same function. Transform-

x_1	x_2	x_3	f
0	2,3	–	3
–	1	–	3
1	0	1	0
default			1

Figure 3.5: A multi-valued table. x_1, x_2, and x_3 are the input variables.

ing each table into a *permutation-invariant canonical form* is an approximate approach to solving this problem; different tables that are not equivalent modulo permutations may also compute the same function. Computing a canonical form (modulo all permutations) is much more expensive([47]); in the interests of quick computation, we have opted for this simpler semi-canonical form.

Definition 18 *Two tables are* **permutation equivalent** *if one can transformed to the other by permuting the rows and columns.*

We assume that the values in each entry are always ordered, so that we do not have to distinguish between $2, 3$ and $3, 2$. To make this entry compact, we use ordered lists of ranges, i.e. $2 - 5, 7 - 8$, to represent each entry.

Definition 19 *A function is* **canonicalizing** *iff it maps all permutation-equivalent tables to a single table, which is called the* **permutation-invariant canonical form** *of the table.*

A function is canonicalizing if it imposes a permutation-invariant total order on rows and columns and then sorts the rows and columns based on this. Finding such a total order is difficult and expensive, so we resort to an order that is partial for certain tables. We count the number of times a particular value appears in the entries in a row or column and order the rows and columns based on this sum. The reason we use this "addition" of the number of times a value occurs in a column as a hash function is because we need a permutation invariant canonical form.

Consider the table in Figure 3.6. If we order the rows and columns according the number of 1's that appear in each row and column, we obtain the table in Figure 3.7. We were

$$\begin{array}{cccccc} & & & & & \sum = \\ & (& 1 & 1 & 0 &) \quad 2 \\ & (& 1 & 1 & 1 &) \quad 3 \\ & (& 1 & 0 & 0 &) \quad 1 \\ \sum = & & 3 & 2 & 1 & \end{array}$$

Figure 3.6: A simple table annotated with the number of 1's in each row and column.

fortunate in this example, since the number of 1's in each row and column is different, but in general, this strategy only produces semi-canonical tables.

$$\begin{array}{cccccc} & & & & & \sum = \\ & (& 0 & 0 & 1 &) \quad 1 \\ & (& 0 & 1 & 1 &) \quad 2 \\ & (& 1 & 1 & 1 &) \quad 3 \\ \sum = & & 1 & 2 & 3 & \end{array}$$

Figure 3.7: The table in canonical form

We can extend these ideas to tables with set-valued entries by converting each entry to an integer. First, each set is transformed to a vector of 0's and 1's. Each 1 represents the presence of a value in the set; each 0 represents the absence, e.g., the entry $2, 3$ would be represented as a vector (1100). A bitwise sum of all such vectors in a row or column (zero-extending them if necessary) gives a vector than can be used to impose a partial order. E.g. The bitwise sum of $(2, 3) = (1100)$ and $(0, 1, 2) = (0111)$ is (1211). (1211) denotes that in the given column there is one 0 value, one 1 value, two 2 values and one 3 value.

These vectors can be transformed to integers to make them easier to manipulate.

Intuition

Note that in a table with n rows and m columns, the total number of 1's in a position in a column cannot exceed n. Similarly, the total number of 1's in a row cannot exceed m. By transforming these vectors to base $b = \max\{m, n\} + 1$ integers, we can sum the integers in

$$\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

Figure 3.8: Identical tables

a row or column, and still ensure that each column sum only includes information about that column (*no carry between (value) positions*). For example, if each entry in a column is the entry $2 = (0100)$, and there are 15 columns. The bitwise sum for the column is $0F00$; F denotes 15 in base 16. If we were to represent the number in base 10, then the sum would be (1500), and due the carry we cannot distinguish between fifteen 2 entries versus one 3 and five 2 entries. Under this representation permutation equivalent rows or columns have the same sum. This may result in some ambiguity. Consider the two tables shown in Figure 3.8; both rows of the given tables have the same sum, and hence are indistinguishable. If this ambiguity is never resolved, then these two rows will never be interchanged. Thus, the fact that the two tables are identical will not be detected. This issue can be resolved by using a secondary tie breaker like the position of the first 1 entry. In general, this problem is part of a larger problem of "symmetries" [56].

Definition 20 *For a table with n rows and m columns, let m_j be the maximum value of the input variable in column j, and let $E_{ij}(k)$ be 1 if the entry in row i and column j contains the value k and 0 otherwise. The numerical representation of this table is an $n \times m$ matrix T with entries*

$$t_{ij} = \sum_{k=0}^{m_j} b^k E_{ij}(k)$$

It is clear that each subset of values at a table entry has unique encoding t_{ij}. Figure 3.9 shows the table of Figure 3.5 converted to a matrix of natural numbers. For this table, $(1 + \max\{m, n\}) = 4$. As an example, the entry $2, 3$ is converted to a base four number: $t_{1,2} = 4^0 \cdot 0 + 4^1 \cdot 0 + 4^2 \cdot 1 + 4^3 \cdot 1 = 80$.

$$\begin{array}{cc} & \sum = \\ \begin{array}{c} \\ \\ \\ \sum = \end{array} \begin{array}{c} \begin{pmatrix} 1 & 80 & 5 \\ 5 & 4 & 5 \\ 4 & 1 & 4 \end{pmatrix} \\ \begin{array}{ccc} 10 & 85 & 14 \end{array} \end{array} & \begin{array}{c} 86 \\ 14 \\ 9 \\ \\ \end{array} \end{array}$$

Figure 3.9: The table converted to a matrix of natural numbers.

$$\begin{pmatrix} 4 & 4 & 1 \\ 5 & 5 & 4 \\ 1 & 5 & 80 \end{pmatrix}$$

Figure 3.10: The table in semi-canonical form

Definition 21 *In an $m \times n$ table (t_{ij}), a row i is* **before** *row k if $\sum_{j=1}^{n} t_{ij} < \sum_{j=1}^{n} t_{kj}$. A column j is* **before** *a column k if $\sum_{i=1}^{m} t_{ij} < \sum_{i=1}^{m} t_{ik}$.*

Definition 22 *The* **semi-canonical form** *of a table t_{ij} is a permutation of the rows and columns of t_{ij} such that if row i is before row k then $i < k$, and if column j is before column k then $j < k$.*

Figure 3.10 shows the table in Figure 3.9 converted to semi-canonical form.

Theorem 3.7 *A table in semi-canonical form represents the same function as the original table under some permutation of variables.*

Hence two tables with the same semi-canonical form represent the same discrete function.

3.7 Experiments and Results

We have implemented the algorithms described in the VIS [6] environment.

In order to to test our procedure, we designed the following experiment. We assume that the design has been read in, and the designer has computed the output function BDDs of each node (as functions of the primary inputs). At this point the designer modifies the original design by either changing the functionality, or just re-optimizing the hardware for some other objective. The designer would like to use the BDDs computed for the old network to efficiently compute the BDDs in the new network. Obviously, we assume that there is a sufficient amount of structural similarity between the old and the network design.

To emulate a design change, we took MCNC, ISCAS and VIS benchmark examples and modified them to obtain a circuit called "new". The original benchmark spec corresponds to the "old" design.

As an experiment we built the function BDDs associated with the "old" design. This is done recursively, by building the BDD at each node as a function of the BDDs of its fanin nodes. Next, we ran the matching algorithm on the old and new designs. If there existed a match from a node in the new network, to the old, we re-used the BDD for the old node by merely substituting the old network BDD variables with the corresponding BDD variables in the new network. If there was no match, we re-computed the BDD by using the BDDs computed for the fanin nodes of the new node. We reported time for this incremental computation (Inc Time) as well as the time for computing the matching (Total Match Time). We also built the BDDs for the new network from scratch, and reported this non-incremental time (Non-Inc Time).

Tables 3.1 and 3.2 report the quality of the matching Vs. the time to match the examples. Columns 2 and 3 list the total and matched number of nodes in the network respectively. The matching times are listed by its component; i.e. time to get the initial matching(Match Initial Time), time to refine the partition(Match Refine Time), and time to generate matching in Column 4, 5 respectively, as well as the total time to match (Total Match Time = initial + refine +time to generate and evaluate the quality of the entire matching cones), in Column 6. Since we used an explicit matching algorithm, it is rightly observed that as the size of the matching increases so does the time to match. The dominant portion of the time appears to be spent in generating the matching rather than the refinement or initial time.

Tables 3.3 and 3.4 report the times for the non-incremental BDD computation (Column

Example	♯ Nodes Total	♯ Nodes in Match	Match Initial Time	Match Refine Time	Total Match Time
alu2	429	6	0.067	0.15	0.217
alu4	126	126	0.117	0.05	0.184
apex6	1031	37	0.083	0.1	0.2
apex7	330	12	0.033	0.017	0.05
bigkey	1369	791	0.317	0.033	1.567
c8	182	15	0.017	0.017	0.034
clma	11382	10973	4.766	3.534	11.783
clmb	10842	10407	4.634	3.416	10.45
cm163a	68	11	0.017	0	0.017
cordic_latches	3468	2873	0.35	0.267	1.617
dalu	1206	1206	0.183	0.083	0.716
des	1182	1174	0.933	0.35	1.95
dsip	4554	3920	0.384	0.283	20.884
i10	2754	2750	0.284	0.6	1.734
i2	364	48	0.067	0.016	0.083
key	1604	980	0.367	0.1	1.967
mark1	133	18	0	0.016	0.016
minmax10	723	87	0.033	0.117	0.15
minmax12	914	104	0.066	0.15	0.233
mm9a	830	637	0.05	0.05	0.316
mm9b	714	106	0.067	0.083	0.167
mult32b	665	253	0.05	0.017	0.084
pair	2141	77	0.217	0.183	0.45
rot	1038	240	0.1	0.117	0.25
s1196	816	126	0.083	0.083	0.2
s1238	847	79	0.1	0.067	0.184
s13207	10065	8713	0.75	1.333	18.583
s1423	1199	298	0.1	0.083	0.317
s1488	711	97	0.084	0.083	0.184
s1494	658	34	0.083	0.083	0.183
s15850	11591	10272	0.933	1.684	12.183
s38584	23775	20839	5.767	7.267	138.434
s9234	6266	5844	0.616	0.85	8.483
term1	257	62	0.033	0.017	0.05

Table 3.1: Quality and Time to Match

Example	♯ Nodes Total	♯ Nodes in Match	Match Initial Time	Match Refine Time	Total Match Time
arbiter	248	194	0.05	0.017	0.084
bakery	415	383	0.1	0.066	0.216
coherence	880	814	0.3	0.2	0.683
counter	58	52	0.017	0	0.017
ctlp3	127	111	0.017	0.016	0.033
dcnew	320	296	0.066	0.034	0.133
eisenberg	400	376	0.134	0.083	0.267
elevator	1327	1227	0.55	0.15	0.95
gigamax	569	549	0.083	0.083	0.233
ping_pong_new	118	110	0.017	0.016	0.05
scheduler	814	774	0.183	0.1	0.583
slider	274	256	0.033	0.017	0.117
tcp	1668	1592	0.567	0.416	2.017

Table 3.2: Multivalued Examples: Quality and Time to Match

Example	Non-Inc Time	Inc Time	Total Match Time	Total (Match+Inc) Time	Matched Nodes	Total Nodes
bigkey	1	0.183	1.65	1.883	791	1369
cordic_latches	2.367	0.066	1.7	1.766	2873	3468
clma	11.6	0.8	11.78	12.68	10973	11382
clmb	11.45	0.8	10.45	11.25	10407	10842
des	2.884	0.017	1.967	1.984	1174	1182
i10	13.334	0.067	1.867	1.934	2750	2750
minmax10	800.734	0.2	0.35	0.55	87	723
minmax12	352.634	0.25	0.467	0.717	104	914
mm9a	27.034	0.033	0.35	0.383	637	830
mm9b	526.0	0.2	0.367	0.567	106	714
pair	1.434	0.884	0.466	1.35	77	2141
s13207	1.6	0.217	18.734	18.941	8713	10065
s1423	1.783	0.133	0.317	0.315	298	1199
s15850	31.617	0.267	12.317	12.584	10272	11591
s38584	10.85	1.35	138.434	139.784	20839	23775

Table 3.3: Incremental Vs. Non_Incremental Update

Example	Non-Inc Time	Inc Time	Total Match Time	Total (Match+Inc) Time	Matched Nodes	Total Nodes
arbiter	0.067	0	0.084	0.084	194	248
bakery	0.15	0.016	0.216	0.232	383	415
coherence	0.5	0.016	0.683	0.699	814	880
counter	0.016	0	0.017	0.017	52	58
ctlp3	0.034	0	0.033	0.033	111	127
dcnew	0.117	0.016	0.133	0.149	296	320
eisenberg	0.2	0.017	0.267	0.284	376	400
elevator	0.35	0.033	0.95	0.983	1227	1327
gigamax	0.45	0.017	0.233	0.25	549	569
ping_pong_new	0.034	0	0.05	0.05	110	118
scheduler	0.267	0.033	0.583	0.616	774	814
slider	0.15	0	0.117	0.117	256	274
tcp	1.116	0.05	2.017	2.067	1592	1668

Table 3.4: Multivalued Examples: Incremental Vs. Non_Incremental Update

2) Vs. the incremental BDD computation (Column 3) and total matching time (Column 4). The times for incremental BDD computation alone were always better than the non-incremental time (obviously using previously computed information is better than no information). However, when we add in the matching time, this is not always the case.

Of the reported example (we only considered those with more than 1 sec of CPU time), most have significantly better total times for the incremental procedure (match time + incremental time) as compared to the non-incremental procedure. Only 2 had significantly worse time for the incremental method, 2 had approximately equal times and the rest always reported better times (incremental + matching) for the incremental method. We report the results on some small multi-valued examples to illustrate that the procedure does indeed work similarly for multi-valued examples. However, the times on these examples is too small for any conclusions. We also report the results on the exact computation (Section 3.4)as compared to the heuristic (Section 3.5). The exact method ran out of memory much faster, and hence we were only able to deal with small examples with the exact method. However, Table 3.5 shows that for examples where the exact method could complete, the heuristic answers were almost always the same.

Example	Heuristic ♯ Nodes in Matching	Exact ♯ Nodes in Matching
apex7	12	12
bbsse	23	23
c8	15	16
cm163a	11	11
i2	48	48
mark1	18	18
minmax10	87	87
minmax12	104	104
mult32b	253	253
term1	62	62

Table 3.5: Exact Vs. Heuristic Common Substructures

We have implemented the matching algorithms, and demonstrated that on BDD building the incremental procedures take less time than the non-incremental. We can conclude that on an average the incremental procedure should less time than the non-incremental. We have also implemented the exact matching, and shown that for small examples the exact answer is almost identical to our heuristic. This demonstrates the effectiveness of our heuristic.

We examined the one example where the matching time far exceeded the non-incremental time, and found that the cause of this problem was the large symmetry in the circuit coupled with the large size of the circuit. There were many possible matchings, and examining them all, while determining the qualities of matchings was expensive. As part of future work, the work of Malik [56] to detect symmetries could be used to speed up our computation. We found that as we increased the size of the example, the matching time increased significantly. This is due to our explicit formulation of the matching algorithm. As future work an implicit formulation of the matching algorithms can used to overcome some of the size limitations.

Our techniques could be extended to deal with matching arbitrary sections of the network, rather than the entire transitive fanin cone. One application would be finding structurally identical sections within a single network, so that information computed at one section may be re-used for another structurally identical portion.

Chapter 4

Incremental FSM Traversal

4.1 Introduction

Reachability is an essential computation in both formal verification [14] and sequential synthesis [57] & [58]. Given a directed graph, and a set of initial nodes in the graph, any node such that there is a path from the initial nodes to it, is denoted as reachable. A *finite state machine* (or FSM) can be represented by a directed graph, which is also called a *state transition graph*. Computing the reachable states (nodes) of this graph is accomplished by any combination of breadth first search (BFS) or depth first search (DFS) [31] exploration of the state transition graph beginning at the initial states.

Unfortunately, this computation explodes when the number of states in the finite state machine becomes very large. This is often called the *state explosion problem*. To overcome this problem, an implicit representation called a binary decision diagram or BDD [40], is sometimes used to represent all the required quantities. e.g. the transition relation, which implicitly represents the FSM's state transition graph, and any set of states (initial, reachable etc.). When BDD's are used, a BFS traversal of the state transition graph is more convenient, and the steps in this BFS traversal can be written as fixed point computations of propositional formulae on the transition relation, initial states etc ([45]).

The process of design is iterative, and the designer may modify the design many times. The current techniques for reachability require that each time the designer modifies the

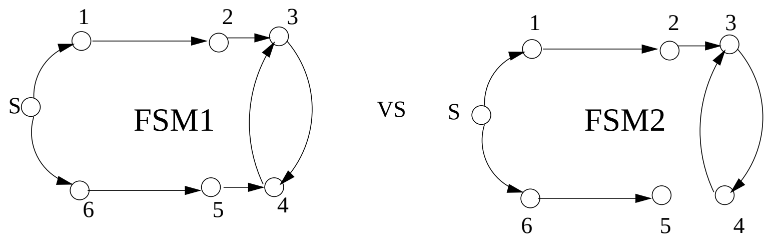


Figure 4.1: Reachability is non-incremental

design, the set of reachable states must be re-computed from the beginning. This results in unnecessary re-computation, which is particularly cumbersome in light of the state explosion problem. Instead, it is preferable if the set of reachable states can be updated incrementally at each iteration of the design process.

This chapter deals with the construction of such incremental algorithms for reachability. The complete reachability analysis is executed only once, and all successive changes are propagated from the previous iteration. We note that knowing only the set of reachable states is not sufficient for updating the reachable set. This can be understood by considering the two examples in figure 4.1. Both have identical reachable states$(1, 2, 3, 4, 5)$, but if edge $(2, 3)$ is deleted, then FSM_1 has a different set of reachable states from FSM_2. This is due to the presence or absence of edge $(5, 4)$, and if no traversal information is stored, then this cannot be determined without examining the entire state space of the two FSM's.

We overcome this problem by storing a reached state relation instead to represent the reachable states, instead of the set of reached states. The spanning tree, or graph that can be generated by any BFS type procedure for reachability are valid reached state relations. We update this relation after every change. For example, refer to the FSM in Figure 4.2. Usual reachability algorithms just store 0-1 reachable information, i.e. whether or not the state is reachable. We store the spanning tree of edges traversed during FSM traversal. We use information about the changes made to the system, and the original spanning graph of the reachable states, to compute a new spanning tree, which represents the new reachable states.

In this chapter we will examine three different alternate representations for the reached state set and traversal information.

A more succinct version of this work can be found in [50].

4.2 Terminology for FSM Traversal

We have defined a *Finite State Machine*, its *states, inputs, outputs*, *transition relation* $T(x, i, o, y)$, and *initial states* $I(x)$ in Chapter 2. Using those definitions, we shall discuss how to compute the reachable states of the finite state machine in this section.

Definition 23 Reachable states: *The set of reachable states is denoted by R, $q \in R$ if and only if there is a path from some initial state $q_0 \in I$ (the set of initial states) to q.*

Since, we are only concerned with reachable behavior in this chapter, we will be using the transition relation after removing input and output dependencies. This will be referred to as $T(x, y)$, where x and y are present state and next state variables respectively. Note that $T(x, y) = \exists_{(\sigma,\gamma)} T(x, y, \sigma, \gamma)$.

Let $R(x)$ denote the reachable states, and $I(x)$ denote the initial states. If fixed points FP are defined as in Chapter 2, then the set of reachable states is computed as the $LFP(f(Q), I)$ of $f(Q(x)) = Q(x) + \exists_y T(y, x) \cdot Q(y)$, given $I(x)$ the initial states. The set of successors $(\exists_y T(y, x) \cdot Q(y))$ of any given set of states is called its *image*, and image computation is the key step in this fixed point computation.

Recollect that this entire set of computations will be done in the context of a system of interacting finite state machines, which may also be represented as a *product machine* (defined in Chapter 2)

Our incremental algorithms will use Theorem 2.2 about fixed points; i.e. for an LFP computation, any subset of the final answer that contains the initial set, returns the correct final answer, when supplied to the LFP as a starting point. A similar statement can be made about GFP computations, as shown in Chapter 5 and 6.

4.3 FSM Traversal

Computing the set of reachable states in the transition relation of a finite state machine is equivalent to doing a traversal of the state transition graph, beginning at the initial states. This traversal may be breadth first, or depth first.

Touati et-al [45] and Burch et-al [8] independently extended this concept to handle reachability in larger systems, by using partitioned implicit methods. All quantities (transition relations, sets of states etc.) are represented by BDD's (binary decision diagrams), and the algorithm is represented by a fixed point computation (refer to Chapter 2).

```
**Algorithm 4.1** $(T(x,y), I(x))$
    $f(Q(x)) = \exists_y T(y,x)Q(y) + Q(x)$
    $R(x) = LFP(f(Q), I(x))$
  **return** $R(x)$
```

Using BDD's the complexity of this algorithm is $O(N^2)$, where N is the number of states. Each image step takes $O(N)$ (the size of the BDD's involved), and since at each step of the fixed point, at least one state must be added, there are at most N steps involved. If the algorithm uses BDDs, this analysis is somewhat meaningless in practice.

Unfortunately, this algorithm is not incremental, and if the designer modifies the system, the reachable states have to be computed from the beginning.

4.4 Incremental Algorithms for Reachability

Let $R(x)$ is a set of reachable states in the system. We want to use information about the changes to the system to incrementally modify $R(x)$. The potential for speedup is that $R(x)$ need not be recomputed from the beginning; intermediate results can be used to avoid unnecessary computations.

Unfortunately, as shown in the example in Figure 4.1 (Section 4.1), just the old set of reachable states is not sufficient to update information. However, the traversal tree that is generated during the reachability computation or a variant of it, is sufficient to update reachability information. Thus, we overcome the aforementioned problem by storing a variant of the traversal tree that can be generated during reachability computations. We call this variant the reached state relation ($P(x, y)$).

Let $P(x, y)$ be any acyclic relation (graph) such that $\exists_y(P(x, y) + P(y, x)) = R(x)$. If $P(x, y)$ is a tree we say that P is a reached state tree.

We implement three different reached state relations. The first chooses P to be the spanning tree that is obtained by retaining only one of the many edges traversed to reach a state from one of its neighbors. The second chooses P to be a spanning graph that is a subset of the transition relation. The third computes, and stores the acyclic transitive closure of the state transition graph, which can also be obtained during normal reachability analysis. Figure 4.2 shows the spanning tree, spanning graph and acyclic transitive closure for a given transition structure. Thus, instead of storing the reachable states $1, 2, 3, 4, 5$, we store the tree $(1, 2), (2, 3), (3, 4), (1, 5)$, the graph $(1, 2), (2, 3), (3, 4), (1, 5)$, or the transitive closure $(1, 2), (2, 3), (3, 4), (1, 5), (1, 3), \ldots$.

Once the designer changes the system, the current $P(x, y)$ is modified using information about the changes made to the system and this process is repeated as often as the system changes.

4.4.1 Characterizing Incremental Changes

There are four different incremental changes to an instance of reachability. Briefly, changes to the system may consist of 1) addition or subtraction of edges to the transition relation, and 2) addition or subtraction of states (and hence edges) to the state space of the machine. Addition and subtraction of states can be characterized in terms of edges. Removing a state from the state space is equivalent (behaviorally) to removing all edges to the state, thus making it unreachable. Similarly, if a state is added to the state space, it is similar to making one of the unreachable states in the state space reachable by adding edges.

Thus, we consider only two types of incremental change: addition and subtraction of edges. For each type we first deal with a set of changes of the same type, and then we provide a general incremental algorithm to handle a complex change with many individual types. The algorithms are given in terms of implicit BDD operations.

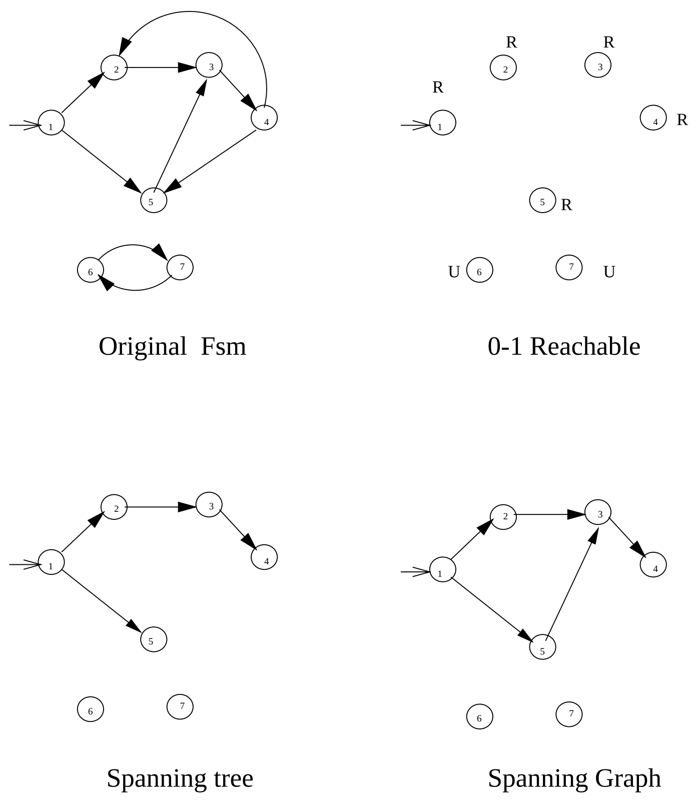


Figure 4.2: Different reached state relations

Actual modifications can be made to the system at many levels; the designer may input the changes in a high-level language like Verilog. Alternately the internal data-structures of the algorithm can be directly modified. These high-level changes must be translated to the addition and subtraction of edges from the FSM. In our system, the designer is allowed to directly modify the component transition relations, or to input new processes to the system.

The designer modifies the original transition relation T to a new transition relation T^{new}. Using T^{new} and T, we create two sets: $\triangle T^{sub}$ and $\triangle T^{add}$. $\triangle T^{sub}$ consists of all deleted transitions, which were removed in T^{new} and $\triangle T^{add}$ consists of all transitions added in T^{new}.

4.4.1.1 Modifications to the System

The designer modifies the original transition relation T to a new transition relation T^{new} by adding and subtracting edges from the transition structure. In practice, this may be done in following ways:

1. **Directly Modification**

   The designer might choose to directly modify the transition structure of the original system. Let $\triangle T^{add}$ and $\triangle T^{sub}$ represent the set of edges, which are to be added and subtracted respectively. The corresponding T^{add} and T^{sub} can be computed by using the following equations.

$$\begin{aligned} T^{sub} &= T \cap \overline{\triangle T^{sub}} \\ T^{add} &= T^{sub} \cup \triangle T^{add} \end{aligned}$$

   If T were a partitioned transition relation, then instead of a single edge deletion/addition, a set of edges are deleted/added from each partition component.

$$\begin{aligned} T_1^{sub} \ldots T_m^{sub} &= \prod_i T_i \cap \overline{\triangle T_i^{sub}} \\ T_1^{add} \ldots T_m^{add} &= \prod T_i^{sub} \cup \triangle T_i^{add} \end{aligned}$$

   A designer can also modify the system by adding or subtracting processes from the system of interacting processes.

2. **Add Processes**

   T_a is the product transition relation of the added processes, T^{old} is the transition relation of the old system, and T^{new} is the new transition relation of the augmented system. R_a is the set of states within the transition structure T_a that are reachable from its initial states $I_a(x)$, and can be computed as $R_a = LFP(f_a(Q), I_a(x))$, where $f_a(Q(x)) = \exists_x T_a(y,x) \cdot Q(y) + Q(x)$. In order to compute T^{add} and T^{sub} in this framework the state space of the original transition relation must be expanded to account for the new processes, and following equations may be used:

$$\begin{aligned}
T &= T^{old} \times R_a(x) \times R_a(y) \\
\triangle T^{sub} &= T \cap \overline{T^{new}} \\
\triangle T^{add} &= 0 \\
T^{sub} &= T^{new} \\
T^{add} &= T^{new}
\end{aligned}$$

This set of equations essentially augments the old T^{old} to the product space of the Cartesian product $T^{old} \times T_a$.

If T were a partitioned transition relation, then this analysis can be extended as follows:

$$\begin{aligned}
T_1^{new} \ldots T_{m+1}^{new} &= T_1^{old} \ldots T_m^{old} \cdot T_a \\
T_1 \ldots T_{m+1} &= T_1^{old} \ldots T_m^{old} \times R_a(x) \times R_a(y) \\
\triangle T_1^{sub} \ldots T_m^{sub} &= T_1^{old} \ldots T_m^{old}(R_a(x) \times R_a(y) \cap \overline{T_a}) \\
\triangle T^{add} &= 0 \\
T_1^{sub} \ldots T_m^{sub} &= T_1^{old} \ldots T_m^{old} \times T_a \\
T_1^{add} \ldots T_m^{add} &= T_1^{old} \ldots T_m^{old} \times T_a
\end{aligned}$$

3. **Subtract Processes**

If processes $T_s(x_s, y_s)$ are subtracted from the system, and T^{old}, T^{new} are as before, then the system can be shrunk to account for these subtracted processes. Quantification will reduce the state space to that of the new system, while preserving the reached state set.

$$\begin{aligned}
T &= \exists_{x_s,y_s} T^{old}(x, y) \\
\triangle T^{add} &= T^{new} \cap \overline{T} \\
\triangle T^{sub} &= 0 \\
T^{sub} &= T \\
T^{add} &= T
\end{aligned}$$

This removes the state space of the deleted processes from the product.

Again, if T were a partitioned transition relation, then this analysis can be extended as follows:

$$\begin{aligned}
T = T_1 \ldots T_m &= \exists_{x_s,y_s} T_1^{old} \ldots T_m^{old} \\
\triangle T^{sub} &= 0 \\
\triangle T^{add} &= T^{new} \cap \overline{T} \\
T^{add} &= T \\
&= \exists_{x_s,y_s} T_1^{old} \ldots T_m^{old}
\end{aligned}$$

We do not have to do the quantification $\exists_{x_s,y_s}$ until the image computation step, and hence we can treat x_s, y_s effectively as inputs. This enables to use partial product methods on such a system.

We have already discussed how to extract the transition relation T^{new} from the new description of the system in Chapter 3. Having once identified and reduced changes to the addition and subtraction of edges from the finite state machine, we will present incremental algorithms that update the reached state relation.

4.4.2 Spanning Tree Algorithm

In this section we deal with an incremental algorithm, which chooses $P(x, y)$ to be a spanning tree that can generated during the course of reachability computations.

4.4.2.1 Computing the spanning tree

The implicit reachability algorithm described in Section 4.3, begins with a current set of the initial states of the FSM. At each stage the image of the current set is computed and added to it. Computing the image of the current set, involves computing the edges of the FSM that begin at any state in the current set, and terminate at any state of the FSM. This is part of a BFS traversal of the state transition graph. During this BFS procedure we choose to select only one of the many edges that terminate at a given state. This returns a spanning tree graph that spans all the reachable states of this FSM. We denote this spanning tree

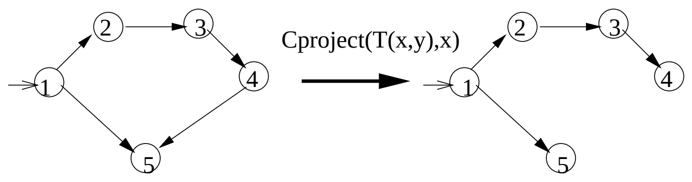


Figure 4.3: Using Cproject

as *rs-spanning tree*. Note that a *tree* must be contiguous. In order to decide which edge to choose as the representative edge, any selector function like "cproject" may be used.

Definition 24 Cproject Operator[59]: *The cproject project operator can be used to extract a tree subset graph of an acyclic graph. The cproject operator is a selection operator, which when given a relation $T(x,y)$, and a reference vertex $\alpha(y) = \alpha_1(y_1), \ldots, \alpha_n(y_n)$ in the $y = y_1, \ldots, y_n$ variables, it is defined as follows:*
$F = cproject(T(x,y), y) = \{(x,y')|y' = closest\ vertex\ to\ \alpha\ s.t.\ T(x,y') = 1\}$
$= \{(x,y')|y' = argmin_{(y|T(x,y)=1)} \perp(\alpha, y)\}$,
where $\perp$ is a distance metric.

The interested reader may refer to [59] for a more detailed description of the cproject operator. For example, the operation of cproject is shown in Figure 4.3.

Thus the spanning tree is computed by the following algorithm, where $P(x,y)$ denotes the rs-spanning tree, $R(x)$ the set of reachable states, and $\tau_0(x,y)$ is the initial rs-spanning tree.

The following algorithm takes as input a starting rs-spanning tree (which can be the tree of edges from initial states), and returns a rs-spanning tree for the reachable states in the FSM.

```
Algorithm 4.2 ($T(x,y), \tau_0(x,y)$)
    $f(Q(x,y)) = cproject(R(x) \cdot T(x,y) \cdot \overline{R(y)}, y) + Q(x,y))$
    where $R(x) = \exists_y (Q(y,x) + Q(x,y))$
    $P(x,y) = LFP(f(Q), \tau_0(x,y))$
  return $P(x,y)$
```

As an example of this procedure consider the example in Figure 4.4

Lemma 4.1 *Algorithm 4.2 is correct, i.e. it returns a reachable states rs-spanning tree of the state transition graph if* $\tau_0(x,y) = cproject(I(x) \cdot T(x,y) \cdot \overline{I(y)}, y)$, *i.e. the initial rs-spanning tree, and* $I(x)$ *is the set of initial states.*

Proof. Proof by induction.

Let $R^i_{actual}(x)$ denote all states reachable by the ith iteration.

Assume at step i, Q(x,y) is a tree and spans all states reachable by ith iteration,

i.e. $R^i_{actual}(x) = R^i(x) = \exists_y(Q(x,y) + Q(y,x))$

Consider step $i+1$, where $R^{i+1}(x) = \exists_y f(Q(x,y)) + f(Q(y,x))$

$f(Q(x,y)) = cproject(R^i(x) \cdot T(x,y) \cdot \overline{R^i(y)}, y) + Q(x,y))$.

$cproject(R^i(x) \cdot T(x,y) \cdot \overline{R^i(y)}, y) \not\models \phi$ (else we have converged).

1. $f(Q(x,y))$ is a tree.

   Consider an edge $(a,s) \in cproject(R^i(x) \cdot T(x,y) \cdot \overline{R^i(y)}, y)$.

   $\Rightarrow cproject(R^i(a) \cdot T(a,s) \cdot \overline{R^i(s)}, y) = 1$.

   By definition *cproject* selects just one edge from $R^i(x) \cdot T(x,y) \cdot \overline{R^i(y)}$.

   $\Rightarrow R^i(a) \cdot T(a,s) \cdot \overline{R^i(s)} = 1$.

   $\Rightarrow R^i(a) = 1$, $T(a,s) = 1$ and $R^i(s) = 0$.

   $R^i(s) = 0$, hence s is not already part of $Q(x,y)$.

   $R^i(a) = 1$, hence a is part of $Q(x,y)$.

   cproject adds only one edge to s.

   $\Rightarrow f(Q(x,y))$ is a tree.

2. $f(Q(x,y))$ spans all states reachable by $i+1$th iteration.

   Consider $s \in R^{i+1}_{actual}(x)$.

   - If $s \in R^i_{actual}(x)$ we are done.

     Otherwise $s \not\in R^i_{actual}(x)$, i.e. $R^i_{actual}(s) = 0$

   - If $\exists_a : f(Q(a,s)) = 1 \Rightarrow s \in R^{i+1}(x)$, then we are done

     Otherwise $\forall_a f(Q(a,s)) = 0$.

     $\Rightarrow \forall_a cproject(R^i(a) \cdot T(a,s) \cdot \overline{R^i(s)}, y) = 0$.

     $\Rightarrow \forall_a R^i(a) \cdot T(a,s) \cdot \overline{R^i(s)} = 0$ (by definition of cproject).

- But $s \in R^{i+1}_{actual}(x) \Rightarrow \exists_a T(a,s) \cdot R^i_{actual}(a) = 1$.
  And we know $R^i_{actual}(s) = 0$.
- $R^i(x) = R^i_{actual}(x)$
  $\Rightarrow \exists_a T(a,s) \cdot R^i(a) = 1$ and $R^i(s) = 0$
  $\Rightarrow T(a,s) \cdot R^i(a) \neq 0$ and $R^i(s) = 0$
  $\Rightarrow R^i(a) \cdot T(a,s) \cdot \overline{R^i(s)} \neq 0$.
  A contradiction, hence $f(Q(x,y))$ spans all states reachable by $i+1$th iteration.

Hence, if $Q(x,y)$ is a tree at the ith iteration, then it is at the $i+1$ iteration, and τ_0 is a tree. Hence by induction Algorithm 4.2 returns a tree. □

A stronger statement about this algorithm for the rs-spanning tree, can be stated as follows:

Theorem 4.2 *The Algorithm 4.2 returns a correct rs-spanning tree, if τ_0 is any subset of the rs-spanning tree that includes all the initial states .*

Proof. From Lemma 4.1, Theorem 2.2 and the fact that τ_0 includes all the initial states .
□

4.4.2.2 Addition of Edges

If the only changes to the system consist of the addition of edges to the transition relation, then the new rs-spanning tree is a superset of the current rs-spanning tree. Note that adding edges to the transition relation can never make a reachable state unreachable and hence can never remove a state (representative edges) from the rs-spanning tree. Hence the new rs-spanning tree must be a superset of the current rs-spanning tree. The following lemma summarizes this:

Lemma 4.3 *If the only change to the system consists of the addition of edges to the transition relation, then $P(x,y) \subseteq P^{new}(x,y)$.*

Proof. Consider some edge $E(x,y) \in P(x,y)$.
$\Rightarrow$ There exists a path from the initial states to the edge $E(x,y)$.

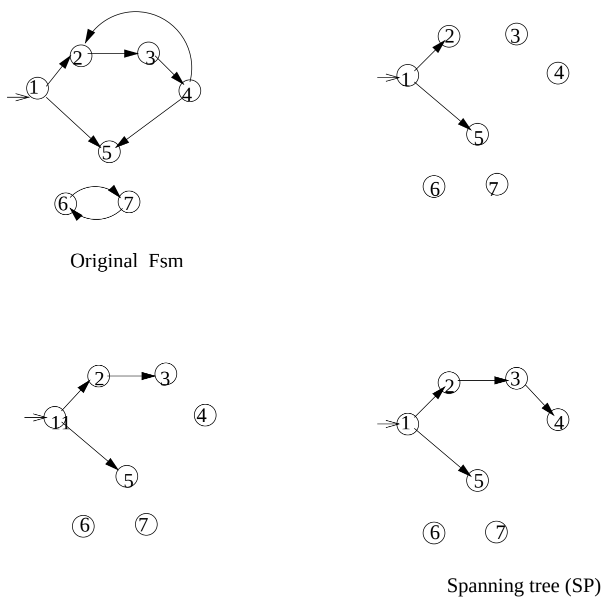


Figure 4.4: Computing the rs-spanning tree $P(x, y)$

The existence of this path is not affected by the addition of any other edges (all edges in the path always remain).
Hence, the edge $E(x, y) \in P^{new}(x, y) \Rightarrow P(x, y) \subseteq P^{new}(x, y)$. $\square$

4.4.2.3 Deletion of Edges

If edges that do not belong to the rs-spanning tree are deleted from the transition relation, they do not affect the rs-spanning tree, and it remains the same. However, if these edges do belong to the rs-spanning tree, then potentially every (eventual) successor edge of each deleted edge may be removed from the rs-spanning tree. After the removal of these edges, we will be left with a proper subset of the rs-spanning tree. This is the starting point for the iterative reachability Algorithm 4.2.

Let $\triangle T^{sub}(x, y)$ denote the edges that are deleted from the transition relation, and $P^{+}(x, y)$ denote the rs-spanning tree minus $\triangle T^{sub}(x, y)$ and all its successors. $P^{+}(x, y)$ may be computed as the greatest fixed point of $P^{+}(x, y) \cdot (\exists_y P^{+}(y, x) + I(x))$, given $P(x, y) - \triangle T^{sub}(x, y)$; i.e. by iteratively deleting all states that have no predecessors. This notion is formalized in the following lemma:

Lemma 4.4 *If the only change to the system consists of the subtraction of edges from the transition relation,*
$f'(Q(x,y)) = (Q(x,y) \cdot (\exists_y Q(y,x) + I(x)))$*, and*
$P^{+}(x,y) = GFP(f'(Q), (P(x,y) - \triangle T^{sub}(x,y)))$
then $P^{+}(x,y)$ *is a tree that contains all the initial states.*

Proof.

1. $P^{+}(x,y)$ contains the initial states.
   The fixed point function $f(Q(x,y)) = (Q(x,y) \cdot (\exists_y Q(y,x) + I(x)))$ always retains the initial states $I(x)$, i.e. $\exists_y Q(I(x),y) = 1 \Rightarrow \exists_y f(Q(I(x),y)) = 1$ (by construction).
   All the initial states are part of $P(x,y)$ by construction (Theorem 4.2).
   Hence $P^{+}(x,y)$ contains the initial states.

2. $P^{+}(x,y)$ is a tree.
   $P(x,y)$ is initially a tree.
   $f(Q)$ removes states that have no predecessor edge. (by construction)
   if some state has no predecessor then $f(f(Q(x,y))) \subset f(Q(x,y))$.
   $\Rightarrow GFP(f(Q), (P(x,y) - \triangle T^{sub}(x,y)))$ cannot terminate if some state has no predecessor edge.
   Hence $P^{+}(x,y)$ is a tree, since it is obtained by removing edges from a tree and its is contiguous (states without predecessors are removed).

Hence proved. $\square$ $P^{+}(x,y)$ is a tree that contains the initial states, and from Theorem 4.2 it can be supplied to Algorithm 4.2.

4.4.2.4 Incremental Rs-Spanning Tree P Algorithm

A general change consists of both the addition, and subtraction of edges. Let T^{new} denote the new transition relation that is obtained by adding, and subtracting the requisite edges from the transition relation, and P^{new} be the corresponding rs-spanning tree. Lemmas 4.3 and 4.4 can be combined to give the following lemma.

Lemma 4.5 *For any general change to the system,*
$f(Q(x,y)) = (Q(x,y) \cdot (\exists_y Q(y,x) + I(x)))$, *and*
$P^+(x,y) = GFP(f(Q), P(x,y) - \triangle T^{sub}(x,y)) \subseteq P^{new}(x,y)$.

Proof. Any change to the reachable set can be characterized by the addition and subtraction of edges from the transition structure. The proof follows from Lemma 4.4 and Lemma 4.3. □

Note that this lemma, in conjunction with Theorem 4.2 can be used to compute a new rs-spanning tree via the following algorithm,

Algorithm 4.3

```
    $T^{new}(x,y) = T(x,y) + \triangle T^{add}(x,y) - \triangle T^{sub}(x,y)$
    $f(Q(x,y)) = (Q(x,y) \cdot (\exists_y Q(y,x) + I(x)))$
    $P^+(x,y) = GFP(f(Q), (P(x,y) - \triangle T^{sub}(x,y))$
  return Algorithm 4.2($T^{new}(x,y), P^+(x,y), I(x)$)
```

Here $P(x,y)$ denotes the rs-spanning tree before the change, T^{new} is the new transition relation, and $I(x)$ the initial set of states. In order to demonstrate this algorithm consider Figure 4.5.

4.4.3 Spanning Graph Algorithm

Since the computation of the spanning tree is somewhat complicated, a variant of this procedure, which computes a spanning graph rather than a tree may also be used. The basic procedure is the same; however this procedure does not use the cproject selector, as it does not require a tree. The computation of this graph $P'(x,y) \subseteq T(x,y)$ is given by the following algorithm, where $\tau_0(x,y) = I(x) \cdot T(x,y) \cdot \overline{I(y)}$:

Algorithm 4.4 *($T(x,y), \tau_0(x,y)$)*

```
    $f(Q(x,y)) = (R(x) \cdot T(x,y) \cdot \overline{R(y)}) + Q(x,y)$
    where $R(x) = \exists_y (Q(y,x) + Q(x,y))$
```

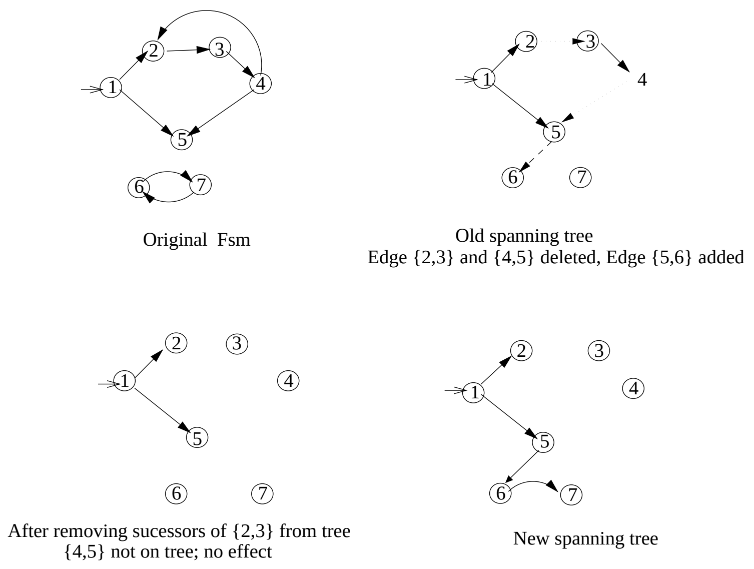

Original Fsm
Old spanning tree
Edge {2,3} and {4,5} deleted, Edge {5,6} added
After removing sucessors of {2,3} from tree
{4,5} not on tree; no effect
New spanning tree

Figure 4.5: Updating the rs-spanning tree $P(x, y)$

```
    $P'(x,y) = LFP(f(Q), \tau_0(x,y))$
  **return** $P'(x,y)$
```

The proof of correctness is similar to the proof for the tree algorithm in the previous section. We denote this spanning graph as *rs-spanning graph*.

4.4.3.1 Addition of Edges

All conclusions that were made for a rs-spanning tree in the previous section, also hold for the rs-spanning graph. Hence Lemma 4.3 also holds for the graph $P'(x, y)$.

4.4.3.2 Deletion of Edges

If edges that do not belong to the rs-spanning graph are deleted from the transition relation, they do not affect the rs-spanning graph, and it remains the same. However, if these edges do belong to the rs-spanning graph, then potentially every (eventual) successor edge of each deleted edge may be removed from the rs-spanning graph. Note that any successor that has another predecessor does not need to be removed. After the removal of these edges, we may be left with a proper subset of the rs-spanning graph. This is the starting point for the iterative reachability Algorithm 4.4. This set can also be computed by using Lemma 4.6, i.e. retaining states that have predecessors.

Lemma 4.6 *If the only change to the system consists of the subtraction of edges from the transition relation,*
$f(Q(x,y)) = (Q(x,y) \cdot (\exists_y Q(y,x) + I(x)))$, *and*
$P^{+}(x,y) = GFP(f(Q),(P(x,y) - \triangle T^{sub}(x,y))) \subseteq P^{new}(x,y)$.

Proof. Similar to Lemma 4.4. □

4.4.3.3 Incremental Rs-Spanning Graph P' Algorithm

A general change consists of both the addition, and subtraction of edges. Let T^{new} denote the new transition relation that is obtained by adding, and subtracting the requisite edges from the transition relation. P'^{new} the corresponding rs-spanning graph is computed via the following algorithm,

Algorithm 4.5
```
    $T^{new}(x,y) = T(x,y) + \triangle T^{add}(x,y) - \triangle T^{sub}(x,y)$
    $f(Q(x,y)) = (Q(x,y) \cdot (\exists_y Q(y,x) + I(x)))$
    $P'^{+}(x,y) = GFP(f(Q),(P'(x,y) - \triangle T^{sub}(x,y)))$
  return Algorithm 4.4($T^{new}(x,y), P'^{+}(x,y)$)
```

Here $P'(x,y)$ denotes the rs-spanning graph before the change, T^{new} is the new transition relation, and $I(x)$ the initial set of states.

4.4.4 Transitive Closure Algorithm

The transitive closure represents all paths in the state transition graph. A state is reachable if there exists a path from the initial states to it. Thus, the transitive closure restricted to the reached set represents the set of reachable states. However, as demonstrated in the previous sections, we cannot retrieve information from cyclic graphs, so we choose to store an acyclic variant of the rs-transitive closure that consists of all reachable edges in the transitive closure, except for a few that are deleted so as to ensure that it is acyclic. We denote this closure as *rs-transitive closure*. We use the existence of an edge to a vertex as an indication of a path to the vertex from an initial state; cycles pose an inherent problem for our incremental computation because an incoming edge to a vertex may just represent a path from the vertex itself, and not another distinct path from an initial state. This issue will become more clear after we present the algorithm.

The rs-transitive closure will be referred to as C, and is used to represent the reachable states. Since C is more dense than the transition relation, it may have a more compact BDD representation. The closure C can be computed by iteratively taking one BFS (reachability) step of the state transition graph, and computing the closure as the sum of the closure computed at the previous iteration and the new additions to the closure from the edges traversed in the current BFS step. There is an additional caveat that no edge that completes a cycle with pre-existing edges, is added to C at any iteration. We heuristically exclude as many edges from the rs-transitive closure as necessary to ensure this acyclicity.

C is computed using a fixed point that computes C_{i+1} as the union of C_i and a set of edges, which are the immediate successors of the edges in C_i (in the transition relation) and do not create a cycle in C_{i+1}.

Let $T(x, y)$ denote the transition relation, where x & y denote the present and next state variables respectively. Let $I(x)$ denote the initial states, and $C_0(x, y)$ denote the initial C supplied. For the current discussion this may be assumed to consist of edges out of the initial states; $C_0(x, y) = T(x, y) \cdot I(x) \cdot (x \not\equiv y)$, however in later sections we will show how this may take other interpretations. The following algorithm gives a means of computing C:

```
**Algorithm 4.6** *($T(x,y), C_0(x,y)$)*
    $f(Q(x,y)) = Q(x,y) + \exists_z Q(x,z) \cdot T(z,y) \cdot \overline{Q(y,x)} + R^i(x) \cdot T(x,y) \cdot \overline{Q(y,x)}$
    *where* $R^i(x) = \exists_y Q(y,x) + Q(x,y)$
    $C(x,y) = LFP(f(Q), C_0(x,y))$
  **return** $C(x,y)$
```

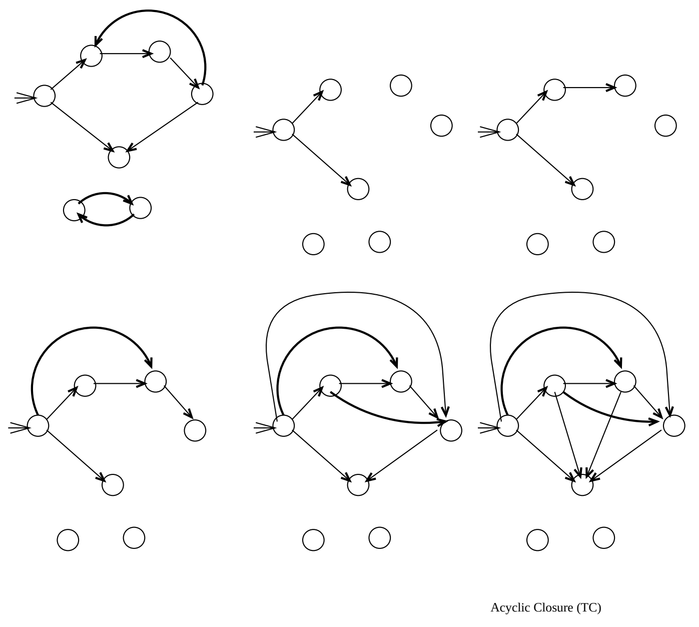


Figure 4.6: Computing C

The functioning of this algorithm is shown in Figure 4.6.

There are many acyclic rs-transitive closures for a given graph. This algorithm picks one according to the starting set C_0. Any C_0 that does not violate a some condition (does not contain some edge that will not be present in the final acyclic rs-transitive closure) is a valid starting point. Note that if needed the reachable set of states can be extracted from C using $R(x) = \exists_y C(y,x) + C(y,x)$, and hence C is sufficient to represent the reachable states.

Lemma 4.7 *Algorithm 4.6 is correct, and returns a valid C, when supplied with the transition relation $T(x,y)$, Initial states $I(x)$, and where $C_0(x,y) = T(x,y) \cdot I(x) \cdot (x \not\equiv y) \cdot \overline{T(y,x)}$, i.e. C is an acyclic closure that spans the reached state set.*

Proof. Proof by Induction.
Assume at some step, $Q(x,y)$ is an acyclic rs-transitive closure of the graph restricted to states reachable in i or less steps.
$R^i(x) = R^i_{actual}(x)$
Consider $f(Q(x,y))$.
If there exists a path $z_0, z_1 \ldots z_{n-1}, z_n$ from $z_0 = a$ to $z_n = s$, $R^{i+1}_{actual}(z_j) = 1$, $Q(s,a) = 0$.

1. Either $\forall_{z_j} R^i(z_j) = R^i_{actual}(z_j) = 1$.
   $\Rightarrow Q(a,s) = 1 \Rightarrow f(Q(a,s)) = 1$.
2. If $\exists_{z_j}$ *such that* $R^i(z_j) = R^i_{actual}(z_j) = 0$, and $R^{i+1}_{actual}(z_j) = 1$.
   Without loss of generality assume $z_j = s$, i.e. exactly $R^i(s) = 0$. If for some intermediate $R^i(z_j) = 0$, then consider a path from a to $z_j = s$ instead.
   $R^{i+1}_{actual}(s) = 1$, since s is reachable by the $i+1$ th step.
   We need to prove that $R^{i+1}(s) = 1$.
   - Either there exists an edge between a and s
     $\Rightarrow T(a,s) = 1,\ R^i(a) = 1$
     $\Rightarrow R^i(a) \cdot T(a,s) \cdot \overline{Q(s,a)} = 1$
     $\Rightarrow f(Q(a,s)) = 1$.
     $\Rightarrow R^{i+1}(s) = 1$.
   - Or, the path can be broken into a path from $z_0 = a$ to z_{n-1} followed by an edge from z_{n-1} to $z_n = s$.
     $\Rightarrow Q(a, z_{n-1}) = 1, T(z_{n-1}, s) = 1$ and $Q(s,a) = 0$.
     (To avoid cycles there must be no path from s to a)
     $\Rightarrow f(Q(a,s)) = 1$.
     $\Rightarrow R^{i+1}(s) = 1$.

Finally, $C_0(a,s) = 1$ if $T(a,s) = 1$, i.e. there is a path from a to s of $length = 1$, and $R^0(a) = I(a) = 1$ □ A stronger result holds for Algorithm4.6.

Theorem 4.8 *Algorithm 4.6 returns a correct C, when given as input $C_0(x,y)$ any subset of the final C that contains the initial states.*

Proof. From Lemma 4.7, Theorem 2.2 and the fact that Algorithm 4.6 is a least fixed point computation. □

4.4.4.1 Addition of Edges

If the only type of change to the system consists of edge addition, then all edges already present in C must remain there, because addition of edges can only add more paths; it can never delete paths. This is formalized by the following lemma:

Lemma 4.9 *If edge addition is the only class of change applied to the system, then* $C(x,y) \subseteq C^{new}(x,y)$.

Proof. The existance of a path between a and s is never affected by the addition of edges, hence all edges already present in $C(x,y)$ must remain.
$C(x,y) \subseteq C^{new}(x,y)$. $\square$

Thus, by Theorem 4.8, $C^{new}(x,y) =$ Algorithm 4.6$(T(x,y), C(x,y))$.

4.4.4.2 Subtraction of Edges

If edge subtraction from the transition relation is the only class of change applied to the FSM, then all successor edges of deleted edges that have no other predecessor edge may be removed in $C^{new}(x,y)$. In addition, edges in C that represent paths no longer in the FSM, may also be deleted. This set can also be represented as the fixed point of $Q(x,y) \cdot (\exists_z Q(x,z) T^{new}(z,y) + T^{new}(x,y)) \cdot (\exists_z Q(z,x) \cdot I(z))$, where $I(x)$ is the set of initial states, $Q(x,y)$ is initially supplied the original $C(x,y)$ computed before the change, and $C^{new}(x,y)$ is the new transition relation.

The first term in this expression $(\exists_z Q(x,z) T^{new}(z,y) + T^{new}(x,y))$ recursively removes edges in C that correspond to paths that no longer exist in the FSM, and the second term $(\exists_z Q(z,x) \cdot I(z))$ removes edges that have no path from $I(x)$.

Lemma 4.10 *If* $T^{new}(x,y) = T(x,y) \cap \overline{\triangle T^{sub}(x,y)}$ *is the new transition relation, then* $f(Q) = Q(x,y) \cdot (\exists_z Q(x,z) \cdot T^{new}(z,y) + T^{new}(x,y)) \cdot (\exists_z Q(z,x) \cdot I(z))$, *and* $C^{+}(x,y) = LFP(f(Q), C(x,y)) \subseteq C^{new}(x,y)$, *i.e.* $C^{+}(x,y)$ *is a subset of rs-transitive closure that contains all the initial states.*

Proof. We need to prove that $C^{+}(x,y) \subseteq C^{new}(x,y)$.
Assume $\exists (a,s)$ *st* $C^{+}(a,s) = 1$.

1. There exists a path from a to s.
   If $T^{new}(a,s) = 1$ then $C^{new}(a,s) = 1$ (by Algorithm 4.6).
   Hence $C^{+}(a,s) = 1 \Rightarrow \exists_{z_n} C^{+}(a,z_n) \cdot T^{new}(z_n,a) = 1$. (by definition of $f(Q)$)

$\Rightarrow \exists_{z_n}(C^+(a, z_n) = 1$ and $T^{new}(a, z_n) = 1)$.

Applied recursively

$\exists z_1 \ldots z_n$ such that $T(a, z_1) = 1, \ldots T(z_{i-1}, z_i) = 1 \ldots T(z_n, s) = 1$.

$\Rightarrow$ there exists a path from a to s.

Hence, it is a closure.

2. a is reachable.

   $C^+(a, s) = 1 \Rightarrow \exists_z C^+(z, a)I(z) = 1$. (by definition of $f(Q)$)

   $\Rightarrow$ there exists a path from initial states $I(x)$ to a.

   a is reachable.

3. $C^+(x, y)$ is acyclic.

   $C^+(x, y) \subseteq C(x, y)$ and $C(x, y)$ is acyclic.

   $\Rightarrow C^+(x, y)$ is acyclic.

There exists a reachable path that creates no cycles, from a to s.

$\Rightarrow C^{new}(a, s) = 1$ Hence $C^+(x, y) \subseteq C^{new}(x, y)$. $\square$

Thus, by Theorem 4.8, Algorithm 4.6$(T(x, y), C^+(x, y)) = C^{new}(x, y)$.

4.4.4.3 Incremental C Algorithm

The lemmas in the previous sections and Theorem 4.8 can be combined to get the following algorithm for any general change:

```
Algorithm 4.7 ($T(x,y), C(x,y), I(x), \triangle T^{sub}(x,y), \triangle T^{add}(x,y)$)
    $T^{new}(x,y) = T(x,y) + \triangle T^{add}(x,y) - \triangle T^{sub}(x,y)$
    $f(Q) = Q(x,y) \cdot (\exists_z Q(x,z) T^{new}(z,y) + T^{new}(x,y)) \cdot (\exists_z Q(z,x) + I(x)$
    $C^+(x,y) = GFP(f(Q), C(x,y)) \subseteq C^{new}(x,y)$
  return Algorithm 4.6($T^{new}(x,y), C^+(x,y), I(x)$)
```

$C(x, y)$ denotes the rs-transitive closure before the change, T^{new} is the new transition relation, and $I(x)$ is the initial set of states.

Theorem 4.11 *Algorithm 4.7 is correct, i.e.*
Algorithm 4.7$(T(x,y), C(x,y), I(x), \triangle T^{sub}(x,y), \triangle T^{add}(x,y)) = C^{new}(x,y)$.

Proof. From Lemma 4.10 and Lemma 4.9. □

4.4.5 Extending Incremental FSM Traversal to Partial Products Heuristics

All the methods described in the previous section have the intrinsic flaw that they require building the monolithic transition relation associated with the product machine. However, this is not necessary for traversal; in fact building and manipulating the monolithic product transition relation is a more time-consuming and expensive method of FSM traversal. In practice, traversal may be done using the partial product heuristics, as described in [45], [60] [61], and [62]. Thus, traversal requires computing the fixed point of

$$\begin{aligned} f(Q) &= Q(x) + \delta Q(x) \\ \delta Q(x) &= (\exists_{x,i} T_1(x,y_1,i)\ldots T_n(x,y_n,i) \cdot Q(x))_{y \leftarrow x} \\ R(x) &= LFP(f(Q), I(x)) \end{aligned}$$

where $\exists_i (T_1(x,y_1,i) \cdot T_2(x,y_2,i) \ldots T_n(x,y_n,i) = T(x,y))$, the product transition relation. We find efficient methods for computing the result $f(Q)$ from the previous expression, rather than forming the product transition relation. This can be extended in order to compute the reached state relation. In this section, we will describe how the reached state relation (we will only be describing the rs-spanning graph representation) can be computed using partial products heuristics. We will rely on the the methods of [61] to efficiently compute an expression of the form $(\exists_{x,i}(T_1(x,y_1,i) \cdot T_2(x,y_2,i) \ldots T_n(x,y_n,i)) \cdot Q(x))$ Thus algorithms 4.4 and 4.5 can be re-written in the partial product context as:

Algorithm 4.8 *(*$T(x,y), \tau_0(x,y)$*)*

$f(Q(x,y)) = Q(x,y) + \delta Q(x,y)$
$\delta Q(x,y) = (\exists_i T_1(x,y_1,i)\ldots T_n(x,y_n,i) \cdot R(x)) \cdot \overline{R(y)}$
$R(x) = \exists_y (Q(x,y) + Q(y,x))$

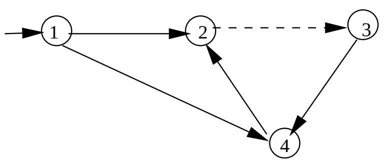


Figure 4.7: Example

```
    $P'(x, y) = LFP(f(Q), \tau_0(x, y))$
  **return** $P(x, y)$
```

Algorithm 4.9

```
    $T^{new}(x, y) = T(x, y) + \triangle T^{add}(x, y) - \triangle T^{sub}(x, y)$
    $f(Q(x, y)) = Q(x, y) \cdot (\exists_y Q(y, x) + I(x))$
    $P'^{+}(x, y) = GFP(f(Q), (P'(x, y) - \triangle T^{sub}(x, y)))$
  **return** *Algorithm 4.8*$(T^{new}(x, y), P'^{+}(x, y))$
```

For a deterministic transition system, the rs-spanning graph, which is a subset of the transition relation is also deterministic, and hence $Q(x, i, y) = Q_1(x, i, y_1) \cdot Q_2(x, i, y_2) \ldots Q_n(x, i, y_n)$. The expressions $R(x) = \exists_{x,i} Q_1(x, i, y_1) \cdot Q_2(x, i, y_2) \ldots Q_n(x, i, y_n)$, and $\delta\, Q(x, y) = ((\exists_i (T_1(x, y_1, i) \cdot T_2(x, y_2, i) \ldots T_n(x, y_n, i) \cdot R(x)))$ are both of the form required by the heuristic algorithms of [61]. A similar extension can be made to compute the quantification operation used for the deletion of edges.

4.5 Efficient Update

Notice that with all the algorithms presented in the previous section, it is possible to first remove, and then replace the same set during one iteration of the algorithm.

For the example described in Figure 4.7, if edge $(2, 3)$ is deleted, then the all three algorithms will first remove and then re-add $(3, 4)$ to the corresponding reached state relations. This is because none of the reached state relations described completely represent all path information, and hence complete update is not possible. All of them do not contain edge $(4, 2)$. In order to explore the possibility of a truly incremental algorithm, we conducted the following thought experiment. Store the complete rs-transitive closure with path count information annotated at each edge.

If an edge is added to the graph, you can update this closure graph by adding all new paths that are created as a result of this edge (similar to Algorithm 4.6). All old paths remain.

If an edge is deleted from this graph, you must reduce the path count of all paths (edges) through this edge. If any edge count drops to zero, it must be removed from the graph. If an edge is removed, it is never put back.

However, the effort required to update this reached state relation is related to the number of paths in the graph rather than the number of states, and hence this procedure is usually far more in-efficient than re-computing the reached state information.

4.6 Experiments and Results

We have implemented the algorithms described in the previous sections, in the HSIS [63] environment, and tested these on some ISCAS 89 and miscellaneous benchmarks. The following graphs and tables (Figure 4.3, Table 4.2, and Figure 4.1) summarize the results. The basic algorithm was run once, and then random changes consisting of addition and subtractions of sets of edges, were made. After these changes were made, both incremental and non-incremental algorithms were run on the new input. This process was repeated. The actual set of edges that are added, and subtracted is randomly chosen. The NR algorithm refers to Algorithm 4.1 reported in section 4.3, IRT refers to Algorithm 4.3, IRG refers to Algorithm 4.5, and $PIRG$ refers to Algorithm 4.8. All successive incremental changes were made directly to the system within the HSIS environment.

Figure 4.1 reports the ratio of the incremental to the non-incremental time for all methods, and Table 4.2 tabulates the incremental time to non incremental time for some representative examples (using the partial product method).

Only the partial product methods were able to handle larger examples, and examples tlc,

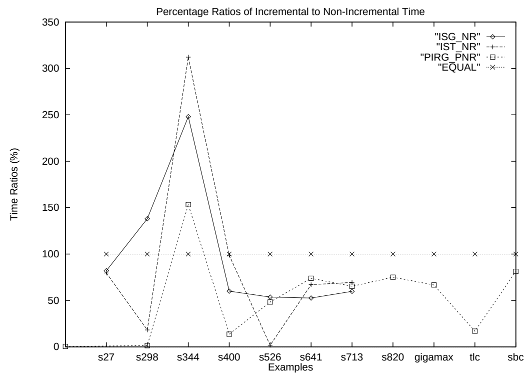


Table 4.1: Ratio Incremental FSM Traversal Vs. 0-1 Reachability

Total Time (sec)		
Example	Incremental PIRG	Non-Incremental PNR
s27	0.01	0.01
s298	0.03	2.26
s344	2.76	1.80
s400	1.72	12.57
s526	3.6	7.44
s641	6.07	9.06
s713	5.78	8.85
s820	0.12	0.16
gigamax	4.07	6.11
tlc	0.09	0.53
sbc	1109.78	1363.86

Table 4.2: Incremental Graph Algorithm (Partial Product)

gigamax, sbc etc. only report partial product times.

Figure 4.3 presents the average ratio of the depth (number of iterations to fixedpoint within the algorithm) taken by the incremental algorithm as compared to the non-incremental algorithm in a single run of both algorithms. Notice that since this ratio is always smaller than 1, the incremental algorithm always takes fewer iterations to reach a fixed point.

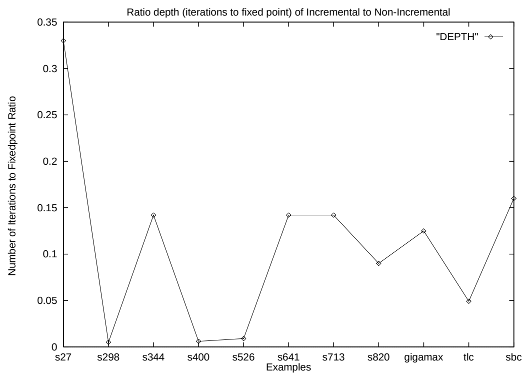


Table 4.3: Ratio Incremental Vs. Non-Incremental Depth

Chapter 5

Incremental Language Containment

5.1 Introduction

Design verification is the process of checking if what the designer specified is what he/she wants. One way to perform design verification on sequential logic circuits is to specify the design (also called the system), as well as the requirements of the design (also called the properties) as a *finite automaton* (or finite state machine), usually by the process of abstraction. Next, we verify that the *language* (the set of behaviors) of the property is a superset of the language (or behavior) of the system. The requirement that the language of the property contains the language of the system is called *language containment*. Language containment fails due to the presence of states that show behavior that is in the system but not in the property. This set of states is called the set of *Fair* states.

In general, the system itself need not be a single finite state machine. It is more commonly expressed as a set of interacting finite state machines that form a compound entity called the *product machine*. Figure 5.1 illustrates a system composed of three interacting finite state machines (M1, M2, M3), with transition relations (T_1, T_2, T_3). The transition relation of this system describes how the current state of the system and inputs relate to the next state and outputs; it is the Cartesian product of the individual transition relations of the component machines, namely $T = T_1 \times T_2 \times T_3$. The problem of language containment has to be solved in this environment of interacting finite state machines [64].

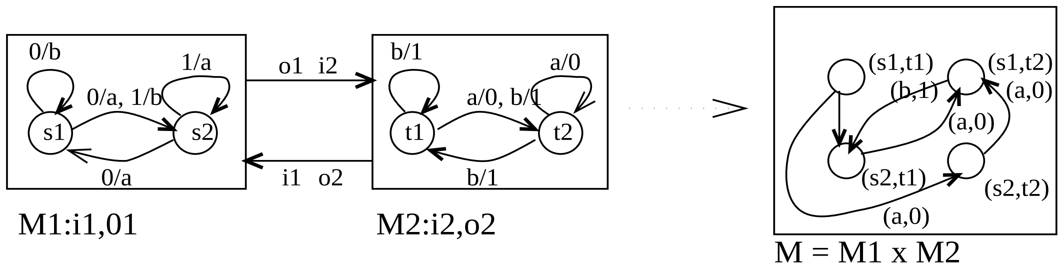


Figure 5.1: A system of interacting finite state machines

Current techniques [14] [15] perform language containment as a single pass. If the designer modifies the design after a solution has been obtained, then the entire language containment algorithm is repeated on the new design. In practice, the process of design is *iterative*; the designer modifies and re-verifies the design many times. If standard language containment algorithms are used in real-life design situations, they often result in redundant re-computation of information because the similarity between the old system and the new system is not utilized. We introduce the concept of *incremental verification*, which allows multiple changes to the system but runs the entire language containment algorithm only once, and propagates successive changes or increments from the latest solution.

The language containment algorithms of Touati et al [14] and Hojati et al [15] start with all reachable states, and successively reduce this until only the *fair* states remain. These algorithms are monotonic in nature, i.e., once a state is removed from the set of potential fair states, it is never added back. Hence, a similar algorithm that starts with any superset of the fair set, would return the fair set. Our algorithm uses information about the change in the system and the original set of fair states to derive a smaller superset of new fair states (smaller than the set of all reachable states). Then, it reduces this superset with an algorithm similar to [15]. Since this superset is much smaller than the set of all reachable states, the incremental algorithm converges faster.

The aim of this exercise is to get the new answer to the verification decision problem, "Is what I specified what I wanted?", using the old fair states (also referred to as $Fair^+$), and the incremental changes that the designer made to the input problem, while spending less time and effort in this computation than if the entire language containment algorithm was run on the new problem.

Our approach to incremental language containment will begin by identifying classes of changes to the system. For each type of change, we show how to modify the $Fair^+$ set to account for it (Section 5.4), and finally we merge these into a single incremental language containment algorithm (Section 5.5). A shorter description of this work has been published in [49].

It is important to keep in mind that all operations are to be carried out in the context of the Binary Decision Diagram (BDD) data structure [40] (defined in Chapter 2). Even though not explicitly stated, all sets and relations are represented as their BDD's [45].

5.2 Some Terminology for Language Containment

The following terms are defined on a Finite State Machine (defined Section 4.2) with transition relation $T(x, i, o, y)$, initial states $I(x)$, reachable states $R(x)$.

Definition 25 Projection*: Given a relation $R(x, y)$, the projection $\Lambda(R(x, y), x)$ denotes the projection of the relation on the x variables, i.e. $\Lambda(R(x, y), x) = \exists_y R(x, y)$.*

If $R(x, y)$ denotes a set of edges in the FSM (e.g. $R(x, y) = T(x, y)$), then $\Lambda(R(x, y), x)$ denotes the of set of predecessor states of the given edges.

Definition 26 Run*: A sequence of states, $r = r_o \ldots r_i \ldots, r \in Q^\omega$, is a run,or a* path *of T for a word $\sigma = (\sigma_0 \ldots \sigma_i \ldots)$, $\sigma \in \Sigma^\omega$, if $r_0 \in I$ and for $i \geq 0$, $T(r_i, \sigma_i, \gamma_i, r_{i+1}) = 1$. The set I refers to the set of initial states.*

The infinity set of a run r, denoted as $\inf(r)$, is the set of states that are visited infinitely many times in r. An accepting run or **fair path** r over T requires that $\inf(r)$ satisfies some acceptance condition C. The acceptance condition C distinguishes different ω-automata (Automata accepting infinite behavior, e.g. L-automata, Buchi, Streett and Rabin automata).

The **behavior** (set of **fair runs**) of the system is a subset of the runs of the system. This subset is specified using fairness constraints on the processes of the system. The **fairness conditions** express restrictions on the infinitary behavior of the finite state machine, and are used to model the system, the environment, and acceptable behaviors. Fairness conditions are modeled differently for different classes of automata. The **language** of an automaton M, represented as $L(M)$, is the set of all strings accepted by it.

Definition 27 Language Containment: *The requirement that the language of the property (or specification) is a superset of the language of the system is called language containment.*

In the language containment paradigm, verification of the system is equivalent to determining if there is a fair path starting at an initial state. This path corresponds to behavior that is generated by the system but rejected by the task or property automaton and it is a witness to the failure of the property. The set of states which are involved in fair behavior are called *Fair states.*

We will be considering the following types of infinitary automata.

Definition 28 Buchi Automata: *Buchi automata are characterized by acceptance conditions that consist of U a subset of the state space of the machine and run r is accepting if and only if some of the states in U are traversed infinitely often, i.e. $(\mathit{inf}(r) \cap U \neq \emptyset)$.*

Definition 29 L-Automata: *The L automaton[16] acceptance condition consists of a pair $\langle R, U \rangle$. $R \subseteq Q \times Q$, is termed the set of* recur *edges, and $U = U_1 \ldots U_n$ is the set of cycle sets (or Buchi sets). Run r is accepting if and only if $\exists_i, inf(r) \subseteq U_i$ or $inf_e(r) \cap R \neq \emptyset$, where $inf_e(r)$ denotes the set of infinitely occurring edges in r.*

Definition 30 L-Process : *An L-process is syntactically the same as an L-automata, with one exception; the acceptance conditions for L-automata are complementary to those of L-processes, i.e. run r is accepting if and only if $\forall_i inf(r) \not\subseteq U_i$ and $inf_e(r) \cap R = \emptyset$, where $inf_e(r)$ denotes the set of infinitely occurring edges in r.*

Definition 31 Streett Automata *[65]: An FSM that accepts infinite behavior, which satisfies the Streett acceptance conditions is called a Streett automaton. Streett acceptance conditions consist of a finite set of ordered pairs* $\mathcal{C} = \{(U_1, V_1), (U_2, V_2), \ldots, (U_n, V_n)\}$ *where* U_i *and* V_i *are subsets of the state space of the machine and run* r *is accepting if and only if* $\forall_i((\mathit{inf}(r) \cap U_i \neq \emptyset) + (\mathit{inf}(r) \subseteq V_i))$*, where* $0 \leq i \leq n$*. This can also be written as* $F^\infty(U_i) + G^\infty(V_i)$*.*

Edge Streett have additional fairness constraints in the form of positive fair edges E_i which must be traversed infinitely often and negative fair edges N_i which must not be traversed infinitely often in any accepting run r.

Definition 32 Rabin Automata *[66]: The fairness conditions for a Rabin Automaton are the complements of the fairness conditions for a Streett automaton, i.e.,* $\forall_i((\mathit{inf}(r) \cap U_i = \emptyset) \cap (\mathit{inf}(r) \not\subseteq V_i))$*.*

Edge Rabin may also have positive and negative fair edges constraints.

Definition 33 Fair$^+$*: The set of states which can reach a* 'fair cycle' *(including those on it), i.e. a cycle which satisfies the fairness constraints, constitute* $Fair^+$*. The presence of a non-empty set* $Fair^+$ *indicates that the automaton has non-empty behavior.*

The least fixed points (LFP) and greatest fixed points (GFP) are defined as in Chapter 2. We are concerned with functions that map sets of states in the FSM to other sets of states. The following operators defined using the fixed point operators of Chapter 2 will be used in this chapter.

Definition 34 Forward Reachable Operator*: Given* $T(x, y)$*, the transition relation and* $A(x)$*, a set of vertices, the forward reachable operator returns the set of vertices which can be reached by* A*.*

The forward reachable operator FR is computed using the following algorithm:

$f(Q(x)) = \exists_x T(x, y) \cdot Q(x)|_{y \leftarrow x} + Q(x)$

```
**FR**$(T(x,y), A)$
  **return** $LFP(f(Q), A(x))$
```

Definition 35 Backward Reachable Operator *: Given $T(x,y)$, the transition relation and $A(x)$, a set of vertices, the backward reachable operator returns the set of vertices that can reach A.*

It can be computed as follows:

```
$f(Q(x)) = \exists_y T(x,y) \cdot Q(y) + Q(x)$
**BR**$(T(x,y), A)$
  **return** $LFP(f(Q), A(x))$
```

Definition 36 Reach Reachable States Operator*: Given $T(x,y)$, the Transition Relation and $A(x)$, a set of states, the Reach Reachable States operator returns the set of states which can reach some state in A or be reached by some state in A. The RRS operator returns the set of states, which are on paths through A.*

The *Reach Reachable States operator* or $RRS(T(x,y), A)$ is computed as follows:

```
**RRS**$(T(x,y), A)$
  **return** $(BR(T(x,y), A) + FR(T(x,y), A))$
```

5.3 Language Containment

Vardi and Wolper [67] observe that the problem of verifying whether a machine (M) satisfies a given property (P) reduces to the problem of checking whether the language of the machine automaton is contained in the language of the property automaton. The language containment check in turn reduces to a *language emptiness* check for the product of the system automaton and the complement of the property automaton. Checking whether $L(M) \subseteq L(P)$ is the same as checking whether the language of $D = M \times \overline{P}$ is empty, i.e., whether $L(M \times \overline{P}) = \phi$.

When P is expressed as an L-automaton, the problem of complementing P is solved by expressing it as an L-process [14]. The acceptance conditions for L-processes and L-automata are complementary and representing $\overline{P}$ by a L-process is easily done (if P is deterministic) by just keeping the same transition structure and complementing the acceptance conditions (the complementation is implicit by the choice of representation). Similarly when P is expressed as a Rabin automaton the problem of complementation is solved by expressing $\overline{P}$ as a Streett automaton, since the acceptance conditions for Rabin and Streett automata complementary. Our experiments use a Streett and Rabin environment, and hence all successive discussions in this report are centered around Streett and Rabin automata, but are also applicable to other classes of automata.

A *language emptiness* check remains to be done, and it is performed by checking the product automata $D = M \times \overline{P}$ for acceptable infinite behavior[14] (or fair paths), which indicate that the language for the system-property product machine is not empty. A cycle is associated with any infinitary behavior in a finite graph, and in order for this infinite behavior to be acceptable, this cycle must also satisfy the fairness constraints. Thus, a machine has a non-empty language if there exists a path from an initial state to cycle that satisfies the fairness constraints. The set of states that lie on such cycles form a set of *fair states*, which cause the *fair or non-empty* behavior. In general, we compute a superset of this set called $Fair^+$, which consists of all states on a path to a *fair cycle.*

Touati et al [14] have presented an algorithm for the computation of the $Fair$ states under Buchi and L-process acceptance conditions. This has been extended by Hojati et al [15] to an algorithm for computation of $Fair^+$ within a Streett environment. This algorithm relies on the following two operators.

Definition 37 Forward Stable Set Operator*[15]: Given a transition relation $T(x, y)$ and a set of vertices $A(x)$, the* forward stable set operator *or $FSS(T(x, y), A)$ returns a set of states in A, which are on a cycle or can reach a cycle in A. Alternately, the FSS operator removes from A all those states which have no successors states (next states) in the transition structure.*

The following algorithm is used to compute the Forward Stable Set operator FSS :

```
$f(Q(x)) = Q(x) \cdot (\exists_y T(x,y).A(x) \cdot Q(y))$
**FSS**$(T(x,y), A)$
  **return** $(GFP(f(Q), A(x)))$
```

Definition 38 Forward Fair Path Operator*[15]: Given a transition relation $T(x,y)$, a set of states $A(x)$, and a set of fairness constraints $C(x)$, the* forward fair path operator *or $FFP(T(x,y), C, A)$, returns a subset of states in $A(x)$ which are on a fair path. For this analysis, $C(x) \prod_i C_i$ are Streett fairness constraints in the form $C_i = F^\infty(U_i) + G^\infty(V_i)$. Hence, FFP returns those states a in A such that for each C_i, either $a \in V_i$ or there is a path in A from a to some state in U_i.*

Note that this operator returns just a path and not necessarily an infinite path. The FFP operator can be computed by using the following algorithm:

```
$f(Q(x)) = (\exists_y T(x,y).A(x) \cdot Q(y) + Q(x))$
**FFP**$(T(x,y), C, A)$
  **return**$(\prod_{i, c_i \in C(x)} (LFP(f(Q), U_i(x) \cdot A(x))) + V_i(x) \cdot A(x))$
```

To extend the FFP to edge Streett automata, an additional term must be added to the above expression that accounts for states such that for each E_j, there is a path from the state to E_j.

The algorithm computes $Fair^+$ by starting with the set of reachable states, and alternately applying the FSS and FFP operators. These operators successively restrict the original set of reachable states to those on a path from an initial state to a cycle (FSS) and those which are on a fair path (FFP). Thus, the set $Fair^+$ is obtained by successively shrinking the set of reachable states until only those states that are on a path from some initial state to a fair cycle remain. The algorithm for verification in the Streett-Rabin environment becomes:

```
**Algorithm 5.1** *Non_Incremental_Language_Containment*
  $Fair^+(x)$ = *Compute_Fair*$^+$
  *if* $Fair^+(x)$ *is empty* **return***(PASS)*
  *else* **return***(FAIL)*
```

The set $Fair^+$ is computed using the following algorithm:

```
Algorithm 5.2 Compute_Fair+
  Restrict the Transition Relation $T(x,y)$ to reachable states

  Set $R(x)$ = Reachable states
     $f(Q(x)) = FSS(T(x,y), FFP(T(x,y), C(x), Q(x)))$
     $Fair^+(x) = GFP(f(Q), R(x))$
  return $Fair^+(x)$
```

The proof of correctness of this algorithm can be found in [15].

This algorithm has a complexity of $O(N^2)$, where N is the number of reachable states in the state space. At each iteration of the fixed point computation, at least one state in the set of reachable states, but not in $Fair^+$ is deleted from the reachable set, and this step takes $O(N)$ time, which results in an overall complexity of $O(N^2)$. This computation of complexity assumes that each step takes $O(1)$ time, and all successive arguments on complexity in this chapter, also make this assumption. Even if this assumption did not hold, the complexities are valid for comparing the incremental method to the non-incremental method.

Though not explicitly stated in the above algorithm, the set of reachable states can also be used to minimize the transition relation BDD. This simplification results in a considerable speedup and will be used throughout this chapter without an explicit mention.

5.4 Incremental Language Containment

5.4.1 Overview

The computation of $Fair^+$ involves successive applications of the FSS and FFP operators, which involve the successive reduction of the set of states involved. It is important to note that the algorithm begins with a superset of the states in $Fair^+$ (namely all reachable states), and eliminates states. Once a state has been removed from this set, it is never added back, and hence the algorithm is monotonic. Our incremental algorithm is based

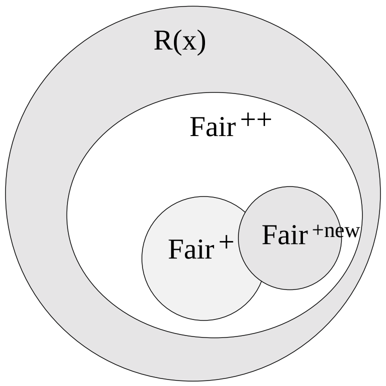


Figure 5.2: Key Idea: incremental language containment

on the intuition of Theorem 2.4 that if any superset of $Fair^+$ is given to Algorithm 5.2, it still returns the set $Fair^+$. The trick lies in using the previously computed $Fair^+$ and the changes to the system to obtain a superset of the new $Fair^+$, which is not necessarily as large as the set of all reachable states, and in most cases is significantly smaller (Figure 5.2). Given a smaller set, the algorithm converges faster and hence the incremental algorithm is typically faster.

5.4.2 Characterizing Incremental Changes

Recall that $Fair^+$ is a set of states that characterize the fair or unwanted behavior in the system. We want to use information about the changes to the system to incrementally modify $Fair^+$. The potential for speedup in this method is that $Fair^+$ need not be recomputed from the beginning; intermediate results can be used to avoid unnecessary computations.

We have categorized six different incremental changes to an instance of the language containment problem. Briefly, changes to the system may consist of 1) addition or subtraction of edges to the transition relation, 2) addition or subtraction of fairness constraints. Addition and subtraction of states can be characterized in terms of edges. Clearly, removing a state from the state space is equivalent (behaviorally) to removing all edges to the state, thus making it unreachable. Similarly, if a state is added to the state space, it is similar to making one of the unreachable states in the state space reachable by adding edges.

Suppose the designer modifies the original transition relation T to a new transition relation

T^{new}. Using T^{new} and T, we create T^{sub} and T^{add}. T^{sub} consists of the original transition relation T minus all transitions, which were removed in T^{new}. These subtracted transitions are referred to as $\triangle T^{sub}$. T^{add} consists of T^{sub} plus all the transitions added in T^{new}. The added set of transitions is referred to as $\triangle T^{add}$. Note that $T^{add} = T^{new}$, but for the purposes of incremental modification we can deal with T^{add} as a single modification to T^{sub} by only adding edges. The exact computation of T^{add} and T^{sub} under different methods for changing input, is described in Section 4.4.1.

Note that fairness constraints never affect the transition structure; they only affect the FFP operator (Section 5.2). The new fairness constraints, with constraints added and subtracted from the original set, are used to compute a new FFP operator. Let C^{add} refer to the original set C plus the new constraints added to the system $\triangle C^{add}$. Also, let C^{sub} denote the final set of constraints, which can also be interpreted as C^{add}, minus the set of subtracted constraints $\triangle C^{sub}$. For brevity, we denote $\overline{\triangle C^{sub}}(x) = \bigcup_{C_i \in \triangle C^{sub}} (\overline{U_i(x) + V_i(x)})$.

In order to prove the correctness of the results, we will make use of the following properties of the FFP and FSS operators:

Theorem 5.1 *The forward stable set operator satisfies the following three properties:*

1. *If* $T^{sub} \subseteq T$,
   $FSS(T^{sub}(x,y), A(x)) \subseteq FSS(T(x,y), A(x))$.
2. *If* $T^{add} \supseteq T$,
   $FSS(T^{add}(x,y), A(x)) \subseteq FSS(T(x,y), A(x)) + RRS(T^{add}(x,y), \Lambda(\triangle T^{add}(x,y), x))$
3. $FSS(T(x,y), A(x)) \subseteq A(x)$

Proof.

1. FSS is a GFP computation.
   $T^{sub} \subseteq T \Rightarrow f_{T^{sub}} \subseteq f_T$
   Where f denote the function in the GFP computation on page 81.
   Hence, by Theorem 2.5 and 2.3
   $FSS(T^{sub}(x,y), A(x)) \subseteq FSS(T(x,y), A(x))$.

2. Consider $s \in FSS(T^{add}(x,y), A(x))$.
   $T^{add} \supseteq T \Rightarrow f_{T^{add}} \supseteq f_T$
   Where f denote the function in the GFP computation on page 81.
   $\Rightarrow FSS(T^{add}(x,y), A(x)) \supseteq FSS(T(x,y), A(x))$.

   If $s \in FSS(T(x,y), A(x))$, then proved.
   Else if $s \notin FSS(T(x,y), A(x))$, then $\exists$ a newly created infinite path with s. (by definition of FSS).
   This path is not in the old FSM $(T(x,y))$, hence it must involve one of the new edges.
   $RRS(T^{add}(x,y), \Lambda(\triangle T^{add}(x,y), x))$ includes all paths through the new edges $(\triangle T^{add})$. (From RRS definition).
   $\Rightarrow s \in RRS(T(x,y) + \triangle T^{add}(x,y), \Lambda(\triangle T^{add}(x,y), x))$.
   Hence proved

3. From the definition of FSS.

$\square$

Theorem 5.2 *The forward fair path operator satisfies the following five properties:*

1. *If* $T^{sub} \subseteq T$,
   $FFP(T^{sub}(x,y), C(x), A(x)) \subseteq FFP(T(x,y), C(x), A(x))$.

2. *If* $C^{add} \supseteq C$,
   $FFP(T(x,y), C^{add}(x), A(x)) \subseteq FFP(T(x,y), C(x), A(x))$.

3. *If* $T^{add} \supseteq T$,
   $FFP(T^{add}(x,y), C(x), A(x)) \subseteq FFP(T(x,y), C(x), A(x)) + RRS(T^{add}(x,y), \Lambda(\triangle T^{add}(x,y), x))$

4. *If* $C^{sub} \subseteq C$,
   $FFP(T(x,y), C^{sub}(x), A(x)) \subseteq FFP(T(x,y), C(x), A(x)) + \overline{\triangle C^{sub}}$

5. $FFP(T(x,y), C(x), A(x)) \subseteq A(x)$

Proof.

1. FFP involves a LFP computation. $T^{sub} \subseteq T \Rightarrow f_{T^{sub}} \subseteq f_T$

   Where f denote the function in the LFP computation on page 82.

   $\Rightarrow FFP(T^{sub}(x,y), C(x), A(x)) \subseteq FFP(T(x,y), C(x), A(x))$.

2. FFP involves a LFP computation. $C^{add}(x) \supseteq C(x) \Rightarrow f_{C^{add}} \subseteq f_C$.

   Where f denote the function in the LFP computation on page 82.

   By Theorem 2.5 and 2.1

   $\Rightarrow FFP(T(x,y), C^{add}(x), A(x)) \subseteq FFP(T(x,y), C(x), A(x))$.

3. Consider $s \in FFP(T^{add}(x,y), C(x), A(x))$.

   $T^{add} \supseteq T \Rightarrow f_{T^{add}} \supseteq f_T$

   Where f denote the function in the LFP computation on page 82.

   $\Rightarrow FFP(T^{add}(x,y), C(x), A(x)) \supseteq FFP(T(x,y), C(x), A(x))$.

   (By Theorem 2.5 and 2.1.

   If $s \in FFP(T(x,y), C(x), A(x))$, then proved.

   Else if $s \notin FFP(T(x,y), C(x), A(x))$, then $\exists$ a newly created path to C containing s.

   (by definition of FFP).

   This path is not in the old FSM, hence it must either involve one of the new edges $\triangle T^{add}(x,y)$.

   $RRS(T^{add}(x,y), \Lambda(\triangle T^{add}(x,y), x))$ includes all paths through the new edges $\triangle T^{add}(x,y)$.

   $\Rightarrow s \in RRS(T^{add}(x,y), \Lambda(\triangle T^{add}(x,y), x))$.

4. Consider $s \in FFP(T(x,y), C^{sub}(x), A(x))$.

   $C^{sub} \supseteq C \Rightarrow f_{C^{sub}} \supseteq f_C$

   Where f denote the function in the LFP computation on page 82.

   $\Rightarrow FFP(T(x,y), C^{sub}(x), A) \supseteq FFP(T(x,y), C(x), A(x))$.

   (By Theorem 2.5 and 2.1.

   If $s \in FFP(T(x,y), C(x), A(x))$, then proved.

   Else if $s \notin FFP(T(x,y), C(x), A(x))$, then $\exists$ a newly created path originally violating C^{sub} containing s. (by definition of FFP).

   This path is not in the old FSM, hence it must have been removed originally because of the newly subtracted constraints.

$\overline{\triangle C^{sub}}$ includes all states (paths) that may violate the subtracted constraints $\triangle C^{sub}$.

$\Rightarrow s \in \overline{\triangle C^{sub}}(x)$.

5. From the definition of FFP.

□

Lemma 5.3 $f(Q) = FSS(T(x,y), FFP(T(x,y), C(x), Q(x))$ *is a monotonically decreasing function.*

Proof. $FFP(T(x,y), C(x), Q(x)) \subseteq Q(x)$ (from definition of FFP)
$FSS(T(x,y), Q(x)) \subseteq Q(x)$ (from definition of FSS)
$\Rightarrow FSS(T(x,y), FFP(T(x,y), C(x), Q(x))) \subseteq Q(x)$
$\Rightarrow f(Q) \subseteq Q$
Hence, $f(Q)$ is a monotonically decreasing function. □

5.4.3 Subtraction of Edges

Consider the system obtained after subtracting a set of edges from the transition relation. Subtracting an edge cannot make any unreachable state reachable, nor can it create a new cycle in the state transition graph. Thus, subtracting an edge can never add a new state to $Fair^+$. Figure 5.3 indicates that deleting edge ab can potentially remove all states in sets

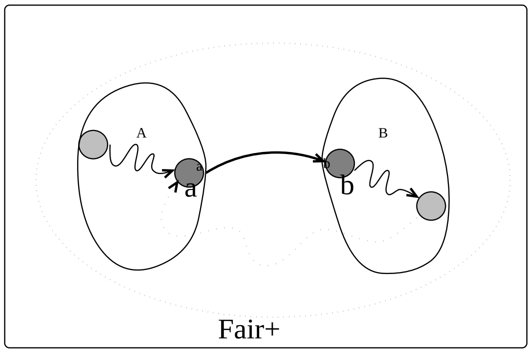


Figure 5.3: Deleting edge ab can potentially remove all states in A and B from $fair^+$

A and B from the set $Fair^+$. The following lemma formalizes this idea.

Lemma 5.4 *The set $Fair^{+new}$ for the new system obtained by deleting edges from the original transition relation is a subset of the $Fair^{+}$ of the original system.*

Proof. $FSS(T^{sub}(x,y), A(x)) \subseteq FSS(T(x,y), A(x))$.
$FFP(T^{sub}(x,y), C(x), A(x)) \subseteq FFP(T(x,y), C(x), A(x))$.
(From Theorems 5.1 and 5.2).
$\Rightarrow FSS(T^{sub}(x,y), FFP(T^{sub}(x,y), C(x), A(x))) \subseteq FSS(T(x,y), FFP(T(x,y), C(x), A(x)))$.
Using $f(Q) = FSS(T(x,y), FFP(T(x,y), C(x), Q(x)))$.
This function is monotonically decreasing. (from Lemma 5.3)
$\Rightarrow f_{T^{sub}} \subseteq f_T$.
$\Rightarrow GFP(f_{T^{sub}}(Q), A(x)) \subseteq GFP(f_T(Q), A(x))$.
$\Rightarrow Fair^{+new}(x) \subseteq Fair^{+}(x)$. □

Thus, if the only change induced in the system consists of subtraction of edges from the state transition graph, then the following algorithm can be used to generate $Fair^{+new}$ given the new transition relation and the old set of states comprising $Fair^{+}$.

Algorithm 5.3 *($T^{sub}(x,y), C(x), Fair^{+}(x)$)*

```
  $I(x)$ = Initial States
  $R^{sub}(x) = FR(T^{sub}(x,y), I(x))$
     $Fair_0(x) = Fair^{+}(x) \cap R^{sub}(x)$
     $f(Q(x)) = FSS(T(x,y), FFP(T(x,y), C(x), Q(x)))$
     $Fair^{+}(x) = GFP(f(Q), Fair_0(x))$
  return $Fair^{+}(x)$
```

Theorem 5.5 *If the only changes induced in the system consist of subtraction of edges from the state transition graph then Algorithm 5.3 is correct and returns $Fair^{+new}$*

Proof. From Lemma 5.4, Theorem 2.4 and [15] □

Computing the conjunction of R^{sub}, and $Fair^{+}$ in step 4. of the Algorithm 5.3 is not necessary to the computation, but increases the efficiency, if the computation of R^{sub} is not expensive. For the evaluation of the complexity of this algorithm, this operation is ignored.

Subtraction of edges can only remove states from $Fair^+$. At each pass of the fixed-point computation in Algorithm 5.3, at least one state, which was in the old $Fair^+$, but not in the new $Fair^+$, is removed. Thus, it converges in at most $\|Fair^+ - Fair^{+new}\|$ steps. But $\|Fair^+ - Fair^{+new}\| = \triangle_{output} \leq \triangle$, and each step takes $O(N)$ time. Hence, the algorithm completes in $O(N \cdot \triangle)$ time.

5.4.4 Addition of Edges

Consider the addition of a set of edges to the state transition graph. This may result in the creation of a new reachable cycle, whose states satisfy the fairness constraints. These states are not necessarily in $Fair^+$. Thus, addition of edges to the state transition graph may increase $Fair^+$. Figure 5.4 indicates that adding edge ab can potentially add all states in

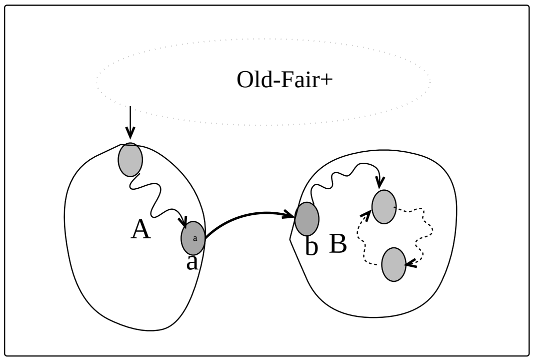


Figure 5.4: Adding edge ab can potentially add all states in A and B to $Fair^+$

sets A and B to the set $Fair^+$. However, we will prove that if the addition of edges results in the addition of one or more states to $Fair^+$, these states must satisfy at least one of the following conditions in the new transition system T^{add}:

- The state belongs to the set $Fair^+(x)$.
- The state can reach or be reached by one of the new transitions. This set $Fair^1$ is computed as:

  $Fair^1(x) = RRS(T^{add}(x,y), \Lambda(T^{add}(x,y), x))$ (Section 5.2).

Lemma 5.6 *The new set $Fair^{+new}$ is a subset of $Fair^{+} + Fair^{1}$, i.e. $Fair^{+new}(x) \subseteq Fair^{++}(x) \equiv Fair^{+}(x) + Fair^{1}(x)$.*

Proof. $FSS(T^{add}(x,y), A(x)) \subseteq FSS(T(x,y), A(x)) + RRS(T^{add}(x,y), \Lambda(\triangle T^{add}(x,y), x))$.
$FFP(T^{add}(x,y), C(x), A(x)) \subseteq FFP(T(x,y), C(x), A(x)) + RRS(T^{add}(x,y), \Lambda(\triangle T^{add}(x,y), x))$.
(Theorems 5.1 and 5.2).
Using $f_{T^{add}}(Q) = FSS(T^{add}(x,y), FFP(T^{add}(x,y), C(x), Q(x)))$.
This function is monotonically decreasing. (from Lemma 5.3)
Hence $f_{T}^{add}(RRS(T^{add}(x,y), \Lambda(\triangle T^{add}(x,y), x))) \subseteq RRS(T^{add}(x,y), \Lambda(\triangle T^{add}(x,y), x))$.
Using this and the distribution of GFP from Theorem 2.5.
$\Rightarrow FSS(T^{add}(x,y), FFP(T^{add}(x,y), C(x), A(x))) \subseteq FSS(T(x,y), FFP(T(x,y), C(x), A(x))) + RRS(T^{add}(x,y), \Lambda(\triangle T^{add}(x,y), x))$.
$\Rightarrow GFP(f_{T^{add}}(Q), A(x)) \subseteq GFP(f_{T}(Q), A(x)) + RRS(T^{add}(x,y), \Lambda(\triangle T^{add}(x,y), x))$.
$\Rightarrow Fair^{+new}(x) \subseteq Fair^{++}(x)$. □

If the only changes to the system consist of edge addition, the new set $Fair^{+new}$ can be computed as a two step process that first computes $Fair^{++}$, and then reduces it by using the Algorithm 5.3.

```
Algorithm 5.4 ($T(x,y), T^{add}(x,y), C(x), Fair^{+}(x)$)
  $Fair^{++}(x) = Fair^{+}(x) + RRS(T^{add}(x,y), \Lambda(T^{add}(x,y), x))$
  $Fair_0(x) = Fair^{++}(x)$
     $f(Q(x)) = FSS(T(x,y), FFP(T(x,y), C(x), Q(x)))$
     $Fair^{+}(x) = GFP(f(Q), Fair_0(x))$
  return $Fair^{+}(x)$
```

Theorem 5.7 *If the only changes induced in the system consist of addition of edges from the state transition graph then Algorithm 5.4 is correct and returns the new set $Fair^{+new}$.*

Proof. From Lemma 5.6, Theorem 2.4 and [15]. □

As noted in Section 5.3, the set of reachable states can be used to simplify the BDD for the transition relation. In Algorithm 5.3 the set of reachable states is not explicitly involved but may be used to simplify the transition relation BDD. It is important to note that for changes described in this section, reachability computations do not need to be carried out by starting at the initial states but need only proceed from the old set of reachable states R.

The Algorithm 5.4 converges in at most $\|Fair^{++} - Fair^{+new}\| = \triangle'$ steps. Since, $\triangle' \leq N$, for small changes, where N is the number of reachable states. Thus, the complexity of this algorithm is $O(N \cdot \triangle')$. Assuming $\triangle' \leq N$, this is faster than running the non-incremental algorithm from the beginning.

5.4.5 Addition of Fairness Constraints

The set $Fair^{+}$ satisfies all the fairness constraints. If new fairness constraints are only added, then the new set $Fair^{+new}$ must satisfy all of the older constraints as well as the new ones. The set $Fair^{+new}$ must be a subset of the old $Fair^{+}$.

Lemma 5.8 *If additional fairness constraints are imposed on the system, then* $Fair^{+new}(x) \subseteq Fair^{+}(x)$.

Proof. $FFP(T(x,y), C^{add}(x), A(x)) \subseteq FFP(T(x,y), C(x), A(x))$.
(Theorems 5.1 and 5.2).
$\Rightarrow FSS(T(x,y), FFP(T(x,y), C^{add}(x), A(x))) \subseteq FSS(T(x,y), FFP(T(x,y), C(x), A(x)))$.
Using $f(Q) = FSS(T(x,y), FFP(T(x,y), C(x), Q(x)))$.
This function is monotonically decreasing. (from Lemma 5.3)
$\Rightarrow f_{C^{add}} \subseteq f_C$.
$\Rightarrow GFP(f_{C^{add}}(Q), A(x)) \subseteq GFP(f_C(Q), A(x))$.
$\Rightarrow Fair^{+new}(x) \subseteq Fair^{+}(x)$. $\Box$

If the only change to the system consists of addition of constraints, the algorithm for computation of the new $Fair^{+new}$ is:

Algorithm 5.5 *($T(x,y), C^{new}(x), Fair^{+}(x)$)*

```
  $Fair_0(x) = Fair^{+}(x)$
     $f(Q(x)) = FSS(T(x,y), FFP(T(x,y), C(x), Q(x)))$
     $Fair^{+}(x) = GFP(f(Q), Fair_0(x))$
```

```
    **return** $Fair^+(x)$
```

Theorem 5.9 *If the only changes to the system consist of addition of constraints then Algorithm 5.5 is correct and returns the new set* $Fair^{+new}$.

Proof. From Lemma 5.8 Theorem 2.4, and [15]. $\square$

Using the same reasoning as Section 5.4.3, this algorithm has a time complexity of $O(N \cdot \triangle)$, where N is the number of reachable states.

The addition of constraints can very easily be used in conjunction with the addition and subtraction of edges. If edges are deleted, in addition to adding constraints, algorithm 5.3 can be used with the FFP operator (including the new constraints) to compute the new set $Fair^{+new}$. If edges are added then Algorithm 5.4 can be used in conjunction with the new FFP operator. The following lemmata formalize this idea.

Lemma 5.10 *If additional fairness constraints are imposed on the system, and edges are only subtracted from the transition structure then* $Fair^{+new}(x) \subseteq Fair^+(x)$.

Proof. From Lemma 5.4 and Lemma 5.8 $\square$

With the previous lemma, it is easily observed that the subtraction of edges and addition of constraints can be simultaneously handled by using Algorithm 5.3 (for the subtraction of edges) with the additional caveat that the FFP operator is modified to include the new constraints.

Lemma 5.11 *If additional fairness constraints are imposed on the system, and edges are only added to the transition structure then* $Fair^{+new}(x) \subseteq Fair^{++}(x)$, *where* $Fair^{++}$ *is as defined in Lemma 5.6.*

Proof. From Lemmas 5.6 and 5.8 $\square$

In a similar manner to the previous analysis, it is observed that the addition of edges and addition of constraints can be simultaneously handled by using Algorithm 5.4 (for the addition of edges) with the additional caveat that the new FFP operator (as defined in Lemma 5.8) is used for the computation.

5.4.6 Subtraction of Fairness Constraints

The set $Fair^+$ contains states involved in infinite behavior that satisfy all fairness constraints C_i. If some constraint $C_i = F^\infty(U_i) + G^\infty(V_i)$ is subtracted, $Fair^+$ still contains states that are involved in infinitary behavior, and satisfy all constraints $C_j \neq C_i$ (as well as C_i). Thus, the set $Fair^+ \subseteq Fair^{+new}$. In addition to the states in $Fair^+$, $Fair^{+new}$ also contains states that may be in infinitary behavior that violates the deducted constraint C_i. Such states are definitely a subset of $(\overline{U_i + V_i})$; namely states not in U_i or V_i.

Since more than one constraint may be subtracted, let $C_i \in \triangle C^{sub}$ denoted the entire set of subtracted constraints. Also, we denote $\overline{\triangle C^{sub}}(x) = \bigcup_{C_i \in \triangle C^{sub}} (\overline{U_i(x) + V_i(x)})$.

Lemma 5.12 *If constraints $C_i = F^\infty(U_i) + G^\infty(V_i)$, $i \in S$ are subtracted from the set of original constraints, then the set*
$Fair^{+new}(x) \subseteq Fair^{++}(x) = Fair^+(x) + \overline{\triangle C^{sub}}(x)$.

Proof. $FFP(T(x,y), C^{sub}(x), A(x)) \subseteq FFP(T(x,y), C(x), A(x)) + \overline{\triangle C^{sub}(x)}$.
(From Theorem 5.2).
Using $f(Q) = FSS(T(x,y), FFP(T(x,y), C(x), Q(x)))$.
This function is monotonically decreasing. (from Lemma 5.3)
Hence $f(\overline{\triangle C^{sub}}(x)) \subseteq \overline{\triangle C^{sub}}(x)$.
Using this and the distribution of GFP from Theorem 2.5.
$\Rightarrow FSS(T(x,y), FFP(T(x,y), C^{sub}(x), A(x))) \subseteq FSS(T(x,y), FFP(T(x,y), C(x), A(x))) + \overline{\triangle C^{sub}(x)}$.
$\Rightarrow GFP(f_{C^{sub}}(Q), A(x)) \subseteq GFP(f_C(Q), A(x)) + \overline{\triangle C^{sub}}(x)$.
$\Rightarrow Fair^{+new}(x) \subseteq Fair^{++}(x)$. $\square$

This leads to the following algorithm for changes, where constraints are only subtracted from the system.

Algorithm 5.6 *$(T(x,y), C^{new}(x), \triangle C^{sub}(x), Fair^+(x))$*

```
  $Fair^{++}(x) = Fair^+(x) + \overline{\triangle C^{sub}}(x)$
  $Fair_0(x) = Fair^{++}(x)$
      $f(Q(x)) = FSS(T(x,y), FFP(T(x,y), C(x), Q(x)))$
```

```
    $Fair^+(x) = GFP(f(Q), Fair_0(x))$
  **return** $Fair^+(x)$
```

If, in addition to constraint subtraction, edges were added (added states $= (\Lambda(\triangle T^{add}(x,y),x))$) to the transition structure, then states in the new $Fair^{+new}$ must satisfy at least one of the following conditions in the new Transition system T^{add}:

- The state belongs to the set $Fair^+$.
- The state can reach or be reached by one of the new transitions. This set $Fair^1$ is computed as:

  $Fair^1(x) = RRS(T^{add}(x,y), \Lambda(\triangle T^{add}(x,y),x))$ (Section 5.2).
- The state violates deleted constraints.

  $Fair^2(x) = \overline{\triangle C^{sub}}(x) = \bigcup_{C_i \in \triangle C^{sub}} (\overline{U_i(x) + V_i(x)})$ (Section 5.2).

Lemma 5.13 *If fairness constraints are subtracted from the system, and edges are only added to the transition structure then*
$Fair^{+new}(x) \subseteq Fair^{++}(x) = Fair^+(x) + RRS(T^{add}, \Lambda(\triangle T^{add}(x,y),x)) + \overline{C^{sub}}(x)$

Proof. From Lemma 5.6 and Lemma 5.12. □
If subtraction of constraints is used in conjunction with addition of edges, then the following algorithm describes the computation of the new $Fair^{+new}$.

Algorithm 5.7 *($T^{add}(x,y), T(x,y), C^{sub}(x), \triangle C^{sub}(x), Fair^+(x)$)*

```
  $Fair^{++}(x) = Fair^+(x) + RRS(T^{add}(x,y), \Lambda(\triangle T^{add}(x,y),x)) + \overline{C^{sub}}(x)$
  $Fair_0(x) = Fair^{++}(x)$
    $f(Q(x)) = FSS(T(x,y), FFP(T(x,y), C(x), Q(x)))$
    $Fair^+(x) = GFP(f(Q), Fair_0(x))$
  **return** $Fair^+(x)$
```

This algorithm has a complexity of $O(N \cdot \triangle')$, where N is the number of reachable states. The next section deals with putting these individual algorithms together to form a general algorithm which handles any change to the system.

5.5 General Algorithm

We describe an incremental algorithm for language containment, when a general set of changes consisting of deletion and addition of edges from the transition structure and addition and subtraction of constraints, is applied to the system.

We begin by separating the augmented transition relation T^{new} into T^{sub}, which consists of the original transitions relation T minus all transitions which were removed in T^{new} and T^{add} which is seen as T^{sub} plus all the transitions, which are added in T^{new}.

The change to the system can be seen as a two stage process; in the first stage, constraints are added and edges are only subtracted from the system. In the second stage constraints are only subtracted and edges are only added to the transition structure obtained from the previous stage. The first stage computes an intermediate $Fair^{+new1}$ under the assumption that the only changes consist of edge subtraction and constraint addition and for this stage, the transition structure T^{sub} is used. The second stage computes the new $Fair^{+new1}$ using as input the intermediate $Fair^{new1}$ and the new transition structure T^{add}.

For a more detailed description of how to compute T^{add}, T^{sub} under different situations refer to the discussion in Section 4.4.1.

Modifications can be made at many different levels. The designer may input the changes in a high level language (e.g. Verilog). Alternately, he/she might choose to augment individual subprocesses in the system of interacting processes by directly modifying the data-structure that stores their transition relations and constraints. In our implementation, the designer is allowed to directly change the individual transition relations, or input process constraints and new processes via the intermediate 'Pif' [68] format.

5.5.1 Iterative Verification

Another alternate system of changes comes from the iterative algorithms of Balarin et-al. [13]. They use iterative methods in the formal verification of digital systems. In order to illustrate how incremental language containment may be used in an iterative system, we consider two cases: the compositional iterative algorithm and the iterative verification of timed automata algorithm of Balarin [13].

Given a system to be verified, Balarin describes an iterative algorithm that begins with some initial abstraction of the machine, calls verification (language containment), and uses the resulting answer to this call to modify the abstraction and make it more accurate. The iterative verification of timed automata functions begins with an untimed machine as the initial abstraction, and the compositional algorithm begins with some subset of the component machines.

Balarin makes changes to the initial system by composing with another FSM. Note that if a designer modifies any machine by composing it with another, it is equivalent to the addition and subtraction of edges from it (our basic unit of change).

If the abstraction passes language containment, then the original machine also passes language containment. However, if the abstraction fails language containment, then the error trace (or report of some unacceptable behavior) in the abstraction is examined. If this behavior is indeed present in the actual machine, we know language containment has indeed failed. If it is not present, another component is created and composed with the abstraction, in order to eliminate this behavior, and refine the abstraction. Thus, our incremental procedure may be applied at each iteration by using the modifying machine to compute changes to the original abstraction, and using the original answer to language containment and change information as input to the incremental algorithms of this chapter.

5.5.2 Incremental Language Containment

The general algorithm for computation of $Fair^{+new}$ is based on Algorithm 5.3 and Algorithm 5.4. The general incremental language containment (ILC) algorithm:

```
Algorithm 5.8 : Incremental_Language_Containment
  $Fair^+(x)$ = Incremental_Compute_Fair$^+$
  if $Fair^+(x)$ is empty return(PASS)
  else return(FAIL)
```

where the algorithm for the incremental computation of $Fair^+$ is:

```
Algorithm 5.9 : Incremental_Compute_Fair^+
  $Fair^{+new1}(x) = Algorithm\ 5.3(T^{sub}(x,y), C^{add}(x), Fair^{+}(x))$
  $Fair^{+new}(x) = Algorithm\ 5.7(T^{add}(x,y), T^{sub}(x,y), C^{sub}(x), \triangle C^{sub}(x), Fair^{+new1}(x))$
return $Fair^{+new}(x)$
```

Theorem 5.14 *Algorithm 5.9 is correct and returns the new set $Fair^{+new}$.*

Proof. The first stage of the algorithm does not involve addition of edges, hence the use of Algorithm 5.3 is valid and returns the correct set of fair states, $Fair^{+new1}$ for this subproblem to the next stage (refer to Theorem 5.5 and Lemma 5.10). The second stage does not involve the subtraction of edges; hence the use of Algorithm 5.7 is valid and the correct set $Fair^{+new}$ is returned (refer to Theorem 5.7, Lemma 5.11 and Lemma 5.13) □

5.6 Experiments and Results

We have implemented the algorithms described in the previous section and tested these on a set of verification benchmarks. Each example was modified, and the $Fair^{+}$ was recomputed for a general change to the system, which consists of addition and subtraction of edges and constraints. The actual edges/constraints that were added or subtracted from the transition relation are arbitrary, and were chosen so as to make the system pass the language containment check.

The first row in each table reports the name of the example, and the iteration number. The second row reports the time taken by the incremental language containment (ILC) algorithm; this includes the time for incremental update of input data, and re-initialization. The last row reports the time for the non-incremental (NLC) algorithm with the non-incremental update; this includes the time for non-incremental input of data and initialization. The last column reports the total incremental, and non-incremental times, summed over all iterations.

We ran the incremental, and non-incremental algorithms on four examples, and made 5 successive sets of changes. The columns labelled with integers $i = 1, 2, 3, 4, 5$ report the times taken on iteration (set of changes) i. The first example, Gigamax, was a description of the gigamax distributed multiprocessor, using a shared memory architecture. The second example, Scheduler, describes a version of the scheduler example by Milner [69], and the system consists of a token ring, where element of the ring, called a cell, communicates with its "job", and its two nearest neighbor cells. The third example, Tcp, describes a simplified version of the TCP/IP communication protocol. The final example, Idlc, describes an industrial data link controller example. All the examples were written in Verilog, and translated into the *blif-mv* format using the vl2mv translator [63]. All successive incremental changes were made directly to the system within the HSIS environment.

Gigamax	1	2	3	4	5	Total
ILC[1]	25.0	9.1	35.2	22.8	25.1	117.3
NLC[2]	42.4	29.1	53.9	41.1	44.9	211.5
Scheduler	1	2	3	4	5	Total
ILC	18.5	0.8	21.7	8.5	19.6	69.2
NLC	25.4	7.7	27.6	23.8	29.2	113.8
Tcp	1	2	3	4	5	Total
ILC	40.6	14.6	97.3	22.3	8.4	183.2
NLC	447.7	420.6	463.9	431.0	417.2	2180.5
Idlc	1	2	3	4	5	Total
ILC	247.1	369.8	463.2	176.6	-	1256.7
NLC	2403.2	2659.4	2461.6	2573.3	-	10094.5

Table 5.1: Incremental Vs. Non-Incremental Language Containment(in seconds)
1: ILC =Incremental algorithm and incremental data update
2: NLC =Non-incremental algorithm and non-incremental data input

The changes themselves were made by examining an error trace (describing some non-empty behavior in the system) generated, and deleting and adding edges and constraints so as to remove the particular error trace, and eventually make the system pass language containment.

The results show that the incremental algorithm was always considerably faster than the non-incremental algorithm.

It should be noted (from the results)that as the size of the example increases (from Gigamax to Idlc), so does the gain from using an incremental algorithm.

Chapter 6

Incremental Model Checking

6.1 Introduction

In this chapter we will discuss how to extend the arguments of Chapter 4 and 5 to Model Checking, an alternate method for formal verification.

In model checking, the property is written as a formula in some temporal logic [70], and the system or design is represented as an FSM. Design verification consists of checking whether the design FSM is a model for the formula.

Checking whether the formula holds on the design FSM is accomplished by traversing it. We are restricting ourselves to Computation Tree Logic (CTL) [19], where formulae are defined on paths in the FSM. The paths in an FSM can be un-wrapped and represented as a tree for computation, and properties in CTL are defined with respect to these paths. For example, "For all paths...". CTL formulae are recursively defined in terms of sub-formulae, which themselves are CTL formulae. The logic consists of path quantifiers (for all paths, there exists a path), forward time temporal operators (at the next state, globally, finally, until), Boolean operands (not, or, and..), and labels on the states (*p is true*). CTL formulae must be defined on some finite state machine. The finite state structure that the CTL formula is defined on is also referred to as Kripke structure. The formula is said to fail on the FSM, if there is a witness path (or set of paths) in the FSM that does not satisfy the formula. For example, the formula "*for all paths globally p is true*", fails on the given FSM, if it can be shown that there exists a path from some initial state along which there is one state where p is not true. The formula *p* in turn may be another CTL formula, which was previously computed. CTL formulae are evaluated by recursively evaluating states that satisfy a sub formula, and using them (as new labels) to compute states satisfying the formula. The evaluation process consists of traversing the finite state machine representing the design.

Current techniques [9] for model checking perform it as a single pass, and often result in redundant re-computation over multiple passes. This particularly relevant, because verification tends to be an expensive and time consuming process. Another strong motivation for incremental model checking comes from the fact that the design rarely checks a single property; usually the he/she tests a large suite of specifications. It is very possible that we may save considerable effort by re-using common information between two different property evaluations.

We introduce the concept of *incremental model checking*, which allows multiple changes to the system but runs the entire model checking algorithm only once, and propagates successive changes or increments from the latest solution. Given an answer to an instance of the model checking problem: "*Does the given design represented as a finite state machine, satisfy the property represented as a formula in temporal logic ?*", we aim to use it and information about incremental changes made to the input problem to compute the answer to a new instance of the problem. We plan on spending less time and effort in this computation than if the entire model checking algorithm was run on the new problem. All algorithms in model checking consist of the computation of fixed points on the states of the FSM representing the design. To accomplish this, we use the insights of Theorem 2.4 and 2.2.

As with the previous chapters in this thesis, model checking is carried out under the context of interacting finite state machines.

The chapter is organized as follows. We define some additional terms to be used in Section 6.2. Next we describe the syntax of CTL and model checking in Section 6.3. Our work begins in Section 6.4, by recognizing that all (small) changes to the system can be translated to the addition and subtraction of edges, states and constraints and relabeling of atomic propositions. Next, we analyze how each of these changes to the system can be propagated to get new answers to and prove the correctness of these techniques. The section summarizes how incremental changes are classified and how each particular change can be handled individually. We briefly discuss the possibility of Canonical CTL structures to minimize re-computation of information. We describe the procedures for handling individual classes of change, and merge them to get a general algorithm for handling any change to the system.

It is important to keep in mind that all operations are to be carried out in the context of the Binary Decision Diagram (BDD) data structure[40] defined in Chapter 2.

6.2 Some Terminology for Model Checking

In this chapter we will be referring extensively to the definitions of a *finite state machine* (FSM) or finite automaton, its inputs, outputs, states, transition relation $T(x, y)$, and initial states $I(x)$. These have already been discussed in Section 4.2. In general, let x represent the present state and y represent the next state. $T(x, y)$ represents the transition relation, which defines a relationship between present states (x variables) and next states (y variables) in the state transition graph, irrespective of input and output. Let LFP and GFP refer to the least and greatest fixed point operators defined in Chapter 2.

In addition, we will be discussing this chapter in the context of ω-automata, where *runs* r, i.e. a sequence of states beginning at the initial states, are infinite (Defined in Section 5.2).

In particular, we will be examining the model checking problem in the context of automata whose acceptance conditions consist of sets of Buchi conditions.

Definition 39 Buchi Constraints *A Buchi automata is characterized by Buchi acceptance conditions, which consist of a set of states $U_i = s_1 \ldots s_n$ at least one of which must be traversed infinitely often in any accepted run. The results presented here are for a set of Buchi acceptance conditions $C = U_1 \ldots U_n$; however, it should be noted that they are easily generalized to other classes of fairness constraints.*

This section describes the computation procedure for some important operators required in this chapter. In particular, the following three operators will be extensively used in this chapter.

The **Forward Stable Set Operator** FSS(T(x,y),A(x)) (defined in Section 5.2) when applied to a set of states A returns a set of states in A, which are on a cycle or can reach a cycle in A. Alternately, the FSS operator removes from A all those states which have no successors states (next states) in the transition structure. The following algorithm is used to compute the Forward Stable Set operator FSS :

```
$f(Q(x)) = Q(x) \cdot (\exists_y T(x,y).A(x) \cdot Q(y))$
**FSS**$(T, A)$
  **return** $(GFP(f(Q), A(x)))$
```

The forward fair path operator in the context of Buchi automata is defined as follows:

Forward Fair Path Operator: Given $T(x,y)$ the transition relation, $A(x)$ a set of states and $C(x)$, a set of fairness constraints, the *forward fair path operator* or $FFP(T(x,y), C(x), A(x))$, returns a subset of states in $A(x)$ which are on a fair path. For our analysis, $C(x)$ are sets of Buchi fairness constraints in the form $C_i = F^{\infty}(U_i)$. Hence, FFP returns those states a in A such that for each U_i, there is a path in A from a to some state in U_i. Note that this operator returns just a path and not necessarily an infinite path. The FFP operator can be computed by using the following algorithm:

```
$f(Q(x)) = (\exists_y T(x,y).A(x) \cdot Q(y) + Q(x))$
**FFP**$(T, C, A)$
  **return**$(\prod_{i, c_i \in C(x)} (LFP(f(Q), U_i(x) \cdot A(x))))$
```

Also recall that given a relation $R(x,y)$, $\Lambda(R(x,y), x)$ denotes the **projection** of the relation on the x variables, i.e. $\Lambda(R(x,y), x) = \exists_y R(x,y)$. If $R(x,y)$ denotes a set of edges in the FSM (e.g. $R(x,y) = T(x,y)$), then $\Lambda(R(x,y), x)$ denotes the projection of set of predecessor states of the given edges. This has been defined in Section 5.2.

In the context of Buchi automata, the $Fair^+$ set, i.e. the set of states that are involved in all accepted behavior (sometimes called "bad" behavior) in the automata, can be obtained by alternately applying the FFP and FSS operators as described in Section 5.3. A simpler procedure can be used for Buchi in particular, as shown in [9] and [71], however for this discussion, we will use the more general form of Algorithm 5.2.

6.3 CTL Model Checking

CTL or Computational Tree Logic, created by Clarke et al [19], is a propositional temporal logic of branching time. CTL is defined on the computational tree created by expanding all the paths in the FSM (also called a Kripke structure) that the formula is defined on. The syntax of CTL is defined recursively using a *path quantifier*, a *temporal operator*, *Boolean operands* and smaller CTL formulae. The smallest CTL formula is called an *atomic proposition* and consists of a label on the states of the FSM.

Since CTL is defined on paths, each formula must have a *path quantifier.* There are two path quantifiers:

- A (for all paths): The formula is said to hold at a given state if for all paths from the state the clause following A is true.
- E (exists a path): The formula is said to hold at a given state if there exists a path from the state where the clause following E is true.

Since CTL is a temporal logic, there are four forward time temporal operators:

- G (Globally): The clause following the G operator holds everywhere (along path or paths of concern).
- F (Finally): The clause following the G operator holds sometime in the future (along path or paths of concern).
- X (Next state): The clause following the X operator holds in the next state (along path or paths of concern).
- U (Until): The clause preceding the U operator must hold until the clause succeeding the operator holds (along all paths of concern).

Using the above CTL formula are defined recursively:

- All atomic propositions (or labels on states) are CTL formulae.
- If p and q are CTL formulae, so are

1. $EGp \qquad AGp$
2. $E(pUq) \quad A(pUq)$
3. $EXp \qquad AXp$
4. $\overline{p} \qquad EFp$
5. $p+q \qquad AFp$

All CTL formulae can be written in terms of the set $EGp, E(pUq), EXp, p+q, \overline{p}$ (e.g. $AFp = \overline{EX\overline{p}}$). The set of *Normalization* rules which convert all formula to the above set are:

Rules 6.1 .

1. $AXp \equiv \overline{EX\overline{p}}$
2. $EFp \equiv E(TrueUp)$
3. $AFp \equiv \overline{EG\overline{p}}$
4. $AGp \equiv \overline{E[TrueU\overline{p}]}$
5. $A(pUq) \equiv \overline{E(\overline{q}U(\overline{p}+\overline{q}+EG\overline{q}))}$

A CTL formulae specified as the above can be read in and stored as a *CTL parse tree*, where each node represents a formula and its immediate fanins the component sub formulae.

Definition 40 CTL Parse Tree*: A CTL parse tree is a tree graph where every node may have the following labels; a path quantifier (E, A) and an temporal operand (G, U, X), or a Boolean operand $(\overline{p}, +)$, or an atomic proposition (p is true). A node labeled with an atomic proposition has no children, and is a leaf node of the tree. Other nodes may have one or two children depending on the label (EGp has labels EG and one child p).*

Figure 6.1 describes a CTL parse tree for the CTL formula $E(pUq)$. Note that CTL parse trees are not a canonical form; the semantically same formula may be represented by many different trees.

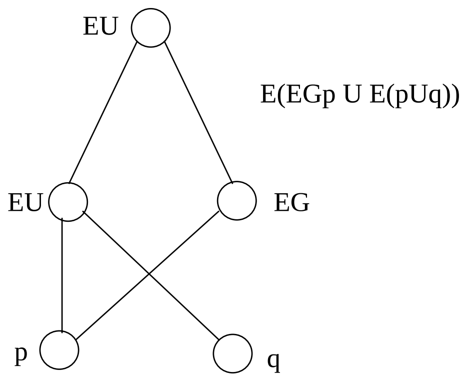


Figure 6.1: A CTL parse tree

Definition 41 CTL Model Checking: *Given a finite structure (FSM) M and a formula f in CTL, model checking determines whether this M defines a model of f, i.e. loosely speaking, whether the initial states of the FSM are part of the set of states satisfying f. If M satisfies f is denoted as $M \models f$.*

The states satisfying various CTL formulae are computed as fixed points (greatest or least fixed points depending on the formula). For the purpose of this work, we will be restricting ourselves to Buchi acceptance conditions, i.e. we will be performing CTL model checking on systems represented as Buchi automata. Let $E_cGp(x)$ denote the set of states satisfying the formula EGp under Buchi fairness $C = U_1 \ldots U_n$. Similarly, $E_C(pUq)(x), E_CXp(x), (p+q)_C(x)$ denote the states satisfying $E(pUq), EXp), (p+q)$ respectively under Buchi acceptance C. Given $p(x), q(x)$ the sets of states satisfying the sub-formulae p and q respectively, the set of states satisfying the CTL formulae can be computed as:

1. States satisfying E_CGp denoted as $E_CGp(x)$:

$$\begin{aligned} f(Q(x)) &= FSS(T(x,y), FFP(T(x,y), C, Q(x))) \\ E_CGp(x) &= GFP(f(Q), p(x)) \end{aligned}$$

   From lemma 5.3 $f(Q)$ is monotonically decreasing.

2. $Fair^+$ states denoted as:
   $Fair^+(x) = E_CG(TRUE)$

3. States satisfying $E_C X p$ denoted as:
   $E_C X p(x) = BImg(p(x) \cdot Fair^+(x))$:

4. States satisfying $E_C(pUq)$ denoted as $E_C(pUq)(x)$:

$$\begin{aligned} f(Q(x)) &= q + p.EX(Q(x)) \\ E_C(pUq)(x) &= LFP(f(Q), q(x) \cdot Fair^+(x)) \end{aligned}$$

The states satisfying $(p+q)_C(x) = p(x) + q(x)$.

Note that item 1 is the most complex computation, because it requires a fixed point within each iteration.

CTL model checking proceeds iteratively, from the leaves of the CTL parse tree to the root. The atomic propositions correspond to the leaves; their immediate fanouts are formulae defined on the atomic propositions and so on. The following is a non-incremental algorithm for CTL model checking:

```
Algorithm 6.1 Non_Incremental_Model_Checking(T(x,y),I(x),p)
  $Fair^+$ = Compute_Fair (defined in Section 5.3)
  $Solution$ = Model_Check($p, Fair^+(x)$) $\bigcap I(x)$
  if $Solution \neq \emptyset$ is empty return(PASS)
  else return(FAIL)
```

Model Checking is done using the following algorithm:

```
Algorithm 6.2 Model_Check($f, Fair^+(x)$)
  Case:
  $f$ is an atomic proposition
  $f(x)$ = Set of states satisfying $f$.
        return $f(x) \cdot Fair^+(x)$
  $f \equiv EGp$
     $p(x)$ = Model_Check($p$)
     $f(Q(x)) = FSS(T(x,y), FFP(T(x,y), C(x), Q(x)))$
```

```
    $E_CGp(x) = GFP(f(Q), p(x))$
        **Return** $E_CGp(x)$
$f \equiv E(pUq)$
    $p(x) = Model\_Check(p)$
    $q(x) = Model\_Check(q)$
    $f(Q) = p(x) + q(x) \cdot EX(Q(x))$
    $E_C(pUq)(x) = LFP(f(Q), q(x) \cdot Fair^+(x))$
        **Return** $E_C(pUq)(x)$
$f \equiv EXp$
    $p(x) = Model\_Check(p)$
    $E_CXp(x) = Fair^+(x) \cdot \exists_y T(x,y) \cdot p(y)$
        **Return** $E_CXp(x)$
$f \equiv p + q$
    $p(x) = Model\_Check(p)$
    $q(x) = Model\_Check(q)$
    $(p+q)_c(x) = p(x) + q(x)$
        **Return** $(p+q)_c(x)$
$f \equiv \overline{p}$
    $p(x) = Model\_Check(p)$
        **Return** $\overline{p(x)} \cdot Fair^+(x)$
```

If $M = (Q, \Sigma, \Gamma, T, I)$ and $M \models f$, then $Non\_Incremental\_Model\_Checking(T(x,y), I(x), f)$ returns TRUE.

Though not explicitly stated in the above algorithm, the set of reachable states can also be used to minimize the transition relation BDD. This simplification results in a considerable speedup and will be used throughout this chapter without an explicit mention.

6.4 Incremental Model Checking

As discussed in the previous section CTL model checking consists of GFP and LFP computations. We discuss the incrementalizing of LFP computations in Chapter 4. For an

LFP, we begin at some initial state A, and apply a monotonically increasing function to it until we reach a final convergent set. In such an LFP computation, if any subset of the final set, which includes the initial set is supplied to the self same LFP computation, it returns the same final answer. However, it will take fewer iterations to converge, because it is closer to the final set. Since, the LFP now takes fewer iterations to converge, it is faster. We have also discussed a similar idea for incrementalizing GFP computations in Chapter 5.

Thus, if the answer to a previous iteration of this LFP (without changes) is B', we intend to use this answer, as well as information about changes to the system to compute a subset C to the new answer B that is larger than the initial set A. The better the C chosen, the more efficient the incremental algorithm.

We rely extensively on Theorem 2.2 and 2.4

6.4.1 Characterizing Incremental Changes

We have determined changes to the system to be one of 5 types.

1. **Changes to the Property.**
   - **Changes to the CTL formula.** The designer might modify the property to be checked or want to check another CTL property. Both these changes may be regarded as changes to the CTL formula. Since CTL formulae are commonly represented as CTL parse trees; these changes may be regarded as changes to the parse tree structure. A trivial example of such a change would be checking the property $EX(EGp)$ as compared to checking EGp. Notice that the set EGp from the previous computation may used as a starting point to re-check $EX(EGp)$. In this chapter, we propose far more extensive ways of preserving information.
   - **Changes to the set of states marked with a proposition.** This set is characterized by the addition and subtraction of states satisfying an atomic proposition. Let $\triangle p^{sub}$ denote the set of states subtracted, and $\triangle p^{add}$ denote the set of states added to states marked p. These changes may also occur as a result of other classes of changes. In particular, if the formula $EG(p + \triangle p)$ was checked as compared to EGp, then the set of states marked p have been changed

to $p + \triangle p$. As result states marked $q = EGp$ are incrementally updated to $q + \triangle q = EG(p + \triangle p)$. At the next level in the CTL formula states marked q are assumed to change to $q + \triangle q$, and the entire argument is repeated. This class may save computation while propagating a change up the CTL parse tree.

2. **Changes to the System.**

   - **Changes to the FSM transition relation.** As discussed previously in Section 5.4.1, this set is characterized by the addition and subtraction of edges from the design FSM. We let $\triangle T^{sub}$ denote the set of edges subtracted from the original FSM. As a result of this subtraction alone, we can obtain an intermediate set T^{sub}. We let $\triangle T^{add}$ denote the set of edges added to the original FSM, and as a result of this addition to T^{sub}, we can obtain the final transition relation $T^{add} = T^{new}$. The addition and subtraction of states from the FSM can be characterized in terms of edges.
   - **Changes to the acceptance conditions.** This class is characterized by the addition and subtraction of constraints from the set of Buchi acceptance sets (See Section 5.4.1). We denote the added constraints by $\triangle C^{add}$ and the subtracted constraints by $\triangle C^{sub}$. The new set of constraints may be denoted by C^{new}. The designer uses this change to ignore certain bad behavior after it has been detected, and examine the remainder for further bad behavior, before attempting to change the design.

     In general, Buchi constraints can also be regarded as CTL formulae. Thus, you can deal with them as with CTL formulae.

6.4.2 Identifying Changes to the CTL parse Tree

Changes to the CTL formulae are characterized by changes to the CTL parse tree. We hope to preserve information by detecting common substructure between two CTL parse trees. However, unlike the work in Chapter 3 we are given an input correspondence as the atomic propositions are known. It is debatable whether significant gains can be obtained by comparing CTL formula that are identical except for the atomic propositions involved, since the truth or falsity of a formula is dependent on the FSM model involved. Consider

Figure 6.2, which shows the parse tree of two different formulae.

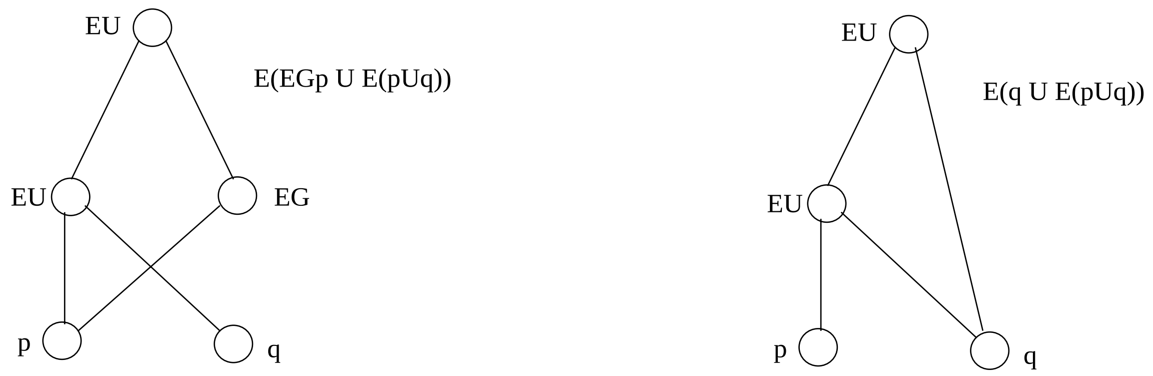


Figure 6.2: Changes to the CTL formula

The sub-formulae $E(pUq)$ are identical in both, thus information can be preserved by using the answer for set $E(pUq)$ in $E(EGp\ U\ E(pUq))$.

Given the old and new CTL formulae, we need to detect whether they have identical subformulae. In general, the same CTL formula can be represented as many semantically identical but syntactically different forms. For example, $EG(g + h) \equiv EG(EGg + h)$. This motivates the need to have canonical forms, which give syntactic equivalence for all semantically identical formulae.

Definition 42 *A representation of a CTL formula is a* **Canonical** *CTL Form if two formulae that are semantically identical (E.g. $EGp \equiv \overline{AF\overline{p}}$), always have the same representation in this form.*

Unfortunately, we will show that to construct such a canonical form would take $\Omega(exp(|f|)$ for a formula f, i.e. at least exponential time in the size of the formula, and hence is not practical.

In order to show this, we use the following theorem from [22]. **Satisfiability** of a CTL formula f is the problem of checking whether there exists any FSM model on which the given CTL formula f holds. This theorem shows that it takes at least exponential time to show that a CTL formula is satisfiable (i.e. $f \not\equiv \emptyset$). For a detailed exposition on satisfiability, completeness and other related terms refer to Garey and Johnson [72].

Theorem 6.1 *[22]The problem of testing satisfiability for a CTL formula is complete for deterministic exponential time, i.e. satisfiability is $\Theta(exp(|f|)$.*

Proof. (Sketch) $\Rightarrow$ Construct an automaton that accepts this formula (tableau), and show that this is exponential in size.
$\Leftarrow$ Reduction from alternating polynomial space bound Turing machines [73]. $\square$

Next, we construct a lemma to show that satisfiability of f reduces to checking the equivalence of CTL formulae.

Lemma 6.2 *The problem of checking whether two CTL formulae are equivalent is at least as hard as checking whether a formula is satisfiable, i.e. equivalence is $\Omega(exp(|f|))$.*

Proof. Assume f is satisfiable.
$\Rightarrow \exists M, \quad M \models f.$
$\Rightarrow \overline{f \equiv \emptyset}.$
Hence satisfiability reduces to equivalence. Satisfiability is $\Theta(exp(|f|)$, and hence equivalence is $\Omega(exp(|f|))$. $\square$

Hence, we combine these results to get the following theorem.

Theorem 6.3 *Finding a canonical form for a CTL formula f is at least $\Omega(exp(|f|)$.*

Proof. From Theorem 6.1, satisfiability of f is $\Omega(exp(|f|))$, and from Lemma 6.2, equivalence is at least as hard as satisfiability.
$\Rightarrow$ Equivalence is at least $\Omega(exp(|f|))$.
Assume that we can find a canonical form in less than exp time (converse).
Using the above construct a procedure for equivalence as follows:
$f_1 \equiv f_2$ can be checked by

1. Reducing f_1 to its canonical form f_1'.

2. Reducing f_2 to its canonical form f_2'.

3. Comparing f_1' and f_2'.

$\Rightarrow$ The canonical forms cannot be exponential in size, or construction would take exponential time.
$\Rightarrow$ The entire procedure takes less than exp time.
$\Rightarrow$ Equivalence takes less than exp time.
A contradiction.
Hence, finding a canonical form for CTL must be exp time ($\Omega(exp(|f|))$). $\Box$

Thus, we cannot construct a canonical form for CTL in less than exponential time, and hence cannot re-use all previously computed information. We propose to re-use some information by using semi-canonical forms for CTL proposed by Deharbe and Borrione [74]. These consist of a set of reduction rules that attempt to semi-canonicalize the CTL parse tree by the use of local reductions. For example, $EF(EFp)$ may be reduced to the simpler form EFp.

Unfortunately, the order in which reduction rules are applied sometimes makes a difference to the final answer. Consider the following example:

Example 6.1 *Consider the CTL formula $EF(EFp) + EF(h)$, and the reduction rules*

1. *EFp + EFh = EF(p+h)*
2. *EF(EFp) = EFp*

Rule 1 applied first reduces $EF(EFp) + EF(h)$ to $EF(Efp + h)$.
No further reduction is possible.
However Rule 2 applied first followed by Rules 1 reduces $EF(EFp) + EF(h)$ to $EF(p + h)$.
The two forms are not syntactically identical.

This order dependency is not desirable. Rules have to be *confluent.*

Definition 43 Confluent Rules: *A set of reduction rules (for CTL) are confluent if the set gives the same final answer irrespective of the order in which the rules are applied.*

Borrione et al [74] define the following two sets of confluent rules.

Rules 6.2 .

1. $AGp \cdot AGh \equiv AG(p \cdot h)$
2. $EFp + EFh \equiv EF(p + h)$
3. $EG((EGp + h) + (p + EGh)) \equiv EGp + EGh$
4. $AF((AFp \cdot h) + (p \cdot AFh)) \equiv AFp + AFh$
5. $AF((p \cdot p')U((A(p'Uh') \cdot h) + (p' \cdot A(pUh))) \equiv A(pUH) \cdot E(p'Uh')$

Rules 6.3 .

1. $AG(AGp) \equiv AGp$
2. $EF(EFp) \equiv EFp$
3. $EG(EGp) \equiv EGp$
4. $AF(AFp) \equiv AFp$
5. $AG(EGp) \equiv AGp$
6. $EF(AFp) \equiv EFp$
7. $AG(EF(AG(EFp))) \equiv EF(AG(EFp))$
8. $EF(AG(EF(AGp))) \equiv AG(EF(AGp))$
9. $AF(EG(AFp)) \equiv EG(AFp)$
10. $EG(AF(EGp)) \equiv AF(EGp)$
11. $AF(AG(AFp)) \equiv AG(AFp)$
12. $AG(AF(AGp)) \equiv AF(AGp)$
13. $A(pUA(pUh)) \equiv A(pUh)$

Theorem 6.4 *Iterative application of the rewrite rules of Rules 6.2 or 6.3 to a CTL formula always terminates.*

Proof. The number of (temporal) operators on the left side of the rewrite rules are always less than the right side. At each step, the CTL formula must lose at least one operator. There are a finite number of operators in a CTL formula, and hence applying the rewrite rules must converge. □

Theorem 6.5 *The set of Rules 6.2 are confluent.*

Proof. Experimentally shown in [74]. □

Theorem 6.6 *The set of Rules 6.3 are confluent*

Proof. Experimentally shown in [74]. □

Definition 44 Semi-Canonical CTL form: *A CTL form f is said to be semi-canonical if all formulae that are syntactically equivalent to it under the re-write rules 6.2 & 6.3 have the form f after re-write.*

Using the results of Theorem 6.4, 6.5, and 6.6 we propose the following algorithm to semi-canonicalize a CTL formula.

Algorithm 6.3

```
  Apply Rules 6.2
  Apply Rules 6.3
  Normalize (Apply Rules 6.1)
  return
```

To identify the changes between two semi-canonical CTL forms, the algorithms of Chapter 3 may be used. As stated before, the truth or falsity of a CTL formula is also dependent on the FSM model, and hence it is questionable as to what gains may be obtained by comparing two CTL formula that are identical except for the atomic propositions. Thus, it is more relevant to extend the algorithms of [75], which rely on knowing the input (atomic proposition) correspondence.

Substructures in the CTL parse tree that are unchanged require no update from differences in the CTL property, and the previous answers may be re-used directly.

6.4.3 Incremental Model Checking

As previously discussed in Chapter 5, edge subtraction and constraint addition cannot add behavior to the system. Hence, the states involved in valid behavior before edge subtraction and constraint addition are a superset of the final valid set. The incremental analysis for $Fair^+$ proceeds similarly to the analysis in Section 5.4. If states previously marked p (atomic proposition or subformula) are changed to $\overline{p}$ it affects states satisfying any CTL formula dependent on p. We consider this change to be a relabeling of states marked p to $\overline{p}$. Under this relabeling, the new set of states satisfying any CTL formula dependent on p is always a subset of the old set. For example, states that are initially marked $E_C Gp$ are a superset of the new $E_C Gp^{new}$. Edge deletion and constraint addition may also affects the states satisfying $E_C Gp$. To understand this, consider the example of edge deletion in Figure 6.3. As a result of the deletion of the edge shown, some state may no longer be able to reach a state that satisfies p. This state can no longer satisfy $E_C Gp^{new}$. Since, $E_C Gp$ is a GFP computation (Section 6.3), we use the ideas in Theorem 2.4 to create an incremental algorithm. Similarly, since $E_C(pUq)$ is an LFP computation, we use the ideas of Theorem 2.2 to incrementalize it. The computation of states satisfying $E_C Xp, p+q, \overline{p}$ are not fixed points; however, these sets are dependent on $Fair^+$, and some gain can be obtained from an incremental procedure for $Fair^+$ to compute these sets.

Before, we begin our discussion, we define the following operator, which can be used to reduce the set of states satisfying a given CTL $E_C(pUq)$ formula to ensure that remaining elements do indeed satisfy it after changes are made.

Definition 45 CTL Reduce Operator *(RP(T,F,p,q)): Given a transition relation $T(x,y)$, the set of states satisfying $F(x)$, the set of states satisfying $p(x)$, and the set of states satisfying $q(x)$, the reduce operator returns those states in the set $F(x)$ that are either in q(x) or in p(x) with a next state in that set $F(x)$.*

This state may be computed using the following algorithm:

```
$f(Q(x)) = Q(x) \cdot (\exists_y T(y,x) \cdot Q(x) \cdot p(y) + q(x))$
$\mathbf{RP}(T, F, p, q)$
  $\mathbf{return}(GFP(f(Q), F(x)))$
```

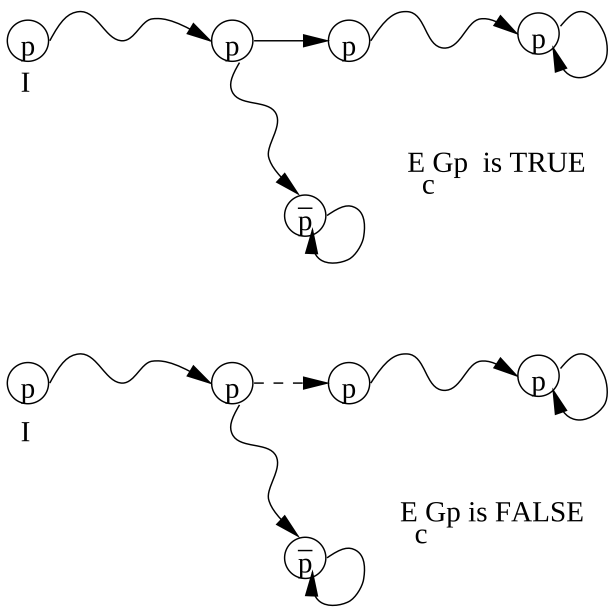


Figure 6.3: $E_C Gp$ changes under edge deletion

We also define a corresponding expand operator that increases the set of states involved in a CTL EGp formula, when changes are made.

Definition 46 CTL Expand Operator *(EO(T,F,p)): Given a transition relation $T(x,y)$, the set of states satisfying $p(x)$, and the set of states satisfying $F(x)$, the expand operator returns those states in $p(x)$ that have a path also exclusively in $p(x)$ to a state in $F(x)$.*

This state may be computed using the following algorithm:

```
$f(Q(x)) = \exists_y T(x,y) \cdot Q(y) \cdot p(x) + Q(x)$
**EO**$(T, F, p)$
  **return**$(GFP(f(Q), F(x) \cdot p(x))$
```

The following lemmas characterize relevant supersets and subsets that may be computed, when a change is made to an instance of model checking.

Lemma 6.7 *If the only changes to the system consist of the subtraction of edges, the addition of constraints, and the relabeling of states marked p to $\overline{p}$, then*

$$Fair^{+new}(x) \quad \subseteq \quad Fair^{+}(x)$$

$$\begin{array}{rcl}
E_C G p^{new}(x) & \subseteq & E_C G p(x) \cdot Fair^{+new}(x) \\
E_C(pUq)^{new}(x) & \supseteq & E_C(pUq)^{--}(x) \\
where \;\; E_C(pUq)^{--}(x) & = & RP(T^{sub}(x,y), E_C(pUq)(x) \cdot Fair^{+new}(x), (p - \triangle p^{sub})(x), q(x)) \\
(p+q)^{new}(x) & = & (p+q)(x) - \triangle p^{sub}(x) \cdot \overline{q}(x) \\
\overline{p}^{new}(x) & = & (\overline{p} + \triangle p^{sub})(x) \cdot Fair^{+new}(x)
\end{array}$$

Proof.

1. $Fair^{+new}(x) \subseteq Fair^{+}(x)$.

   (From Lemma 5.10, as formula changes do not affect the $Fair^{+}$ states).

2. $E_C G p^{new}(x) \subseteq E_C G p(x) \cdot Fair^{+new}(x)$

   $T^{new}(x) = T^{sub}(x) \subseteq T(x)$.

   $C^{new}(x) = C^{add}(x) \supseteq C(x)$.

   $\Rightarrow f^{new} \subseteq f$,

   where the f's refer to the f in the fixed point computation of $E_C G p$ on page 106.

   $\Rightarrow E_C G p^{new}(x) \subseteq E_C G p(x)$ (From Theorem 2.3).

   Also, $E_C G p^{new}(x) \subseteq Fair^{+new}(x)$ (by definition) .

   $\Rightarrow E_C G p^{+}(x) \subseteq E_C G p(x) \cdot Fair^{+}(x)$

3. $E_C(pUq)^{new}(x) \supseteq E_C(pUq)^{--}(x)$.

   - $E_C(pUq)^{new}(x) \subseteq Fair^{+}(x)$ (by definition).
   - $E_C(pUq)^{new}(x) \supseteq E_C(pUq)^{--}(x)$.

     Assume not.

     $\Rightarrow \exists s \in E_C(pUq)^{--}(x), s \not\in E_C(pUq)^{new}(x)$.

     $E_C(pUq)^{new}(s) = 0$.

     $\Rightarrow$

     - Either $q(s) = 0$ and $p(s) = 0$.

       $\Rightarrow E_C(pUq)^{--}(s) = 0$.

       A contradiction by the construction of the RP operator on page 116.
     - Or $q(s) = 0$ and for all next states s', $E_C(pUq)^{--}(s') = 0$.

       A contradiction by the construction of the RP operator on page 116.

$\Rightarrow E_C(pUq)^{new}(x) \supseteq E_C(pUq)^{--}(x)$.

4. $(p+q)^{new}(x) = (p+q)(x) - \triangle p^{sub}(x) \cdot \overline{q}(x)$ (by definition)

5. $\overline{p}^{new}(x) = (\overline{p}(x) + \triangle p^{sub}(x)) \cdot Fair^{+new}(x)$ (by definition)

□

If edges are added, constraints are subtracted, or states are relabeled from $\overline{p}$ to p then the arguments are reversed. As shown in Section 5.4, the new set $Fair^{+new}$ is now a superset of the old $Fair^+$. Similarly, the new set of states satisfying any CTL formulae dependent on p is a superset of the old set. The following lemma summarizes the relationship between old and new state sets under this class of changes.

Lemma 6.8 *If the only changes to the system consist of the addition of edges, the subtraction of constraints, and the relabeling of states marked $\overline{p}$ to p, then*

$$\begin{aligned}
Fair^{+new}(x) &\subseteq Fair^+(x) + RRS(T^{add}(x,y), \Lambda(\triangle T^{add}(x,y),x)) + \overline{\triangle C^{sub}}(x) \\
E_C G p^{new}(x) &\subseteq E_C G p^{++}(x) \\
\textit{where}\ \ E_C G p^{++}(x) &= E_C G p(x) + EO(T^{add}(x,y), (\Lambda(\triangle T^{add}(x,y),x)), (\triangle p^{add} + p)(x)) + \\
&\quad (\triangle p^{add} + p)(x) \cdot \overline{\triangle C^{sub}(x)}. \\
E_C(pUq)^{new}(x) &\supseteq E_C(pUq)(x). \\
(p+q)^{new}(x) &= (p+q)(x) + \triangle p^{add}(x). \\
\overline{p}^{new}(x) &= \overline{p}(x) - \triangle p^{add}(x).
\end{aligned}$$

Proof.

1. $Fair^{+new}(x) \subseteq Fair^+(x) + RRS(T^{new}(x,y), \Lambda(\triangle T^{add}(x,y),x)) + \overline{\triangle C^{sub}}(x)$
   (From Lemma 5.13, as formula changes do not affect the $Fair^+$ states).

2. $E_C G p^{new}(x) \subseteq E_C G p^{++}(x)$

   - Adding states, subtracting constraints and relabeling states marked $\overline{p}$ to p cannot remove states marked $E_C GP$, hence $E_C G p^{new}(x) \supseteq E_C G p(x)$.
   - Consider $s \in E_C G p^{new}(x)$.

- Either s is either in new p^{add} states that may be fair.

  $\Rightarrow (\triangle p^{add} + p)(s) \cdot \overline{\triangle C^{sub}}(s) = 1.$

  $\Rightarrow s \in E_C G p^{++}(x).$
- Or s in new behavior created by $\triangle T^{add}$ or C^{sub}, which also satisfies p.

  By the definition of EO operator on page 117.

  $\Rightarrow s \in E_C G p^{++}(x).$

3. $E_C(pUq)^{new}(x) \supseteq E_C(pUq)(x).$
   $T^{new}(x) = T^{add}(x) \supseteq T(x).$
   $C^{new}(x) = C^{sub}(x) \subseteq C(x).$
   $\Rightarrow f^{new} \supseteq f,$
   where the f's refer to the f in the fixed point computation of $E_C(pUq)$ on page 107.
   $E_C(pUq)^{new}(x) \supseteq E_C(pUq)(x).$ (Theorem 2.1)

4. $(p+q)^{new}(x) = (p+q)(x) + \triangle p^{add}(x)$
   (by definition)

5. $\overline{p}^{new}(x) = \overline{p}(x) - \triangle p^{add}(x)$
   (by definition)

□

6.4.4 General Changes

We can combine Lemmas 6.7, 6.8 to get the following theorem that characterizes the relevant supersets and subsets to an instance of model checking with changes.

Theorem 6.9 *For a general change to an instance of model checking:*

$$\begin{aligned}
Fair^{+new}(x) &\subseteq Fair^{+}(x) + RRS(T^{new}(x,y)\Lambda(\triangle T^{add}(x,y),x)) + \overline{\triangle C^{sub}}(x) \\
E_C G p^{new}(x) &\subseteq E_C G p^{++}(x) \\
\textit{where} \quad E_C G p^{++}(x) &= E_C G p(x) + EO(T^{new}(x,y),(\Lambda(\triangle T^{add}(x,y),x)),(\triangle p^{add} + p - \triangle p^{sub})(x)) + \\
&\quad (\triangle p^{add} + p - \triangle p^{sub})(x) \cdot \overline{\triangle C^{sub}}(x) \\
E_C(pUq)^{new}(x) &\supseteq E_C(pUq)^{--}(x)
\end{aligned}$$

$$\begin{aligned} \textit{where}\quad E_C(pUq)^{--}(x) &= RP(T^{new}(x,y), E_C(pUq)(x)\cdot Fair^{+new}(x), (\triangle p^{add} + p - \triangle p^{sub}), q) \\ (p+q)^{new}(x) &= (p+q)(x) - \triangle p^{sub}(x)\cdot \overline{q}(x) + \triangle p^{add}(x) \\ \overline{p}^{new}(x) &= \overline{p}(x) + \triangle p^{sub}(x) - \triangle p^{add}(x) \end{aligned}$$

Proof. From Lemmas 6.7 and 6.8. □

Using the above Theorem, we propose the following incremental algorithm

Algorithm 6.4 *Incremental_Model_Check*$(f, Fair^{+new}(x))$

```
 Case:
 f is an atomic proposition
       return $f(x)\cdot Fair^{+new}(x)$
 $f \equiv EGp$
   $p^{new} = (\triangle p^{add} + p - \triangle p^{sub})$
   $p^{new}(x) = (\triangle p^{add} + p - \triangle p^{sub})(x)$
         $= Incremental\_Model\_Check(p^{new}, Fair^{+new}(x))$
   $f(Q(x)) = FSS(T^{new}(x,y), FFP(T^{new}(x,y), C^{new}(x), Q(x)))$
   $E_CGp^{new}(x) = GFP(f(Q), E_CGp^{++}(x))$
   where   $E_CGp^{++}(x) = E_CGp(x) + EO(T^{new}(x,y), (\Lambda(\triangle T^{add}(x,y),x)), (\triangle p^{add} + p - \triangle p^{sub})(x))+$
           $(\triangle p^{add} + p(x) - \triangle p^{sub})\cdot \overline{\triangle C^{sub}}$
       Return $E_CGp^{new}(x)$
 $f \equiv E(pUq)$
   $p^{new} = (\triangle p^{add} + p - \triangle p^{sub})$
   $p^{new}(x) = (\triangle p^{add} + p - \triangle p^{sub})(x)$
         $= Incremental\_Model\_Check(p^{new}, Fair^{+new}(x))$
   $q(x) = Incremental\_Model\_Check(q)$
   $f(Q(x)) = (\triangle p^{add} + p - \triangle p^{sub})(x) + q(x)\cdot EX(Q(x))$
   $E_C(pUq)^{new} = LFP(f(Q), E_C(pUq)^{--}(x))$
   where   $E_C(pUq)^{--}(x) = RP(T^{new}(x,y), E_C(pUq)(x)\cdot Fair^{+new}(x), (\triangle p^{add} p - \triangle p^{sub})(x), q(x))$.
       Return $E_C(pUq)^{new}$
 $f \equiv EXp$
   $p^{new} = (\triangle p^{add} + p - \triangle p^{sub})$
   $p^{new}(x) = (\triangle p^{add} + p - \triangle p^{sub})(x)$
```

```
            $= \mathit{Incremental\_Model\_Check}(p^{new}, Fair^{+new}(x))$
        $E_C X p^{new}(x) = Fair^{+new}(x) \cdot \exists_y T^{new}(x,y) \cdot (\triangle p^{add} + p - \triangle p^{sub})(y)$
            **Return** $E_C X p^{new}(x)$
    $f \equiv p + q$
        $p^{new} = (\triangle p^{add} + p - \triangle p^{sub})$
        $p^{new}(x) = (\triangle p^{add} + p - \triangle p^{sub})(x)$
            $= \mathit{Incremental\_Model\_Check}(p^{new}, Fair^{+new}(x))$
        $q(x) = \mathit{Incremental\_Model\_Check}(q)$
        $(p+q)_c^{new}(x) = (p+q)(x) - \triangle p^{sub}(x) \cdot \overline{q}(x) + \triangle p^{add}(x)$
            **Return** $(p+q)_c^{new}(x)$
    $f \equiv \overline{p}$
        $\overline{p^{new}}(x) = (\overline{p} + \triangle p^{sub} - \triangle p^{add})(x) \cdot Fair^{+new}$
            **Return** $\overline{p^{new}}(x)$
```

Theorem 6.10 *Algorithm Incremental_Model_Check is correct.*

Proof. From Theorem 6.9. □

As a final comment, it is necessary to note that the proposed subsets are not necessarily the best.

6.5 Conclusions

We have extended the arguments of Chapter 5 to model checking, and shown how to exploit the fixed point nature of CTL model checking to get incremental algorithms. This analysis may be very useful when checking many different CTL formulae. The implementation of this proposed method remains to be done. We assume that the results of CTL model checking (as well as subformulae) are cached. In the methodology we propose, whenever a new formula is read in, it is transformed into the semi-canonical form described on page 115. The parse tree for this semi-canonical form is compared against existing CTL formula by using the common substructure techniques of Chapter 6. This comparison helps to find commonalities that may be re-used. We identify sub-formulae in the new CTL formula that have already been computed, and CTL model checking proceeds using these subformulae as starting points. If additional changes are made to the FSM itself, the methods described if Chapter 4 can be used to detect these changes, and compute the resulting T^{add}, T^{sub} etc relations. This change information can also be supplied to Algorithm 6.4 to incrementally re-compute CTL model checking.

Chapter 7

Incremental Synthesis

7.1 Introduction

Logic synthesis refers to the process of optimizing a logic description of a circuit, given as a net-list of logic (Boolean) gates [57]. This representation can be optimized for area(minimum), delay (minimum or meeting requirements), and power(minimum). Since these problems are hard to solve exactly, heuristic algorithms are generally used. However, these algorithms are unstable; if a small change is made in the network function, the output of the synthesis algorithm may vary greatly from the previous implementation. A designer can invest effort in optimizing the original design by hand, so it is desirable that most of the hand-designed or optimal parts be preserved, even when changes are made to the specification. In addition, the network may have already been implemented in silicon at a lower level of the design hierarchy, and it can be inconvenient to change.

Previous algorithms for the problem of incremental synthesis have dealt with post-rectification (Watanabe et al [34]), and preserving cones of logic (Brand et al [76]) in the design. Some relevant work has also been done by Kukimoto and Fujita [77] but this is concerned with FPGA's rather than general logic. In addition, this work restricted re-synthesizable parts of the network to all nodes at a level, rather than a general re-synthesis region. Other approaches to this problem, which use Boolean unification were proposed by Fujita et al [36], and Lin et al [38], however these approaches do not consider the optimality of sub-regions in the network as a factor in choosing candidate regions for re-synthesis. None of the above approaches have dealt directly with preserving the "highly-optimal" parts of the circuit.

Changes to the system are defined as changes in the functions computed at the primary output nodes. The problem is stated as: We are given a logic design that has inputs $x_i \in x$, outputs $z_i \in z$, and an implementation I of $z = F(x)$. I has already been optimized for an objective, which may be power, delay or area. Now, due to engineering change, a new specification $z = F_{\text{new}}(x)$ is given. We want to find some sub-region R of the implementation I that alone may be re-synthesized so that I with R re-synthesized, implements the new function F_{new}.

We propose and experiment with two solutions to this problem; an exact algorithm and a heuristic one. The exact algorithm implicitly enumerates all regions R and chooses the best one. Unfortunately, this can only handle very small circuits. The heuristic proposes an iterative solution to the problem; we begin with a small region for re-synthesis (selected using some criteria), and iteratively expand that region until a solution is obtained. At each stage, we test if re-synthesizing this region alone can realize the new specification. As a second pass, we trim the region iteratively, so that it becomes minimal in the sense that no subset of the current region can realize the change in functionality.

However, not all minimal regions are equivalent in terms of their power, area or delay optimality. To compare two different minimal re-synthesis regions, we use a heuristic evaluation criteria for the acceptability of regions for re-synthesis called sensitivity. In this chapter we compute the sensitivity (or acceptability for re-synthesis) for power. In this respect, we rely heavily on the work done by Lennard [78] for the computation of power sensitivities of nodes. This sensitivity criteria is used to pick nodes in the iterative scheme.

We assume that F and F_{new} are completely specified; the extensions to incomplete specifications are straightforward. The designer can designate which regions may not be re-synthesized, or order the regions in terms of where re-synthesis is more acceptable. We attempt to find the minimal re-synthesis region.

This chapter is organized as follows: Section 7.2 describes the terminology and definitions in this chapter. In Section 7.3 we give a procedure for determining whether re-synthesis of a given a sub-region of the network can realize an implementation with the new functionality

$z = F_{\text{new}}(x)$. In Section 7.4.2, we formulate the exact algorithm using the results of the previous section. Next, we briefly discuss the need for estimates of the goodness of a node for re-synthesis (or sensitivity). The next section (7.4) discusses the heuristic algorithm and its motivation. This is an iterative algorithm for incremental synthesis that begins with an empty re-synthesis region, and iteratively picks nodes from the rest of the network to add to the region (in order of their sensitivity). We present some experimental results in Section 7.5.

7.2 Terminology for Synthesis

Recall that a **completely specified Boolean function** F with n inputs and m outputs is a mapping $F : B^n \longrightarrow B^m$, where $B = \{0, 1\}$. If $m = 1$ the **onset** and **offset** are the set of points satisfying $F(x) = 1$ and $F(x) = 0$ respectively. A **minterm** v of a function F is a vertex (i.e. a point in B^n) such that $F(v) = 1$.

The **size** of $F(x)$, $(|F(x)|)$ denotes the number of minterms (onset points) in $F(x)$. An **incompletely specified** Boolean function is a mapping $F : B^n \longrightarrow Y^m$ where $Y = \{0, 1, \star\}$ ($\star \Rightarrow F$ can be 0 or 1). If $m = 1$ the **onset**, **offset**, and **don't care set (dcset)** are the set of points such that $F(x) = 1$, $F(x) = 0$, and $F(x) = \star$ respectively.

We're interested in Boolean functions, because the optimization of digital circuits relies heavily of Boolean functions and their manipulation. In this context, we define a network or circuit as:

A **Boolean network** (Figure 7.1) $\mathcal{N}$, is a directed acyclic graph (DAG) such that each node in $\mathcal{N}$ has a Boolean function $(n = f_n(n_1 \ldots n_m))$. There is a directed edge from node n_i to node n if the function f_n is dependent on node n_i, node n_i is a **fanin** of a node n, and node n is a **fanout** of node n_i. A node n_i is a **transitive fanin** of a node n if there is a directed path from n_i to n, and n is called a **transitive fanout** of n_i. The inputs $\mathbf{x} = (x_1, \ldots, x_n)$ of the Boolean network are called **primary inputs** and outputs $\mathbf{z} = (z_1, \ldots, z_m)$ are called **primary outputs**. Nodes with at least one fanin and one fanout are called **internal**.

A more general form of representing Boolean relationships is an observability relation.

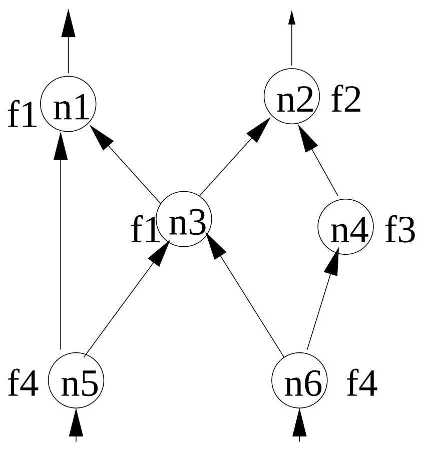


Figure 7.1: Network

Definition 47 *An* **Observability Relation** *is a mapping* $O^F(x, z) : B^n \times B^m \longrightarrow B$, *where* x *are inputs and* z *outputs. Given any function of the network with inputs* $x = (x_1, \ldots, x_n)$ *and outputs* $z = (z_1, \ldots, z_m)$*,* $(z = F(x))$ *may also be represented as its* **observability relation** $O^F(x, z) := z\overline{\oplus}F(x)$*. Given inputs* x *and outputs* z*, an observability relation is characterized as* $O^F(x, z) = 1$ *if* $z = F(x)$ *and* $O^F(x, z) = 0$ *if and only if* $z \neq F(x)$*.*

Given a boolean function $S(n)$, the *projection* $(\Omega_I(S(n)))$ is the representation of set $S(n)$ in terms of the variables of some input set I. In the context of nodes in a network, if a node n has a function $n = G(I)$ associated with it, and I denotes the set of inputs, then $\Omega_I(S(n)) = G^{-1}(S(n)))$.

Definition 48 Composition of Functions*: Given two Boolean observability relations* $F_1(x, u)$ *and* $F_2(u, z)$*, the composition of the two functions* $F_1(x, u) \odot |_u F_2(u, z) = \exists_u F_1(x, u) \cdot F_2(u, z)$*. Given two networks, with Boolean observability relations* $F_1(x, u)$ *and* $F_2(u, z)$*, their composition has the observability relation* $F_1(x, u) \odot |_u F_2(u, z) = \exists_u F_1(x, u) \cdot F_2(u, z)$*. The composition network can be obtained by connecting the two as shown in Figure 7.2. Note that one can compose the corresponding functions* $z = f_2(u), u = f_1(x)$ *as* $z = f_2(f_1(x))$*.*

A relation $F_1(x, u)$ *is said to be* compatible *with a relation* $F_2(u, z)$ *iff* $F_1(x, u) \odot_u F_2(u, z) \neq \phi$*.* $F_1(x, u)$ *is* compatible *with a relation* $F_2(u, z)$ *and* consistent *with* $F(z, z)$ *iff* $F_1(x, u) \odot |_u F_2(u, z) = F(x, z)$*.*

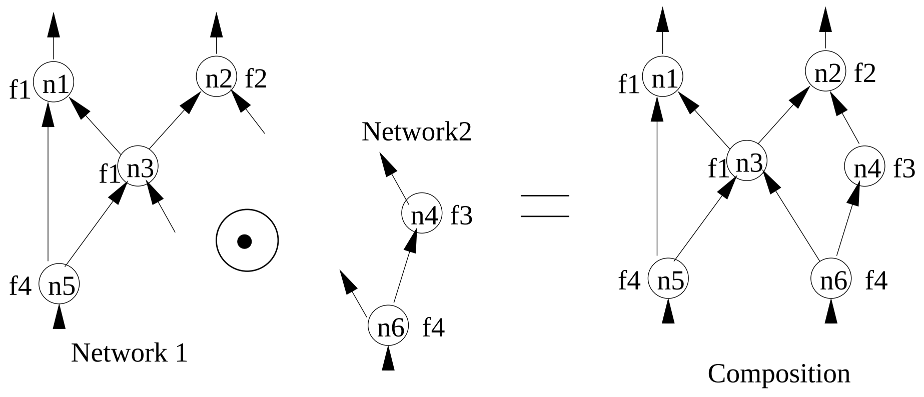


Figure 7.2: Composition

Apart from the usual logic connectives, we will also be using the following operands in first order propositional logic.

Definition 49 *The* **cofactor** F_{x_i} *of* F *(completely specified) with respect to variable* x_i *is the function* F *evaluated at* $x_i = 1$*.*

Definition 50 **Consensus or universal quantification** $\forall$ *is defined as* $\forall_{x_i} f(x_1, \ldots x_n) = f_{x_i} \cdot f_{\overline{x_i}}$*. It is the largest Boolean function contained in* f *that is independent of* x_i*. The consensus satisfies the following properties:*

$$\begin{aligned} \forall_x f &\subseteq f \\ \forall_x (f+g) &\supseteq \forall_x f + \forall_x g \\ \forall_x f \cdot g &= \forall_x f \cdot \forall_x g \end{aligned}$$

Definition 51 **Smoothing or existential quantification** $\exists$ *is defined as* $\exists_{x_i} f(x_1, \ldots, x_n) = f_{x_i} + f_{\overline{x_i}}$*. It is the smallest Boolean function containing* f *that is independent of* x_i*. The smoothing satisfies the following properties:*

$$\begin{aligned} \exists_x f &\supseteq f \\ \exists_x (f+g) &= \exists_x f + \exists_x g \\ \exists_x f \cdot g &\subseteq \exists_x f \cdot \exists_x g \end{aligned}$$

One Boolean function of interest is the Boolean difference:

Definition 52 *The* **Boolean difference** *of a function f with respect to a variable x is defined as $\frac{\partial f}{\partial x} \equiv f_x \overline{f}_{\overline{x}} + \overline{f}_x f_{\overline{x}}$. This function gives all the conditions under which the value of f is* **influenced** *by the value of x. Its complement consist of the conditions under which f is* **insensitive** *to x.*

7.3 Conditions on a Valid Re-synthesis Region

As stated before, we have a logic function (possibly multi-output) $z = F(x)$ $(z = (z_1 \dots z_m))$, which is a function of inputs $x = (x_1 \dots x_n)$. We already have a network realization for this function. The logic function of this network is changed to a new function $z = F_{\text{new}}(x)$. We may also represent the new function as a separate network. The objective is to realize $F_{\text{new}}(x)$, while simultaneously preserving as much of the old network structure I (particularly hand-optimized portions) as possible.

We recognize the following sub-problem: Given a network with original functionality $z = F(x)$, which is to be changed to $z = F_{\text{new}}(x)$, and a region for re-synthesis R (see Figure 7.3) with inputs v and outputs u, determine whether the new function can be implemented by re-synthesizing the region R exclusively.

To answer this question, we first compute an observability relation for the region R, that is consistent with the overall implementation $z = F_{\text{new}}(x)$ and compatible with the implementations for the remainder of the original network. Next we impose conditions on this relation that ensure that it is implementable.

The overall observability relation for the original circuit is characterized by $O^F(x, z) = z \overline{\oplus} F(x)$, and the new relation is characterized by $O^F_{\text{new}}(x, z) = z \overline{\oplus} F_{\text{new}}(x)$. The region of re-synthesis R, with inputs v and outputs u, is characterized in its current implementation, by an observability relation $O^R(v, u)$ that is consistent with $O^F(x, z)$ and compatible with the remainder of the network. The remainder of the network is characterized by the network N (with the region R deleted), with inputs u and x and outputs z and v. Its characteristic function is given by $N(x, v, u, z)$. Figure 7.3 illustrates these regions, and their inputs and outputs. In this section we illustrate how to compute the required functionality of a

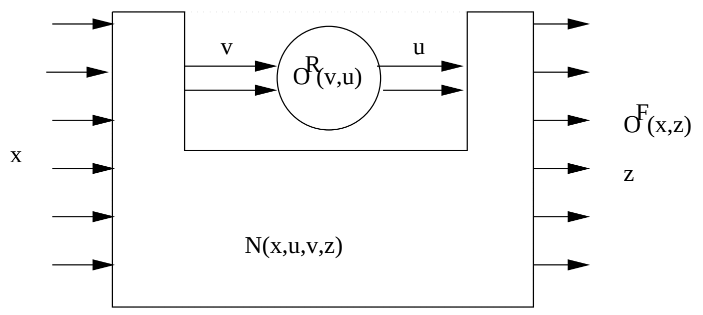


Figure 7.3: Re-synthesis region

predefined region when the output function is changed. We first state the following theorem, which is adapted from [79].

Theorem 7.1 *The maximum observability relation for region R that is consistent with $N(x,v,u,z)$ and compatible with $O^F(x,z)$ is:*
$O^R(v,u) = \forall_{x,z}(N(x,v,u,z) \Rightarrow O^F(x,z))$.

Proof. We need to prove that O^R composed with N gives F, i.e. $O^R(v,u) \odot |_{u,v} N(x,v,u,z) \equiv O^F(x,z)$.

The network N puts no restrictions between sets of allowable u, v, i.e. v is not dependent on u or vice versa. However, for F to be produced by the final network additional restrictions must be placed on u, v's. Thus,
$\exists_{u,v} N(x,v,u,z) \supseteq O^F(x,z)$.

- $O^R(v,u) \odot |_{u,v} N(x,v,u,z) \subseteq O^F(x,z)$.
  Assume not,
  $\Rightarrow \exists_{x',z'}$ such that $O^F(x',z') = 1$ and $O^R(v,u) \odot |_{u,v} N(x',v,u,z') = 0$
  $\Rightarrow \exists_{u,v}(\forall_{x,z}\overline{N(x,u,v,z)} + O^F(x,z)) \cdot N(x',u,v,z') = 0$.
  - Either $N(x',v,u,z') = 0$. (use $\exists_x f \supseteq f$)
    Substitute in $\Rightarrow \exists_{u,v}(\forall_{x,z}\overline{N(x,u,v,z)} + O^F(x,z)) \cdot N(x',u,v,z')$.
    $\Rightarrow 0 = \exists_{u,v} N(x,v,u,z) \supseteq O^F(x,z)$.
    $\Rightarrow O^F(x,z) = 0$.
    A contradiction.

- Or $\forall_{x,z}\overline{N(x,u,v,z)} + O^F(x,z) = 0$
  for $(x',z'), \overline{N(x',u,v,z')} + O^F(x',z') = 0$
  We know from the previous part that $N(x',v,u,z') \not= 0$ and $O^F(x',z') \not= 0$.
  A contradiction.

- $O^R(v,u) \odot |_{u,v} N(x,v,u,z) \supseteq O^F(x,z)$.
  Assume not.
  $\Rightarrow \exists_{x',z'}$ such that $O^F(x',z') = 0$ and $O^R(v,u) \odot |_{u,v} N(x',v,u,z') = 1$
  $\Rightarrow \exists_{u,v}(\forall_{x,z}\overline{N(x,u,v,z)} + O^F(x,z)) \cdot N(x',u,v,z') = 1.$
  $\Rightarrow \forall_{u,v}(\exists_{x,z}N(x,u,v,z) \cdot \overline{O^F(x,z)}) + \overline{N(x',u,v,z')} = 0.$
  $\Rightarrow \overline{N(x',u,v,z')} = 0$ and $(\exists_{x,z}N(x,u,v,z) \cdot \overline{O^F(x,z)}) = 0$. (use $\exists_x f \supseteq f$)
  $\Rightarrow \overline{O^F(x',z')} = 0$
  $\Rightarrow O^F(x',z') = 1$
  A contradiction.

Hence, $O^R(v,u) \odot |_{u,v} N(x,v,u,z) = O^F(x,z)$. □

When F has been changed to F_{new}, we can simply replace F by F_{new} in the above and give a condition for realizability of the new functionality.

Theorem 7.2 *Let $O^R{}_{new}(v,u) = \forall_{x,z}(N(x,v,u,z) \Rightarrow O^F_{new}(x,z))$. The new functionality can be realized by re-synthesizing R iff $(\forall_v \exists_u O^R{}_{new}(v,u) = 1)$,*

Proof. From Theorem 7.1 $O^R_{\text{new}}(v,u) \odot |_{u,v} N(x,v,u,z) \equiv O^F_{\text{new}}(x,z)$.
The relation has be completely specified for a valid behavior to exist within it.
i.e. $\forall_v \exists_u O^R{}_{\text{new}}(v,u) = 1$. □

A relation that satisfies Theorem 7.2 cannot directly yield a hardware realization, since hardware can only implement functions. In general, $O^R{}_{\text{new}}(v,u)$ is not a function. However, any $O^R{}_{\text{new}}(v,u)$ satisfying Theorem 7.2 has at least one function as a subset. To find such a function, we have to solve a set of Boolean equations. The following theorem from Boolean unification [80], details all solutions $y = G(x)$ to the equation $f(x,y) = 1$, when y is a single variable.

Lemma 7.3 *If $f(x,y) = 1$, where y is a single variable, $\overline{f(x,0)} \subseteq f(x,1)$.*

Proof. $f(x,y) = 1$.
$\Rightarrow y \cdot f(x,1) + \overline{y} \cdot f(x,0) = 1$.
$f(x,1) \supseteq y \cdot f(x,1)$ and
$f(x,0) \supseteq \overline{y} \cdot f(x,0)$.
$\Rightarrow f(x,1) + f(x,0) = 1$.
$\Rightarrow (\overline{f(x,0)} \Rightarrow f(x,1))$.
$\Rightarrow \overline{f(x,0)} \subseteq f(x,1)$. $\square$

Theorem 7.4 *The solutions $y = G(x)$ to an equation $f(x,y) = 1$, where y is a single variable, can be characterized by the inequalities $\overline{f(x,0)} \subseteq G(x) \subseteq f(x,1)$.*

Proof. From Lemma 7.3 $\overline{f(x,0)} \subseteq f(x,1)$.

- $y = g(x) = f(x,1)$ is the largest function such that $f(x,y) = 1$.
  Assume not; $\exists y = U(x)$ such that $U(x) \supset f(x,1)$ and $f(x,U(x)) = 1$.
  Since $U(x) \supset f(x,1)$, $\exists_s, U(s) = 1$ and $f(s,1) = 0$.
  But then, $1 = f(s,U(s)) = f(s,1) = 0$.
  A contradiction, hence $y = g(x) = f(x,1)$ is the largest function st $f(x,y) = 1$.

- $y = g(x) = \overline{f(x,0)}$ is the smallest function st $f(x,y) = 1$.
  Assume not; $\exists y = L(x)$ such that $L(x) \subset \overline{f(x,0)}$ and $f(x,L(x)) = 1$.
  Since $L(x) \subset \overline{f(x,0)}$, $\exists_s, L(s) = 0$ and $f(s,0) = 0$.
  But then, $1 = f(s,L(s)) = f(s,0) = 0$.
  A contradiction, hence $y = g(x) = \overline{f(x,0)}$ is the smallest function st $f(x,y) = 1$.

Hence, $\overline{f(x,0)} \subseteq G(x) \subseteq f(x,1)$ for any $G(x)$ satisfying $f(x,G(x)) = 1$. $\square$ The following theorem, adapted from [80], characterizes a family of functions that yield a valid implementation of the relation. We assume that $O^R{}_{\text{new}}(v,u)$ satisfies Theorem 7.2. Note that Theorem 7.5 is a generalization of Theorem 7.4 for multiple output variables.

Theorem 7.5 *If $O^R{}_{new}(v,u)$ is the observability relation for a region R, then any function $u = g(v) = g_1(v) \dots g_m(v)$, $(u = u_1 \dots u_m)$ that satisfies $f_i(v) \leq g_i(v) \leq (f_i + d_i)(v)$, where*

$f_i + d_i = \overline{r_i} = ((\exists_{\ldots u_j \ldots_{(j \neq i)}} O^R new(v,u)) \cdot \prod_{k<i}(u_k \overline{\oplus} g_k(v)))|_{(u_i=1)}$

$r_i + d_i = \overline{f_i} = ((\exists_{\ldots u_j \ldots_{(j \neq i)}} O^R new(v,u)) \cdot \prod_{k<i}(u_k \overline{\oplus} g_k(v)))|_{(u_i=0)}$

satisfies the relation $O^R new(v, g(v)) = 1$. At every stage f_i represents the onset, d_i the don't care set, and r_i offset from which g_i is chosen.

Proof.

- If $m = 1$, this is true from Theorem 7.4.

- Assume for $m = n$ the theorem holds.
  Consider $m = n + 1$.
  $O^R(v, u_1, \ldots u_n, u_{n+1})$ is a function with $m = n + 1$ outputs.
  $\exists_{u_{n+1}} O^R(v, u_1 \ldots u_n, u_{n+1})$ is a function with $m = n$ outputs.
  Hence, any function $u = g(v) = g_1(v) \ldots g_n(v)$, $(u = u_1 \ldots u_n)$ that satisfies
  $f_i(v) \leq g_i(v) \leq (f_i + d_i)(v)$, where
  $f_i + d_i = \overline{r_i} = ((\exists_{\ldots u_j \ldots_{(j \neq i, n+1)}} \exists_{u_{n+1}} O^R \text{new}(v,u)) \cdot \prod_{k<i}(u_k \overline{\oplus} g_k(v)))|_{(u_i=1)}$
  $\Rightarrow$
  $f_i + d_i = \overline{r_i} = ((\exists_{\ldots u_j \ldots_{(j \neq i)}} O^R \text{new}(v,u)) \cdot \prod_{k<i}(u_k \overline{\oplus} g_k(v)))|_{(u_i=1)}$
  And
  $r_i + d_i = \overline{f_i} = ((\exists_{\ldots u_j \ldots_{(j \neq i, n+1)}} \exists_{u_{n+1}} O^R \text{new}(v,u)) \cdot \prod_{k<i}(u_k \overline{\oplus} g_k(v)))|_{(u_i=0)}$
  $\Rightarrow$
  $r_i + d_i = \overline{f_i} = ((\exists_{\ldots u_j \ldots_{(j \neq i)}} \exists_{u_{n+1}} O^R \text{new}(v,u)) \cdot \prod_{k<i}(u_k \overline{\oplus} g_k(v)))|_{(u_i=0)}$
  is a valid solution for the first n outputs.

- Note that $\exists_{u_{n+1}} O^R(v, u_1 \ldots u_n, u_{n+1})$ is independent of u_{n+1}. For $m = n+1$, the solutions $g_1 \ldots g_n$ are already consistent with any solution g_{n+1} as they are independent of u_{n+1}.

- We can compute a solution for the $n + 1$th output, $u_{n+1} = g_{n+1}(v)$ as follows:
  Any function that is

  - Consistent with $u_1 \ldots u_n = g_1(v) \ldots g_n(v)$, i.e.
    $\forall_{i \leq n+1} u_i \equiv g_i(v) \Rightarrow u_i \overline{\oplus} g_i(v) = 1$ and

– Satisfies the relation, i.e.$O^R(v, u_1, \dots u_n, g_{n+1}(v)) = 1$

is a valid solution.

- Hence, one such set of functions is: $f_{n+1}(v) \leq g_{n+1}(v) \leq (f_{n+1} + d_{n+1})(v)$, where

$f_{n+1} + d_{n+1} = \overline{r_{n+1}} = ((\exists_{\dots u_j \dots_{(j \neq i)}} O^R\text{new}(v, u)) \cdot \prod_{k<i}(u_k \overline{\oplus} g_k(v)))|_{(u_{n+1}=1)}$

$r_{n+1} + d_{n+1} = \overline{f_{n+1}} = ((\exists_{\dots u_j \dots_{(j \neq i)}} O^R\text{new}(v, u)) \cdot \prod_{k<i}(u_k \overline{\oplus} g_k(v)))|_{(u_{n+1}=0)}$

□

We can use the freedom provided by the d_i to optimize the function. Reordering the u_i gives rise to different functions; in general a good ordering should be found.

7.4 Minimum Re-Synthesis

We examine two algorithms for minimal re-synthesis; an exact one, which is guaranteed to find the exact minimum region, and a heuristic one that attempts to find a small region for re-synthesis. In both these strategies, we use the method for determining whether a region is sufficient to realize the new functionality (Section 7.3). Before we give the details of the iterative algorithm, we impose certain restrictions on the structure of valid regions.

We require a loose form of structural contiguity restriction on R. In particular, we do not allow the inputs of R to be dependent on outputs from R. This structural restriction is needed to use Theorem 7.2. For instance the left hand side region of Figure 7.4, R, which is composed of two non-contiguous regions, is not a valid region, since in network N_1 input A depends upon C. The right hand side region is valid. This restriction is imposed, since re-synthesis of such a region R might possibly create a combinational cycle within the network, by synthesizing C to be depend on A.

7.4.1 Searching for Minimal Regions

Definition 53 *A re-synthesis region is minimal if every node that can be removed from R while preserving structural continuity, destroys the ability, by re-synthesis of R, to obtain the new functionality F_{new}.*

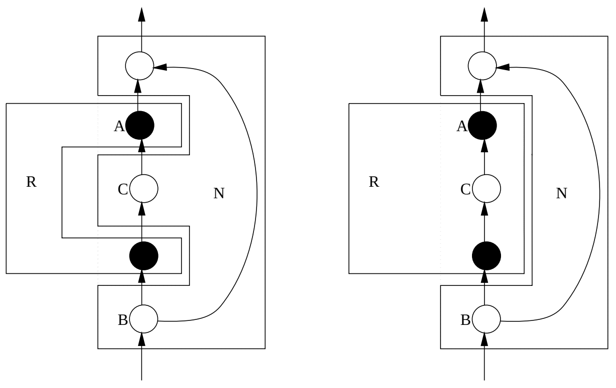


Figure 7.4: Invalid and valid regions

It is easy to see that, if a region R is sufficient for re-synthesis (Theorem 7.2), then every superset $R \subseteq R'$ is also sufficient for re-synthesis. Consequently, if we greedily remove nodes from a feasible contiguous region, while maintaining feasibility and structural validity, our search is guaranteed to terminate in a locally minimal region. However, finding a global minimum is much harder; We examine a strategy that implicitly enumerates all feasible sub-regions and is guaranteed to find a global minimum.

7.4.2 Exact Minimum Re-synthesis

The strategy for getting the exact minimum re-synthesis region consists of implicitly enumerating all regions, and examining this set to find the minimum solution.

7.4.2.1 Implicit Enumeration of Regions

In order to enumerate all regions implicitly, we introduce one variable r_i for each element in the region R of interest and each node that fans into the region. This set of nodes will be called the *extended region* E. Figure 7.5 shows an example with six nodes in the extended region, four of which are in the region. This example will be used to illustrate all the concepts introduced in this section. We introduce six variables $r_1, r_2, ..., r_6$. We will use $\vec{r}$ to denote the vector of $r_i's$. A sub-region of E can be represented as an assignment to $\vec{r}$, with the usual interpretation that a node is in the region iff its corresponding r_i variable is 1. In the example, $(1,1,1,0,0,0)$ denotes the region $\{r_1, r_2, r_3\}$. A function of $\vec{r}$, represents

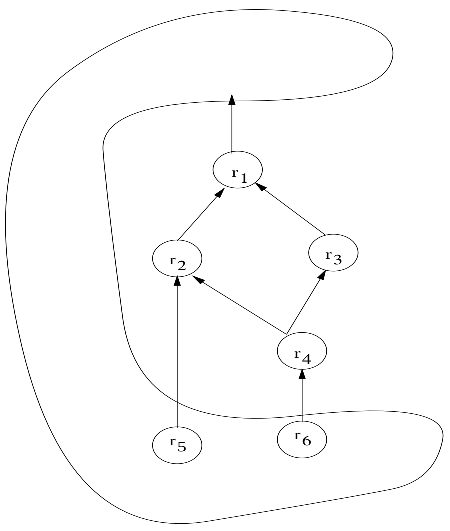


Figure 7.5: Region with variables r_i for each node in the extended region

a set of sub-regions. In our example, $f(\vec{r}) = r_1 r_2 + r_3 r_4$ represents the set of seven regions, $\{\{r_1, r_2\}, \{r_1, r_2, r_3\}, \{r_1, r_2, r_4\}, \{r_3, r_4\}, \{r_1, r_3, r_4\}, \{r_2, r_3, r_4\}, \{r_1, r_2, r_3, r_4\}\}$.

In this space, if region r contains r', it is denoted as $r' \leq r$.
Let $I(r)$ denote the set of one node regions, i.e. $I(r) = \prod_i r_i \cdot \prod_{j \neq i} \overline{r_j}$.
Let $I_i(r)$ denote the node i, i.e. $I_i(r) = r_i \cdot \prod_{j \neq i} \overline{r_j}$.

7.4.2.2 Structural Validity of Regions

Recall that in the previous section we placed restrictions on valid regions. In our example, $\{r_1, r_4\}$ is *not* a structurally valid region, and $\{r_1, r_2\}$ *is* one. A structurally valid region can be encoded using its upper and lower "frontiers". Referring again to our example, the upper frontier of $\{r_1, r_2\}$ is $\{r_1\}$ and the lower frontier is $\{r_3, r_4, r_5\}$. We will henceforth refer to structurally valid regions as just *valid* regions. Using well known techniques to traverse a graph implicitly, we can write a function $StValid(r)$ to specify all structurally valid (contiguous) regions. We can also compute relations that denote the upper and lower frontiers of a region $UFront(r, r^U)$ and $LFront(r, r^L)$.

Let $G(r, r')$ represent the connectivity in the graph, i.e. $G(a, b) = 1$ implies node b is fanout of node a.

Let $G^*(r, r')$ represent the connectivity information in the network as follows: If $G^*(a, b) = 1$ then there is a path in the network from node a to node b. G^* is also called the closure of the network, and may be computed using a fixed point formulation.

A region is structurally valid if it is contiguous, i.e. each successor, which is also a predecessor is also in the region.

$$StValid(r) \;=\; \forall_{s,t}(Succ(r,s) \cdot Pred(r,t) \Rightarrow \prod_i (I_i(s) \cdot I_i(t) \Rightarrow I_i(r)))$$

Let $Succ(r, r')$ represent the successor relation, i.e. $Succ(a, b) = 1$ implies that region b is in the transitive fanout of region a. Similarly, let $Pred(r, r')$ represent the predecessor relation, i.e. $Pred(a, b) = 1$ implies that region a is in the transitive fanout of region b.

All relations can be represented as predicates over the extended r space. For example, to compute the $Succ(r, r')$ region, we can iterate over each singleton node $I_i(r')$, a region r is a successor if either r is the singleton node itself $I_i(r) = 1$ or the region r contains a singleton node s $(s \leq r)$ such that s is successor of some singleton node t $(G^*(s, t) = 1)$, which turns out to be the singleton node itself $(I_i(t) = 1)$.

Thus, we can compute all the required relation using such predicates on the extended space. To summarize:

$$\begin{aligned} Succ(r,r') &= \prod_{i \in Network\ Nodes} (I_i(r') \Leftrightarrow (I_i(r) + \exists_{s,t} G^*(s,t) \cdot (s \leq r) \cdot I_i(t)) \\ Pred(r,r') &= \prod_{i \in Network\ Nodes} (I_i(r) \Leftrightarrow (I_i(r') + \exists_{s,t} G^*(s,t) \cdot (t \leq r') \cdot I_i(s)) \end{aligned}$$

$LFront(r, r')$ of a region r consists of those predecessors of r that are not already part of r. It can be computed by finding all regions of predecessors of nodes in the region that are not in the region.

$$LFront(r,r') \;=\; \exists_s Pred(r,s) \cdot \prod_{i \in Network\ Nodes} (I_i(r') \Leftrightarrow I_i(s) \cdot \overline{I_i(r)} \cdot (\exists_{u,t} G(u,t) \cdot I_i(u) \cdot (t \leq r)))$$

A similar computation can be made for $UFront$. We will not discuss their computation any further. We also require a sub-region to have enough behavior to implement the new

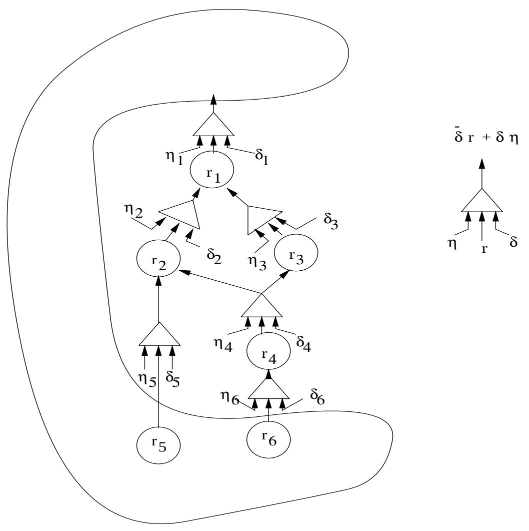


Figure 7.6: Transformed network for computing universal relation

functionality. We call this requirement *sufficiency*. The rest of this section will build tools to implicitly characterize sufficiency. The encoding of a valid region in terms of its upper and lower frontier is crucial for this characterization.

7.4.2.3 Universal Relation

Recall that our aim is to implicitly enumerate all valid regions that are sufficient for re-synthesis. We transform the network by introducing two variables δ_i and η_i, for each node, represented by some variable r_i in the extended region.

Figure 7.6 shows the new variables introduced and the associated transformed network for our example. The functionality of the triangle-shaped node is given by $r_i\overline{\delta_i} + \eta_i\delta_i$. i.e, it uses δ_i as a selector variable to choose between r_i and η_i. We then construct a *Universal Relation*, $U(x, z, r, \delta, \eta)$, which is the observability relation of the transformed network, with all the r_i's made observable. The intuition behind construction the universal relation is that it encodes the N relation (refer to Theorem 7.2) for all sub-regions of R. To obtain the N relation for a sub-region S from U:

- Set the δ_i's corresponding to the upper frontier of S to 1 and all other δ variables to 0.

- Existentially quantify all r_i's that are *not* in the lower frontier of S.
- Existentially quantify all η_i's that are *not* in the upper frontier of S.

In the example, $\exists_{\eta_1\eta_4\eta_5\eta_6}\exists_{r_1r_2r_3r_6}U(x,z,r,\delta,\eta)|_{\delta=(0,1,1,0,0,0)}$ yields the N relation for the sub-region $\{r_2,r_3\}$.

7.4.2.4 Dynamic Quantification

We need one last step in implicitly enumerating all sufficient and valid sub-regions. We need to be able to do the quantifications mentioned above "on the fly". For this purpose, we introduce additional sets of variables $\hat{r}$ and $\hat{\eta}$, to get the *dynamically quantified* universal relation $U^{DQ}(x,z,r,\hat{r},\delta,\eta,\hat{\eta})$. The idea is that existential quantifications of U can be transformed into co-factor operations in U^{DQ}. For example, $\exists_{\eta_1\eta_4\eta_5\eta_6}\exists_{r_1r_2r_3r_6}U(x,z,r,\delta,\eta)|_{\delta=(0,1,1,0,0,0)}$ will be equivalent to $U^{DQ}(x,z,r,\hat{r},\delta,\eta,\hat{\eta})|_{\delta=(0,1,1,0,0,0),\hat{r}=(0,0,0,1,1,0),\hat{\eta}=(0,1,1,0,0,0)}$.

7.4.2.5 Computation of U^{DQ} from U

We consider the following abstract version of the problem. Let a_s denote the projection of a with respect to a set of indices s and let $\overline{s}$ denote the complement of s. For example, if $a=(1,1,0,1,0,1)$ and $s=\{1,3,6\}$, then a_s is $(1,0,1)$ and $a_{\overline{s}}$ is $(1,1,0)$. Given some relation $T(x,y)$, we seek to compute $T^{DQ}(x,\hat{x},y)$ such that $(x,\hat{x},y)\in T^{DQ}$ iff $(x_s,y)\in\exists_{x_{\overline{s}}}T(x,y)$, where s represents the set of positions where $\hat{x}$ has 1's.

To understand this definition better, let us consider some special cases first. When $\hat{x}$ is a vector of all 1's we seek the original relation $T(x,y)$. When $\hat{x}$ is a vector of all 0's, we seek the relation $\exists_x T(x,y)$. When $\hat{x}$ is a vector of some 0's and some 1's, we seek to "dynamically quantify" some of the x_i's from T depending on the value of $\hat{x}$.

The following theorem gives a procedure for computing T^{DQ}.

Theorem 7.6 *If*

$$\begin{aligned} T^{EQ}(x,\hat{x},y) &= \exists_z\{T(z,y)\cdot\prod_i[\hat{x}_i\Rightarrow(x_i\Leftrightarrow z_i)]\} \\ &\Rightarrow\ T^{EQ}=T^{DQ} \end{aligned}$$

Proof.

- $T^{EQ} \supseteq T^{DQ}$.

  Let $(a,b,c) \in T^{DQ}$. This implies that if we existentially quantify from T, the x_i's corresponding to the 0 entries of b, the resultant relation will contain (a_s, c), where s is the set of indices corresponding to the 1 entries of b. Consequently, there must exist an entry in T that matches with (a,c) in positions defined by the s. By the definition of T^{EQ} it must contain (a,b,c).

- $T^{EQ} \subseteq T^{DQ}$.

  Let $(a,b,c) \in T^{EQ}$ and let s be the set of indices corresponding to the 1 entries of b. This means that there exists $(\hat{a}, c) \in T$, where $\hat{a}$ and a match at positions defined by s. Thus $(a,b,c) \in T^{DQ}$.

□

Note that T is related to T^{DQ} in the same way that U is related to U^{DQ}. We can directly apply Theorem 7.6 to compute U^{DQ} from U.

7.4.2.6 Implicit Enumeration of Valid and Sufficient Regions

We now have all the tools necessary to implicitly enumerate all regions that are structurally valid *and* are sufficient to realize the changed functionality. Recall the definition of relations $StValid$, $LFront$ and $UFront$. The enumeration is provided by the following theorem:

Theorem 7.7 *Let r^I denote a vector that specifies the initial region R. A subregion S specified by s is valid and sufficient for re-synthesis if and only if*

$$\{(s \leq r^I) \cdot StValid(s) \cdot \forall_{s_L} \eta_U [UFront(s, \eta_U) \cdot LFront(s, s_L) \Rightarrow Sufficient(s_L, \eta_U)]\} \quad = \quad 1$$

where $Sufficient(s_L, \eta_U)$ is defined as:

$$\forall_r \exists_\eta \{\forall_{xz}[\exists_\delta\{(\delta = \eta) \cdot U^{DQ}(x,z,r,\hat{s}_L,\delta,\eta,\hat{\eta}_U)\}] \Rightarrow O^F{}_{new}(x,z)\}$$

Proof. By combining the conditions for structural validity and sufficiency of implementation. The condition $(s \leq r^I)$ ensures that S is a sub-region of R. The definition of $Sufficient(s_L, \eta_U)$ is just a restatement of Theorem 7.2 with the new tools in hand. □

7.4.3 Exact Algorithm

If the initial region R is taken to be the entire circuit, Theorem 7.7 can be used to implicitly enumerate all valid and sufficient re-synthesis regions for the entire circuit. We could represent these solutions as a BDD and search through the solution space for the desired optimum solution. Our objective is to minimize the number of nodes in the region of re-synthesis. For this objective, the problem reduces to finding a path from the root of the BDD to the terminal "1-vertex", such that the number of edges with label 1 is minimized. This is a shortest path problem and can be solved in time proportional to the size of the BDD.

The cost function we use for defining the optimal sub-region is the number of nodes. In principle, one can also handle more complex cost functions like weighting each node by the number of literals in its factored form.

7.4.4 Heuristic Algorithm

In practice, the exact algorithm fails when handling larger circuits. We propose an iterative algorithm first adds nodes to the region, one at a time, while maintaining contiguity until a feasible region is obtained. Then, it removes nodes, while maintaining contiguity until a minimal region is obtained. In reality, *not all minimal regions are comparable.* Some have better results in terms of power, area or delay. Thus, we need to come up with an objective that decides which minimal regions are better for re-synthesis. This leads to the idea of an objective sensitivity:

In order to evaluate nodes to be chosen to add to a re-synthesis region, we use a measure called the sensitivity of a node (or region).

Definition 54 *The sensitivity of a node is the expected decrease in the objective function (area, delay or power) that is expected if the node is re-synthesized.*

Obviously, it is too expensive to calculate the exact change in objective; in part because we do not know the exact change. Hence, sensitivity is an *estimate.*

In the experimental section, we will describe one measure of sensitivity to power. The sensitivity allows us to choose nodes so as to get the minimal re-synthesis region that also has the best power performance.

The iterative algorithms works as follows: First, the designer is allowed to mark out regions that may not be re-synthesized. For the remaining network, we compute the approximate sensitivities for every node in the network. We use an iterative algorithm that progressively adds nodes to the re-synthesis region.

Algorithm 7.1

```
Mark regions designer wants unchanged ($M$)
$R = \emptyset$
While $R$ not sufficient (Theorem 7.2)
    $p$ = Node of highest sensitivity (excluding $M$ and $R$)
    $R = R + p$
Reduce $R$ to get a minimal region (Section 7.4.1)
Compute $u = R(v)$ (Theorem 7.5)
Synthesize $u = R(v)$ to get $R_{new}$
replace $R$ with $R_{new}$
return
```

We add the node of highest sensitivity to the resynthesis region R, and examine whether re-synthesizing the current region R could achieve the new functionality. If not, we add the nodes of next highest sensitivity to the region and repeat the process. This greedy process iteratively tries to determine a small partition of the initial circuit that has high potential for gains in the objective (power for our current implementation) that can be re-synthesized in order to implement the new functionality. In order to make the region minimal (Section 7.4.1), we post process it by attempting to remove nodes to get a smaller feasible region.

7.5 Experiments and Results

In the previous sections, we have described how to identify sufficient regions, a brief explanation for measures of picking nodes and an iterative algorithm that combines the both. In this section, we will experimentally examine these strategies.

We have implemented the exact and heuristic algorithms and sensitivity measures described in this chapter in SIS [57]. Though not explicitly stated during this chapter, we used the BDDs to represent our functions and relations. Logical predicates may be represented as a sequence of BDD operations; we used BDD's for the computation of logical predicates.

To emulate a design change, we took benchmark examples with external don't cares and optimized them to obtain a circuit called "old". The new spec F_{new} for each example is the same circuit benchmark but without the external don't cares given. In some circuits without external don't cares, we wrote a script that flipped random bits in the tables of the blif file. This procedure yielded a new specification called "new".

7.5.1 Experiments with Minimum Re-Synthesis

We used three methods to do incremental re-synthesis: the heuristic method proposed in Section 7.4.2, the exact method described in section 7.4.3 and the hybrid approach that combines both. Our cost function is the total number of nodes in the re-synthesis region. We tabulate the results of our experiment on some sample examples in Table 7.1. Theoretically, the exact method should always produce the minimum number of nodes in the region, and the hybrid method must always be as good as or better than the heuristic method. The experimental results concur with this. A "*" entry denotes that the experiment did not complete because we ran out of memory. As expected, the hybrid method completes in some cases where the exact method runs out of memory.

No optimization scripts were used to get the given results that compare the exact method with the heuristic and hybrid methods.

7.5.2 Experiments with Low Power Re-Synthesis

Our primary objective is to preserve as much of the old implementation as possible by re-synthesizing the re-synthesis region alone. However, we would also like to get "good" power results while preserving as much of the old network structure as possible.

In particular, we will experiment with networks and the optimality criteria of power. Thus, we need to illustrate how to compute a power sensitivity; i.e. a criteria to pick nodes that

	♯ Total	♯ Region			Run Time (seconds)		
Example	Nodes	Heuristic	Exact	Hybrid	Heuristic	Exact	Hybrid
alu2	32	7	*	5	6.4	*	22.5
alu3	35	8	*	4	9.7	*	25.4
C17	10	2	2	2	0.1	0.2	0.1
dekoder	21	4	2	2	0.2	1.9	0.3
dk27	35	14	*	10	2.1	*	24.7
b1	10	2	1	1	0.1	0.1	0.1
cm138a	25	7	2	2	0.3	1.0	0.9
cm151a	20	3	2	2	0.2	0.9	0.9
cm42a	30	2	2	2	0.2	3.6	0.3
cm82a	24	3	1	2	0.1	0.3	0.1
x2	26	5	2	2	0.5	3.9	1.1
z4ml	16	4	3	4	0.5	1.5	1.3

Table 7.1: Exact Vs. Heuristic Algorithms

give better power numbers when re-synthesized.

7.5.2.1 Illustration: Power Sensitivity

To illustrate a measure for sensitivity for objectives like area, delay and power, we pick a sensitivity for power.

The definition of sensitivity is easily extended to regions. We rely on the work done by Lennard [78] et al on computing the power sensitivity of a node. A node n is a good candidate for re-synthesis if local change in activity (power) plus change in activity in the transitive fanout reduces overall power. A method for determining expected activity $E(n)$ is outlined below:

Consider a node n in the network with immediate fanins $n_1 \ldots n_m$ (refer to Figure 7.7). Node n computes a function $f_n(n_1 \ldots n_m)$ of its fanins. Let A_{n_1} denote an arbitrary set of minterms that are added to the onset of fanin n_1. Let this be the only change made to the fanins of n. The set of of minterms that are added to the onset of f_n are those minterms in A_{n_1} that actually change the value of the function f_n from 0 to 1. Similarly, the set of minterms that are removed from the onset of f_n are those minterms in A_{n_1} that actually

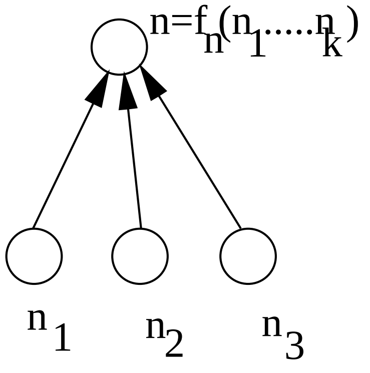


Figure 7.7: Sensitivity computation

change the value of f_n from 1 to 0. Let A_n denote the set of minterms that are added to the onset of f_n and R_n denote the set of minterms that are removed from the onset of f_n due to this change. A_n and R_n can be computed as follows (Recall that $\Omega_I(f(x))$ is the projection of points in $f(x)$ onto the input space):

1. $A_n(A_{n_1}) = \Omega_I(f_n|_{n1=1} \cdot \overline{f_n|_{n1=0}}) \cdot A_{n_1}$.
2. $R_n(A_{n_1}) = \Omega_I(\overline{f_n|_{n1=1}} \cdot f_n|_{n1=0}) \cdot A_{n_1}$.

The quantities $S^+{}_n(n_1) = f_n|_{n1=1} \cdot \overline{f_n|_{n1=0}}$, and $S^-{}_n(n_1) = \overline{f_n|_{n1=1}} \cdot f_n|_{n1=0}$ are called the functional positive and negative sensitivities. Similar measures can be computed for the set of minterms that are added and subtracted from the onset of f_n, when R_{n_1} minterms are subtracted from the onset of its fanin (actually by taking 1 and 2 and interchanging added A and removed R).

These quantities have no real significance as yet, since we do not know the exact change that is actually made to an internal node of the network. However, if we assume that *any change in the onset size is equally likely*, the expected size of the sets A_n and R_n can be computed with just the knowledge of the size of the change (without knowing the actual minterms in the change!) by computing the expectations of the quantities in equations (1) and (2):

- $E(|A_n(A_{n_1})|) = |A_{n_1}| \frac{|\Omega_I(S^+{}_n(n_1)) \cdot \overline{f_{n_1}}|}{|\Omega_I(\overline{f_{n_1}})|}$

- $E(|R_n(A_{n_1})|) = |A_{n_1}| \frac{|\Omega_I(S^{-}{}_n(n_1)) \cdot f_{n_1}|}{|\Omega_I(f_{n_1})|}$

The expected size of the set A_n is a measure of the change in switching for the given node. Thus, we can compute a switching probability p by dividing the expected size of the onset change $E(A_n(A_{n_1}))$ by the original size of onset.

Given a probability p of evaluating to high at a node, the functional transition activity is given by 2(1-p)p. Switching happens if the signal switches from 0 to 1 (with probabilities $1-p$ and p respectively), or from 1 to 0. Hence probability of the $0, 1$ switch is $(1-p)p$, and probability of the $1, 0$ switch is $p(1-p)$. The total probability is $2p(1-p)$. Thus, a p far from 0.5 implies a smaller transition activity. Given a change in onset size at a given node n_1, the expected change in onset size can be derived for all nodes in the transitive fanout of n_1. Thus, the expected onset sizes of nodes directly relate to their switching probabilities; the higher the estimates the more the sensitive the node.

We use the estimated changes $E(A_n(A_{n_1}))$ and $E(R_n(A_{n_1}))$, as sensitivity (estimated) measures of how much switching change can result at output n from re-synthesizing node n_1. If this number is high, then node n is very *sensitive* to changes in node n_1.

We use this analysis to compute a "good" ordering of nodes for re-synthesis; we prefer to add nodes with high sensitivity to the re-synthesis region.

The efficacy of this measure of the sensitivity of a node has been demonstrated statistically in [78]. For the purpose of this chapter, we will not discuss this further.

7.5.2.2 Experiments with Power Sensitivity based Re-Synthesis

In general our primary goal conflicts with an objective of minimal power; assuming we have good power optimization routine, we can always get better power results by completely ignoring "old" (hence we have more flexibility). Thus we expect to get worse power results for the incremental algorithm as compared to "new". However, using a good measure of power sensitivity to pick nodes to add to the re-synthesis region should give us better power results than using any random method to pick the nodes. We expect the power numbers, $P_{new} \leq P_{sensitivity} \leq P_{random}$. There may be many minimal regions in a network that can implement the same functional change; our objective is to pick the minimal region that gives good power results.

Thus, to summarize, we ran the following experiment: we optimized the old and new specifications using a power optimization scripts, we also implemented the new specification by incrementally re-synthesizing the optimized old network to get the new functionality. In our experiments, we used two methods for choosing the region for re-synthesis: "sensitivity" and "random". In "sensitivity", we chose the new node to be put in the region of re synthesis according to the power sensitivity measure discussed in Section 7.5.2.1. In "random" we chose the new node randomly. Note that both approaches do not take the ability of a node to re-synthesize the new function into account when choosing a node. The iterative algorithm (Section 7.4.4) and the greedy search for a minimal region (Section 7.4.1) attempt to minimize the resynthesis region. We compared the sizes of the regions, as well as total power of the resultant network obtained for both random and sensitivity based measures. We used two different scripts for power optimization, the first was from Buch et al [81], and the second was the script.rugged script from SIS [57].

Figure 7.2 summarizes the percentage of the network preserved; these numbers were consistently over 50% and quite often as high as 90%. We were indeed preserving large portions of the old network. However, at this stage we do not know what the exact minimum answer is; our future work will determine this.

We also tabulate the results of our experiment on the first power script in Table 7.3. Isyn denotes the results obtained by our incremental synthesis and Nsyn denotes results obtained by complete resynthesis. We see that for 10 of the 14 examples in Table 7.3, "sensitivity" produces circuits with equal or better power numbers than "random". In four examples, "random" produces circuits with lower power. It was also interesting to note that a circuit like cu could be fixed to the new functionality by re-synthesizing just one node, and this gave power results which were just as good as re-synthesizing the entire circuit. The power numbers were computed assuming a $20Mhz$ clock and a Vdd of $5v$.

We also experimented with another optimization script (script.rugged without don't care optimization), which was not specifically targeted for power. Table 7.4 reports the results for this script. Even using this script, out of the 14 examples, 10 showed better or equivalent results for the sensitivity based method.

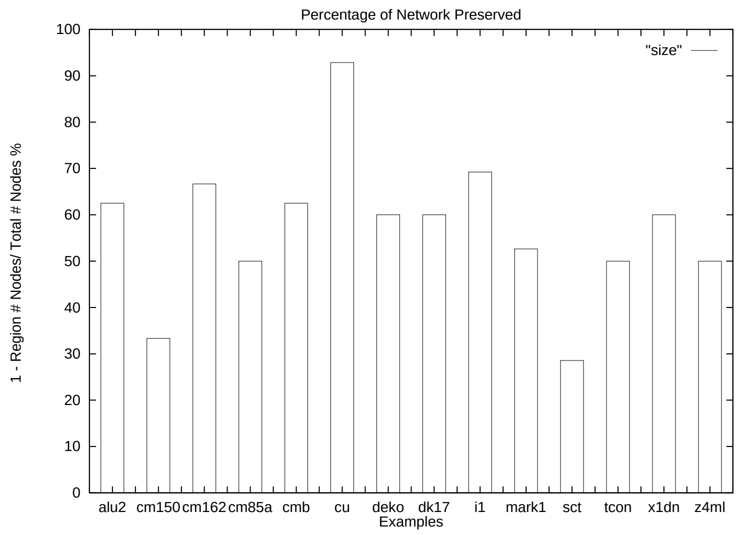


Table 7.2: Size of the Re-synthesis Region

	♯ Total	♯ Region		ISyn Power		NSyn Power	
Example	Nodes	Sensitivity	Random	Sensitivity	Random	Old	New
alu2	16	6	6	589.4	589.4	583.1	478.6
cm150a	9	6	6	272.8	272.8	243.4	244.6
cm162a	12	5	4	205.2	247.7	163.4	177.3
cm85a	6	3	3	291	274.1	201.3	189.1
cmb	8	5	3	279.6	316.4	229.3	256.5
cu	14	1	1	214.6	224.6	214.6	214.6
dekoder	10	4	4	206.9	208.3	199.8	183.8
dk17	30	15	12	833.7	517.7	310.9	263.3
i1	13	4	4	183.6	183.6	193	183.6
mark1	57	30	27	538.5	605	524	267
sct	21	12	15	341.8	315	288.1	283.9
tcon	8	5	4	133.4	133.4	140	133.4
x1dn	10	5	4	545.7	545.7	443.8	435.9
z4ml	8	3	4	315.2	276.5	163.4	169.2

Table 7.3: Script1: Sensitivity Vs. Random Regions

	♯ Total	♯ Region		ISyn Power		NSyn Power	
Example	Nodes	Sensitivity	Random	Sensitivity	Random	Old	New
alu2	21	10	12	1093.1	1169.6	502.8	411.6
cm150a	9	3	3	275.4	275.4	243.4	244.6
cm162a	12	5	6	202	248.9	163.4	177.3
cm85a	6	3	3	281	237.3	201.3	189.1
cmb	8	6	3	261.5	292.3	229.3	256.5
cu	14	1	1	214.6	214.6	214.6	214.6
dekoder	11	2	4	199.7	204.4	173.1	180.1
dk17	31	16	11	563.8	666.9	289.2	258
i1	13	4	5	183.6	183.6	193	183.6
mark1	57	27	26	421.1	452.3	496.5	254.4
sct	21	13	14	294.6	274.8	288.1	283.9
tcon	8	5	5	133.4	133.4	140	133.4
x1dn	10	5	3	509.1	481.2	443.8	435.9
z4ml	8	4	4	267.1	240.1	163.4	169.2

Table 7.4: Script2: Sensitivity Vs. Random Regions

In only 4 examples did the random measure have smaller regions. Thus, on an average the sensitivity was producing smaller re-synthesis regions, with better power results. This is surprising since smaller regions mean less flexibility and hence higher power. This is attributable to the fact the sensitivity measure is picking good nodes for power re-synthesis.

7.5.3 Conclusions and Future Work

Given an original network and a changed specification, we have shown how to realize the new specification while preserving much of the old network. In particular, we have defined and used a measure of power sensitivity of the node, and shown that by choosing nodes for re-synthesis according to this , we get mostly better results than any random selection of nodes. Our method of re-synthesis is effective in preserving much of the old network. It is also effective in picking the re-synthesis region so as to get good power results (as compared to any other random strategy).

As part of future work, it is necessary to examine different measures of the sensitivity of a node (region) (wrt to different objectives), and evaluate the performance of the iterative algorithm using these strategies. In this chapter, we have described both a greedy strategy for re-synthesis, and an exact strategy for minimum sufficient sub-region. The exact technique used enumerates all sufficient sub-regions implicitly and searches for the best solution using any desired cost function. If the given region of re-synthesis is taken to be the entire circuit, the algorithm can be theoretically used to solve the minimum re-synthesis problem. We also gave a hybrid algorithm to be able to handle larger circuits.

Because of the way we have defined structural validity, we do not consider regions in which inputs to the region are dependent on the outputs from the region. Finding the minimum region with the structural validity condition relaxed is an open problem.

We find that we can handle circuits of modest sizes at this time. One possibility is to use don't cares to minimize our intermediate BDDs. We have a lot of don't cares because we care about only structurally valid regions. However, we find that functions like "bdd_between" in our BDD package are unable to use the don't cares effectively. Handling large sets of don't cares in minimizing BDDs effectively will increase the sizes of circuits we can handle. Heuristics for early quantification, dynamic re-ordering in the presence of large don't care sets will be helpful.

In Section 7.3 we implemented one particular function from the entire class of possible functions. One future extension is to examine the entire class of solutions for the most optimal implementation, using the work of Watanabe et al [82] on heuristic Boolean minimization. Since we are recomputing a new ${O^R}_{\text{new}}(v, u)$ many times during the iterative algorithm, it becomes pertinent to explore incremental ways of updating the relation, rather than recomputing it from the beginning. We expect that some of the methods adapted from [83] may be used.

Chapter 8

Conclusions and Future Work

The main contributions of this thesis have been:

1. To motivate the need for incremental algorithms in CAD; in particular logic synthesis and formal verification.
2. To provide a theoretical framework for the construction of such incremental algorithms.
3. To construct incremental algorithms for formal verification and logic synthesis and experimentally demonstrate their utility.

We have addressed all these aspects in this thesis.

In Chapter 2, we have proved some key theorems on fixed point algorithms. Most algorithms in formal verification are of this form. We used these insights to construct incremental algorithms for verification. We began by addressing the problem of FSM traversal, the key FSM verification operation, in Chapter 4. We showed how by storing some variant of the traversal graph, we are able to update the FSM traversal information more efficiently in the event of change. We examined the effectiveness of storing different variants of the traversal information.

In Chapter 3 we began by identifying how commonality between two designs represented as networks may be detected. We presented both a heuristic and an exact method for determining commonality, and showed that the proposed heuristic performed almost as well as the exact method. We also showed that considerable gains could be achieved by re-use of information.

Next in Chapter 5 we examined the problem of language containment, a key computation in verification, and showed how incremental algorithms may be constructed. We demonstrated that significants gains may be obtained by the use of these algorithms. In Chapter 6 we extended the arguments to the problem of model checking, another method of formal design verification. We briefly discussed the construction of canonical structures that ease the detection of change.

In Chapter 7, we break away from the verification problem, and address the incremental synthesis problem. We briefly discuss previous approaches to the problem and distinguish them from our approach. The key improvement lies in the fact that we propose a methodology that tries to preserve optimal and hand optimized regions of the design. We conclude with results of the effectiveness of our approach.

This thesis is just the first step towards an iterative prototyping environment, which would fit in better with the design flow. A key failing of this work is the fact that it is not possible to gain access to such an environment, so we used random changes to simulate the design process.

8.1 Future Work

It is evident that there is a tradeoff between space and time in the incremental approach. We give up in space by storing information from previous design iterations, in the hope that this may give us gains in time. However, it is possible to construct examples, where this tradeoff does not pay. Thus, one avenue of future research is the examination of methods to determine when to apply the incremental methods, and when to run the entire computation from the beginning. The incremental methods need only be used when we are guaranteed to have gains from its use. For large changes where there is no common information between the old and new design the incremental algorithm need not be applied.

There is also the issue of how many previous iterations to store, as well as which information to store. Can we get significant gains form storing more than one design iteration ? This issue needs to be addressed. A point of note is that recently, there has been a trend in the synthesis industry towards re-usable design blocks, and this fits in with our philosophy of re-use.

Another relevant avenue is of research is how information for re-use may be written out and stored in main memory for future use. Most of our computations are done using BDD's. It would be interesting to formulate a method for storing BDD managers with all component BDDs in main memory for later re-use. This allows for libraries of results and computations to be stored for future reference.

It is also important to re-implement all the mentioned algorithms in the context of new on-going research in efficient verification and synthesis; e.g. improvements like partial product heuristics, partitioned systems, more efficient BDD manipulation etc. Our methods have not made use of don't care information to efficiently compute incremental sets. We can extend our analysis to utilize the small BDDs that may be obtained by the use of don't cares. For example, instead of running the RRS computation on page 90, on $\triangle T^{add}$, it might make more sense to minimize $\triangle T^{add}$ with existing transitions in T^{old} whose behavior is already included in the old fair set. To conclude, this research is just one step in the direction. The real proof of the pudding lies in the development of an iterative design rapid prototyping system with all these incremental algorithms in place. Such a prototyping system would read in a high level description of the system, and keep track of the incremental changes. If the differences are small and incremental, one way to do this is using the methods used in incremental compilers. Then, these incremental differences need to be propagated down the entire design flow. In the framework that our research was described, first the network that is built from the high-level description is incrementally rebuilt. This would involve adding and subtracting portions of logic (gates, latch elements etc) to the existing network. Next the BDD ordering information must be updated. If new latches or inputs have to be added to the system, the corresponding variables must be added in to the old BDD ordering. In this, we are helped by dynamic ordering techniques, which can re-order upon the introduction of these new variables. Alternately, the static ordering algorithms can also be incrementalized to add in new BDD variables in the existing old order. The new variables must also be added /subtracted from the existing data structures for image computation, and further propagated through the new network, while using the methods described to re-use as much of the old information as possible to update the transition function or relation BDD's. Then, these updated functions can be used in conjunction with the incremental verification algorithm described to get more efficient verification. A similar incremental design flow can be proposed for synthesis of the circuits.

There is no doubt that such an approach to CAD is more suited to fitting in with the way IC design is done.

Succinct versions of this work can be found in [49], [50], [84], [85], [86].

Bibliography

[1] D. Thomas and P. Moorby, "The Verilog Hardware Description Language," (Nowell, MA), Kluwer Academic Publishers, 1991.

[2] B. Kernighan and D. Ritchie, *The C Programming Language.* Prentice Hall, 1978.

[3] "IEEE Standarad VHDL Language Reference Manual," vol. Std 1076-1993, IEEE, 1993.

[4] R. A. Walker and R. Camposano, *A Survey of High Level Synthesis Systems.* Kluwer Academic Publishers, Boston, 1991.

[5] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," Tech. Rep. UCB/ERL M92/41, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, May 1992.

[6] R. K. Brayton et al., "VIS: A System for Verification and Synthesis," in *Proc. of the Conf. on Computer-Aided Verification*, pp. 428–432, 1996.

[7] M. A. Breuer and A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems.* Computer Science Press, 1976.

[8] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill, "Symbolic Model Checking: 10^{20} States and Beyond," *Information and Computation*, vol. 98, no. 2, pp. 142–170, 1992.

[9] K. L. McMillan, *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[10] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel Logic Synthesis," *Proceedings of the IEEE*, vol. 78, pp. 264–300, Feb. 1990.

[11] H. Touati, C. Moon, R. K. Brayton, and A. Wang, "Performance-Oriented Technology Mapping," in *Proc. sixth MIT VLSI Conf.*, pp. 79–97, Apr. 1990.

[12] B. Preas and M. Lorenzetti, *Physical Design Automation of VLSI Systems.* Benjamin Cummings Publishing, 1988.

[13] F. Balarin, "Iterative Methods for Formal Verification of Digital Systems," Tech. Rep. Doctoral Thesis, UCB/ERL M94/ , Electronics Research Lab, Univ. of California, Berkeley, CA 94720, 1994.

[14] H. Touati, R. K. Brayton, and R. P. Kurshan, "Checking Language Containment using BDDs," in *Proc. of Intl. Workshop on Formal Methods in VLSI Design*, (Miami, FL), Jan. 1990.

[15] R. Hojati, T. R. Shiple, R. K. Brayton, and R. P. Kurshan, "A Unified Environment for Language Containment and Fair CTL Model Checking," in *Proc. of the Design Automation Conf.*, (Dallas, Texas), pp. 475–481, June 1993.

[16] R. P. Kurshan, *Automata-Theoretic Verification of Coordinating Processes.* Princeton University Press, 1993.

[17] W. Thomas, "Automata on Infinite Objects," in *Formal Models and Semantics* (J. van Leeuwen, ed.), vol. B of *Handbook of Theoretical Computer Science*, pp. 133–191, Elsevier Science, 1990.

[18] Z. Har'El and R. P. Kurshan, "Software for Analytical Development of Communication Protocols," *AT&T Technical Journal*, pp. 45–59, Jan. 1990.

[19] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Trans. Programming Languages and Systems*, vol. 8, no. 2, pp. 244–263, 1986.

[20] O. Coudert and J. C. Madre, "A Unified Framework for the Formal Verification of Sequential Circuits," in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 126–129, Nov. 1990.

[21] A. Aziz, "Formal Methods in VLSI System Design," Tech. Rep. Doctoral Thesis, UCB/ERL , Electronics Research Lab, Univ. of California, Berkeley, CA 94720, 1996.

[22] E. A. Emerson and C. L. Lei, "Modalities for Model Checking: Branching Time Strikes Back," in *Proc. ACM Symposium on Principles of Programming Languages*, pp. 84–96, 1985.

[23] R. K. Brayton, R. Rudell, A. L. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A Multiple-Level Logic Optimization System," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 1062–1081, Nov. 1987.

[24] G. Ramalingam and T. Reps, "On the Computational Complexity of Incremental Algorithms," Tech. Rep. TR 1033, University of Wisconsion, Madison, University of Wisconsion, Madison, 1991.

[25] T. Reps, T. Teitelbaum, and A. Demers, "Incremental context dependent analysis for language-based editors," *ACM Transactions on Programming Languages*, pp. 449–477, 1983.

[26] D. Yellin and R. Strom, "INC: A language for incremental computations," *ACM Transactions on Programming Languages*, pp. 211–236, 1991.

[27] F. Zadek, "Incremental data flow analysis in a structured program editor," *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction*, pp. 132–143, 1984.

[28] G. Taylor and J. Ousterhout, "Magic's Incremental Design Rule Checker," *Proc. of the Design Automation Conf.*, pp. 702–708, 1984.

[29] P. Swartz and K. Mednick, "Incremental Design Rule Checking," *IEEE Custom Integrated Circuits Conference*, pp. 734–739, 1985.

[30] G. Cheston, "Incremental Algorithms in Graph Theory," Tech. Rep. Doctoral Thesis, Univ. of Toronto, Canada, 1976.

[31] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms.* Mcgraw Hill, 1990.

[32] O. Sokolosky and S. Smolka, "Incremental Model-Checking in Modal Mu-Calculus," in *Proc. of the Conf. on Computer-Aided Verification*, vol. 818, pp. 351–363, Springer-Verlag, June 1994.

[33] Y. K. etal, "Efficient Prototyping System Based on Incremental Design and Module by Module Verification," in *Proc. Intl. Symposium on Circuits and Systems*, pp. 924–927, May 1995.

[34] Y. Watanabe and R. K. Brayton, "Incremental Sythesis for Engineering change," in *Workshop Notes of the Intl. Workshop on Logic Synthesis*, (Tahoe City, CA), May 1991.

[35] D. Brand, "The Taming of Sythesis," in *Workshop Notes of the Intl. Workshop on Logic Synthesis*, (Tahoe City, CA), May 1991.

[36] M. Fujita, Y. Tamiya, Y. Kukimoto, and K.-C. Chen, "Application of Boolean unification to combinational logic synthesis," in *Proceedings of IEEE International Conference on Computer-Aided Design*, pp. 510–513, November 1991.

[37] Y. Kukimoto and M. Fujita, "Rectification Method for Lookup-Table Type FPGA's," in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 54–61, Nov. 1992.

[38] C. Lin, K. Chen, S. Chang, M. Marek-Sadowska, and K. Cheng, "Logic Synthesis for Engineering Change," in *Proc. of the Design Automation Conf.*, pp. 647–652, June 1995.

[39] P. Chernoff, "Lectures in topology and analysis," 1993.

[40] R. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," *IEEE Trans. Computers*, vol. C-35, pp. 677–691, Aug. 1986.

[41] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," in *Proc. of the Design Automation Conf.*, pp. 40–45, June 1990.

[42] R. Rudell, "Dynamic Variable Ordering for Binary Decision Diagrams," in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 42–47, Nov. 1993.

[43] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment," in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 6–9, Nov. 1988.

[44] R. K. Ranjan, A. Aziz, R. K. Brayton, B. Plessier, and C. Pixley, "Efficient formal design verification data structure and algorithms," Tech. Rep. UCB/ERL M94/100, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, Oct. 1994.

[45] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Implicit State Enumeration of Finite State Machines using BDD's," in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 130–133, Nov. 1990.

[46] D. Brand, A. Drumm, S. Kundu, and P. Narain, "Incremental Synthesis," in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 14–18, Nov. 1994.

[47] J. Burch and D. Long, "Efficient boolean function matching," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pp. 408–411, November 1992.

[48] R. K. Brayton, M. Chiodo, R. Hojati, T. Kam, K. Kodandapani, R. P. Kurshan, S. Malik, A. L. Sangiovanni-Vincentelli, E. M. Sentovich, T. Shiple, K. J. Singh, and H.-Y. Wang, "BLIF-MV: An Interchange Format for Design Verification and Synthesis," Tech. Rep. UCB/ERL M91/97, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, Nov. 1991.

[49] G. M. Swamy and R. K. Brayton, "Incremental Formal Design Verification," in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 458–465, Nov. 1994.

[50] G. M. Swamy, V. Singhal, and R. K. Brayton, "Incremental methods for Fsm Traversal," in *Proc. Intl. Conf. on Computer Design*, pp. 590–595, Oct. 1995.

[51] J. Hopcroft, "An $n \log n$ algorithm for minimizing states in a finite automaton," in *Theory of Machines and Computations. Proceedings of an International Symposium on the Theory of Machines and Computations.* (Z. Kohavi and A. Paz, eds.), (Haifa, Isreal), pp. 189–196, Academic Press, 1971.

[52] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli, "A Fully Implicit Algorithm for Exact State Minimization," in *Proc. of the Design Automation Conf.*, pp. 684–690, June 1994.

[53] G. M. Swamy, P. Mcgeer, and R. K. Brayton, "An Exact Logic minimizer using BDD based Methods ," Tech. Rep. UCB/ERL M92/127, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, 1992.

[54] G. M. Swamy, P. Mcgeer, and R. K. Brayton, "An Exact Logic minimizer using BDD based Methods ," Tech. Rep. "Masters Thesis" UCB/ERL M93/94, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, 1993.

[55] E. J. McClusky, "Minimization of Boolean Functions," *Bell System Technical Journal*, vol. 35, 1956.

[56] S. Malik, J. Mohnke, and P. Molitor, "Limits of Using Signatures for Permutation Indepedant Boolean Matching," in *Proc. Intl. Workshop on Logic Synthesis*, (Tahoe), May 1995.

[57] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Sequential Circuit Design Using Synthesis and Optimization," in *Proc. Intl. Conf. on Computer Design*, pp. 328–333, Oct. 1992.

[58] H. Cho, G. D. Hachtel, and F. Somenzi, "Redundancy Identification and Removal Based on Implicit State Enumeration," in *Proc. Intl. Conf. on Computer Design*, pp. 77–80, Oct. 1991.

[59] B. Lin and F. Somenzi, "Minimization of Symbolic Relations," in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 88–91, Nov. 1990.

[60] J. Burch, E. Clarke, and D. E. Long, "Representing Circuits More Efficiently in Symbolic Model Checking," in *Proc. of the Design Automation Conf.*, June 1991.

[61] R. K. Ranjan, A. Aziz, R. K. Brayton, B. Plessier, and C. Pixley, "Efficient formal design verification data structure and algorithms," in *Proc. Intl. Workshop on Logic Synthesis*, (Tahoe City, CA), May 1995.

[62] R. Hojati, S. Krishnan, and R. K. Brayton, "Heuristic Algorithms for Early Quantification and Partial Product Minimization," Tech. Rep. UCB/ERL M94/11, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, 1994.

[63] R. Brayton et al., "HSIS: A BDD-Based Environment for Formal Verification," in *Proc. of the Design Automation Conf.*, pp. 454–459, June 1994.

[64] A. Aziz, V. Singhal, G. M. Swamy, and R. K. Brayton, "Minimizing Interacting Finite State Machines," in *Proc. Intl. Conf. on Computer Design*, pp. 255–261, Oct. 1994.

[65] R. S. Streett, "Propositional Dynamic Logic of Looping and Converse is Elementary Decidable," *Information and Control*, vol. 54, pp. 121–141, 1982.

[66] M. O. Rabin, *Automata on Infinite Objects and Church's Problem*, vol. 13 of *Regional Conf. Series in Mathematics.* Providence, Rhode Island: American Mathematical Society, 1972.

[67] M. Y. Vardi and P. L. Wolper, "An Automata-Theoretic Approach to Program Verification," in *Proc. IEEE Symposium on Logic in Computer Science*, pp. 332–334, 1986.

[68] R. Hojati, V. Singhal, and R. K. Brayton, "Edge-Streett/Edge-Rabin Automata Environment for Formal Verification Using Language Containment," Tech. Rep. UCB/ERL M94/12, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, 1994.

[69] R. Milner, *Communication and Concurrency.* New York: Prentice Hall, 1989.

[70] E. A. Emerson, "Temporal and Modal Logic," in *Formal Models and Semantics* (J. van Leeuwen, ed.), vol. B of *Handbook of Theoretical Computer Science*, pp. 996–1072, Elsevier Science, 1990.

[71] The VIS Group, "VIS: A System for Verification and Synthesis," Tech. Rep. UCB/ERL M95/104, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, 1995.

[72] M. R. Garey and D. S. Johnson, *Computers and Intractability.* W. H. Freeman and Co., 1979.

[73] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[74] D. Deharbe and D. Borrione, “Symbolic Model Checking with Past and Future Temporal Modalities: Fundamentals and Algorithms,” in *Current Issues in Electronic Modelling*, vol. 1, Kluwer, 1995.

[75] D. Brand, “Verification of Large Synthesized Designs,” in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 534–537, Nov. 1993.

[76] D. Brand, “Incremental Synthesis,” in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 126–129, Nov. 1992.

[77] Y. Kukimoto and M. Fujita, “Rectification method for lookup-table type FPGA’s,” in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pp. 54–61, November 1992.

[78] C. Lennard, *Estimation Techniques to Guide Low Power Resynthesis Algorithms For Combinational Random CMOS Logic.* PhD thesis, University of California Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, Aug. 1995. Memorandum No. UCB/ERL M95/75.

[79] H. Savoj and R. K. Brayton, “Observability Relations and Observability Don’t Cares,” in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 518–521, Nov. 1991.

[80] F. M. Brown, *Boolean reasoning : the logic of Boolean equations.* Boston : Kluwer Academic Publishers, 1990.

[81] C. Lennard, P. Buch, and A. Newton, “Guided Logic Synthesis for Low Power Design,” in *Proceedings of the International Symposium on Low Power Electronics and Design*, Aug. 1996.

[82] Y. Watanabe and R. K. Brayton, “Heuristic Minimization of Multiple-Valued Relations,” *IEEE Transactions on Computer-Aided Design*, vol. Vol. 12, pp. 1458 – 1472, October 1993.

[83] H. Savoj, *Don’t Cares in Multi-Level Network Optimization.* PhD thesis, University of California Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, May 1992.

[84] G. M. Swamy, V. Singhal, and R. K. Brayton, “Incremental methods for Fsm Traversal,” in *Proc. Intl. Workshop on Logic Synthesis*, (Tahoe), May 1995.

[85] G. M. Swamy, S. Rajamani, C. Lennard, and R. K. Brayton, "Minimal Logic Resynthesis," Tech. Rep. UCB/ERL M96/22, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, 1994.

[86] G. M. Swamy, , S. Edwards, and R. K. Brayton, "Identifying Common Substructure for Incremental Methods," Tech. Rep. UCB/ERL M96/21, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, 1994.