

Université de Montréal

Détermination des bornes de l'activité de commutation des circuits logiques

par

Jindrich Zejda

Département d'Informatique et de Recherche Opérationnelle

Faculté des Arts et des Sciences

Thèse présentée à la Faculté des études supérieures
en vue de l'obtention du grade
de Philosophiæ Doctor (Ph. D.)
en Informatique

Mars, 1999

© Jindrich Zejda, 1999

University of Montreal

Bounding of Switching Activity in Logic Circuits

by

Jindrich Zejda

Department of Computer Science and Operational Research

Faculty of Arts and Sciences

Thesis presented to the Faculty of Graduate Studies
in partial fulfillment of the requirements to obtain the grade
of Philosophiæ Doctor (Ph. D.)
in Computer Science

March, 1999

© Jindrich Zejda, 1999

Université de Montréal
Faculté des études supérieures

Cette thèse intitulée:

Détermination des bornes de l'activité de commutation des circuits logiques

présente par
Jindrich Zejda

a été évaluée par un jury composé des personnes suivantes:

El Mostapha Aboulhamid président-rapporteur

Eduard Cerny directeur de recherche

Nicolas C. Rumin co-directeur de recherche

Marc Feeley membre du jury

John P. Hayes examinateur externe

Thèse acceptée le:

Abstract

Electronic systems became an essential component of our lives. The demand for increased functionality is satisfied by higher integration which, however, make the design process more complex and introduces new problems. These problems must be considered in every step involved in the design of electronic systems.

One of these steps is verification - checking, whether what is manufactured will work properly. Among several properties that must be verified, power consumption is becoming more important than ever before. In the most commonly used CMOS technology a considerable amount of power is consumed due to switching which is described by switching activity. Switching activity in its broad sense is a measure of topological and temporal distribution of signal transitions for given operating environment of the circuit. This thesis addresses the problem of estimation of switching activity.

The thesis presents algorithms, implementation, and results of a new method based on constraint resolution for finding an upper bound on switching activity in the combinational part of a synchronous sequential circuit. The obtained switching activity is the major component for computing circuit power consumption (peak power) and several reliability parameters (e.g., voltage drops in power busses, electromigration). It is a static (input-pattern independent) method. The constraint system representing the circuit is built of constraints defined by gates and the operating environment of the circuit. The variables of the constraint system are all possible waveforms abstracted into four classes and expressed as sets of transitions for each unit of discrete time. The constraints of the constraint system are derived from the gates. Each gate is translated into a projection function which constrains each of its terminals based on the values on all other terminals independently of the netlist distinction between gate inputs and outputs. The method rapidly com-

puts an upper bound on the switching activity. The bound is further tightened by case analysis.

The constraint system captures only local relations between nets and gates. Two major techniques were used to capture a global picture of the circuit and use this acquired information to speed up or improve the analysis. They are reconvergent region analysis and global learning.

The method has two major applications: estimation of peak power and estimation of peak current. Both application were tested with our C++ implementation on ISCAS'85 benchmark circuits and the quality of the results for different heuristics was compared. The results show that each heuristic is more suitable for a different type of circuit. The method achieved values of about 1.5 to 3x, and 1.3 to 4x for the ratio of the upper and lower bounds of the switching and peak switching activity, respectively. The method was also compared with exhaustive simulation on a set of MCNC circuits. The exact¹ value of peak switching activity (peak current) was obtained for all tested MCNC circuits. The exact value of switching activity (power) was obtained on most of the tested MCNC circuits.

The current implementation supports the fixed gate delay model and shares most of the code with a timing verification method based on constraint resolution. The performance is further improved by a parallel implementation of the case analysis on a network of inexpensive workstations. The parallel configuration consists of one master and many slaves. The master is responsible for maintaining the current state of the search space, dynamically deciding which parts must be searched, and distributing jobs to slaves. The scheduling algorithm is slave-failure safe and the master performs periodic state saving for recovery from its own eventual failure. Our C++ implementation shows speedup of 8 on a homogeneous network of 10 workstations, and 47 on a heterogeneous network of 87 workstations.

1. "Exact value" means that the lower bound from the simulation is equal to the upper bound (under the fixed delay gate model).

Résumé

Les systèmes électroniques sont devenus une partie essentielle de notre vie. On peut les trouver partout, que ce soit dans les systèmes de contrôle de feux d'intersection, les systèmes de navigation pour avions et satellites, les systèmes de télécommunication ou les ordinateurs de haute performance. La réponse à la demande de nouvelle fonctionnalité et de plus grande performance nécessite une plus grande intégration. Cette grande intégration génère de nouveaux problèmes - qui ont pu être négligés jusqu'alors. Mais aujourd'hui on doit considérer tout ces problèmes dans chaque étape du processus de développement des systèmes électroniques.

Une de ces étapes est la vérification - étape qui assure qu'une fois le produit est fabriqué, il fonctionne correctement. Parmi les propriétés que l'on doit vérifier, la consommation du courant devient plus importante que jamais. Pour la technologie la plus utilisée aujourd'hui - CMOS - la plus importante partie du courant est causée par la commutation des signaux logiques qui est décrite par l'activité de commutation. L'activité de commutation est une mesure de distribution de transition temporelle et spatiale dans le circuit sous une condition d'environnement donnée. Cette thèse s'adresse au problème d'estimation de l'activité de commutation.

La thèse présente les algorithmes, l'implémentation, et les résultats d'une nouvelle méthode pour trouver une borne supérieure d'activité de commutation dans la partie combinatoire d'un circuit synchrone séquentiel basée sur une résolution de contraintes. L'activité de commutation est un composant majeur pour le calcul de consommation du courant d'alimentation du circuit, et pour plusieurs paramètres de fiabilité (par exemple chute de voltage dans des réseaux de distribution du courant, ou electromigration). La méthode est statique, c'est à dire que son résultat est indépendant des vecteurs de test sur les entrées du circuit.

L'idée de base de la méthode est la suivante: Le circuit électronique est considéré comme un système de contrainte. Les portes logiques et l'environnement du circuit sont représentés par des con-

traintes. Les signaux du circuit sont représentés par les variables du système de contraintes. Toutes les ondes de signaux sur un nœud du circuit sont groupées en quatre classes et représentés par une ensemble de transitions pour chaque intervalle de temps (Figure 1). Les classes sont indépendantes - si une onde réelle est représentée par un sous-ensemble de classe, elle ne peut avoir aucune transition dans n'importe quelle autre classe. Cette propriété est utilisée dans l'analyse des cas décrit plus tard.

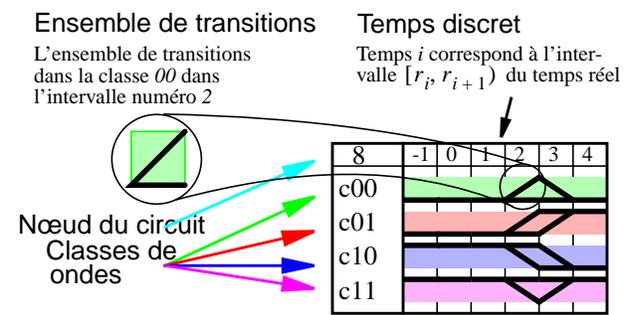


FIGURE 1: Onde abstraite

Chaque porte logique introduit des relations entre les ondes abstraites qui contraignent tous ses terminaux. La valeur de chaque nœud est exprimée comme l'intersection de la valeur du nœud courant et les fonctions de portes, dans les deux directions - sorties à entrée, et

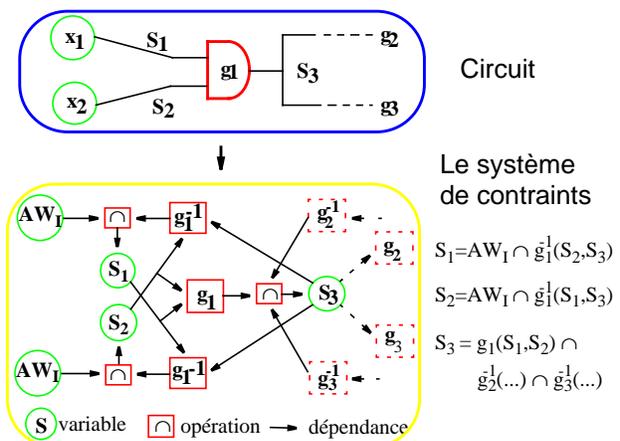


FIGURE 2: Le système de contraintes

entrée à sorties (Figure 2). Cette méthode permet de calculer rapidement une borne supérieure de l'activité de commutation.

Dans la thèse, la méthode est illustrée par un petit circuit - c17. Si le circuit est une partie combinatoire d'un circuit séquentiel, les entrées peuvent changer seulement par temps d'horloge (zero, observées les ondes sur nœuds 0 jusqu'à 4 dans Figure 3). Les autres ondes sont le résultat de propagation de ces contraintes (Figure 3). Dans cet exemple toutes les portes ont un délai unité.

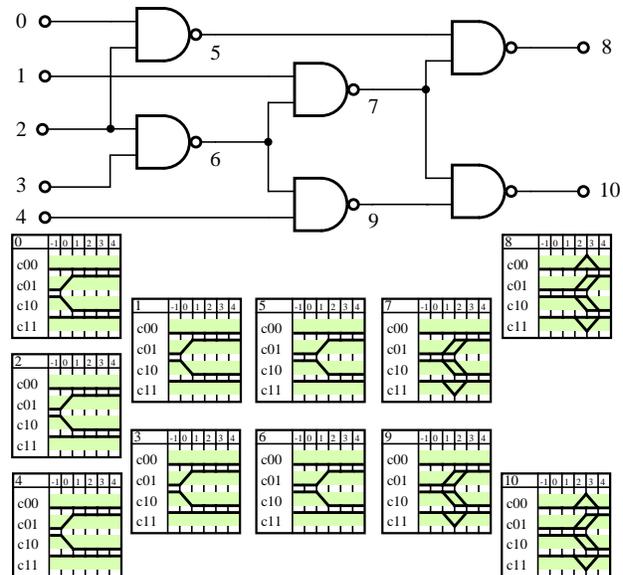


FIGURE 3: Circuit c17 - propagation nationale

Les ondes abstraites contiennent toutes les ondes réelles possibles. Donc, l'activité de commutation (nombre de transitions pendant une période d'horloge) est calculée comme la somme du nombre de transitions, pour chaque onde abstraite. Dans la Figure 3 c'est 15. En utilisant la librairie physique cette valeur peut être convertie à la consommation de courant.

Pour évaluer la fiabilité du circuit, il est important de calculer le courant maximal dans le réseau. Le courant maximal peut être calculé à partir du profil d'activité de commutation qui est obtenu en comptant les transitions de toutes les ondes pendant chaque intervalle de temps individuellement (Figure 4).

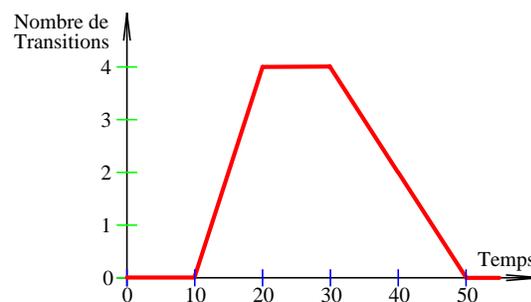
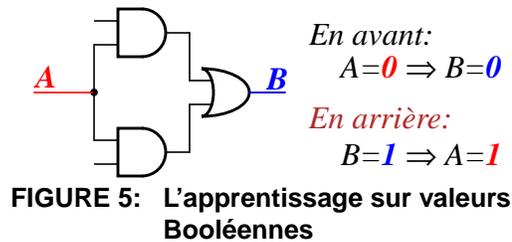


FIGURE 4: Circuit c17 - profil d'activité de commutation

Ensuite, la borne supérieure est améliorée par une analyse de cas basée sur l'indépendance des classes d'ondes dans les ondes abstraites. Dans cet exemple, l'analyse de cas a déterminé que la borne supérieure est 14 transitions pour une période d'horloge. Le nombre de transitions maximale pour un intervalle du temps n'était pas amélioré: la valeur initiale est la valeur exacte.

Le système de contraintes exprime seulement la relation locale entre des portes et nœuds du circuit. Deux autres techniques ont été introduites pour utiliser plus d'information sur le circuit:



l'analyse de régions de reconvergence

et l'apprentissage global (Figure 5). La méthode implémentée en C++ a été testée sur l'ensemble de circuits benchmark ISCAS'85 et l'efficacité de plusieurs heuristiques a été comparée. Les résultats ont démontrés que l'efficacité de chaque heuristique dépend de la topologie et de la fonctionnalité du circuit. La borne supérieure était de 1.5 à 3x la borne inférieure de l'activité de commutation (consommation) et 1.3 à 4x de l'activité de commutation maximale (courant maximal). D'autres propriétés de la méthode ont été testées sur les circuits MCNC et circuits ISCAS'85 avec délais de portes d'une librairie industrielle lsi_10k. Nous avons obtenu la valeur exacte¹ sur la majorité des circuits.

L'implémentation qui a été testée supporte des modèles de portes avec délais fixes et elle partage une grande partie du code avec une méthode de vérification temporelle basée sur la résolution de contraintes. La performance est ensuite améliorée par l'analyse de cas sur un réseau de stations de travail. La configuration parallèle est centralisée (étoile). La station au centre maintient l'espace de recherche, analyse, décide quelle parties sont intéressantes, et distribue les tâches aux autres (esclave). L'algorithme de répartition est résistant aux fautes sur les esclaves et la

1. "La valeur exacte" c'est à dire que la borne inférieurs obtenu par simulation est égale à la borne supérieure (sous les modèles de portes avec délais fixes).

station au centre préserve périodiquement son état sur un disque. Notre implémentation parallèle a démontré l'accélération d'un facteur 8 sur 10 stations de travail et 47 sur un réseau hétérogène de 87 stations.

Table of Contents

Abstract	iv
Résumé	vi
Table of Contents	xi
List of Tables	xviii
List of Figures	xxi
List of Symbols, Keywords, and Abbreviations	xxix
Preface	xxxv
<i>Chapter 1: Introduction</i>	1
1-1. Design flow	2
1-1.1 General design flow	2
1-1.2 Design flow in electronic industry	3
1-2. Verification of electronic circuits	6
1-2.1 Classification of verification methods	6
1-2.1.1 Functional verification	6
1-2.1.2 Testability	7
1-2.1.3 Timing verification	8
1-2.1.4 Verification of power consumption	8
1-2.1.5 Verification properties on a behavioral model	9
1-2.1.6 Verification of an RTL description	10
1-2.1.7 Verification at the gate level	11
1-2.1.8 Verification at the transistor level	12
1-2.1.9 Verification at the layout (mask) level	14
1-2.1.10 Verification by simulation	15
1-2.1.11 Formal verification	15
1-2.2 Timing verification	17
1-2.3 Verification of power consumption	20
1-3. The problem of timing and power verification and the contributions of this thesis	21
1-3.1 Problem definition	22
1-3.2 Outline of the proposed solution to bounding switching activity ...	23
1-3.3 Original contributions	24
1-3.4 Plan of the thesis	25

Chapter 2: Verification methods for timing and switching activity	27
2-1. Modeling	28
2-1.1 Timing and power properties of synchronous circuits	28
2-1.2 Delay representation	31
2-1.3 Gate delay models	32
2-1.4 Combinational circuit delay model	32
2-1.5 The false path problem	34
2-1.6 Spatial and temporal correlation	35
2-1.7 Component parameter correlation	36
2-1.8 Toggle power	38
2-1.9 Conversion from switching activity to power	38
2-2. Universal and timing-specific verification methods	40
2-2.1 Exhaustive simulation	40
2-2.2 Path-oriented methods	41
2-2.2.1 Static sensitization	42
2-2.2.2 Dynamic sensitization	43
2-2.2.3 Viability sensitization	44
2-2.2.4 Floating-mode sensitization	44
2-2.2.5 Lower bound sensitization	46
2-2.2.6 Vigorous sensitization	47
2-2.2.7 Comparison of several sensitization criteria	47
2-2.3 Algorithms for solving the false path problem	48
2-2.3.1 An algorithm for computing the longest viable path	48
2-2.3.2 An algorithm based on ATPG	50
2-2.4 Optimization-based methods	53
2-2.4.1 An algorithm for computing the exact circuit delay	53
2-2.4.2 An algorithm based on constraint satisfaction	54
2-2.4.3 Methods treating correlated component delays	54
2-2.5 Various problems and notes	57
2-2.5.1 Transition delay and the minimum clock period	57
2-2.5.2 Zero-width glitches	58
2-2.5.3 Standardization in delay and power calculation	58
2-3. Power verification methods	60
2-3.1 Random simulation	60
2-3.2 Probabilistic power estimation methods	61
2-3.2.1 Interval gate delays and power analysis	65
2-3.3 High-level approaches to power verification	68
2-3.3.1 High-level power analysis	68
2-3.3.2 Optimization of circuits for low power at the circuit level ..	70
2-3.3.3 Optimization of circuits for low power at the logic level ...	71
2-3.3.4 High level power optimization	73
2-3.3.5 System and software level power optimization	74
2-3.3.6 Synthesis for low power	74

2-3.4	Low-level approaches to power verification	75
2-3.5	Pattern independent methods for computing an upper bound on power dissipation	77
2-3.5.1	Uncertainty waveforms and partial input enumeration	77
2-3.5.2	Power estimation based on constraint resolution	78
2-3.6	Power estimation in sequential circuits	79
2-3.7	Methods analyzing current distributions on a chip	79
2-3.7.1	Voltage drops	79
2-3.7.2	Current distributions	80
2-4.	Literature survey - conclusions	81

Chapter 3: Bounding switching activity using constraint resolution 83

3-1.	Circuit modeling - the constraint system	83
3-1.1	Waveform classification	84
3-1.2	Operations over space of abstract waveforms	87
3-1.3	Power analysis	91
3-1.3.1	Initial propagation and estimate	92
3-1.3.2	Transition counting	92
3-2.	Case analysis	93
3-2.1	Principles of case analysis	93
3-2.2	Case analysis on waveform classes	94
3-2.3	Case analysis in a circuit	95
3-2.4	Decision tree	96
3-2.5	Case analysis algorithms	97
3-2.6	Net selection for case analysis	98
3-2.7	Properties of the constraint system and of the case analysis	99
3-3.	Example	100
3-4.	Reconvergence analysis	104
3-4.1	Introduction	104
3-4.1.1	Use of reconvergent regions in fault analysis	105
3-4.1.2	Complexity of fault analysis and region overlap	105
3-4.1.3	Secondary reconvergence	106
3-4.2	Fault analysis based on reconvergent regions	107
3-4.2.1	The fault model and dynamic reductions	108
3-4.2.2	Results	110
3-4.3	Reconvergence in timing and power verification	110
3-4.4	Algorithm for reconvergence analysis	112
3-5.	Learning	114
3-5.1	Global implications in ATPG	115
3-5.2	Use of global implications in the analysis of switching activity ..	116
3-6.	Heuristics for net selection in case analysis	118

3-6.1 Fanout	118
3-6.2 Fanout on primary inputs	119
3-6.3 Dynamically created list of nets	120
3-6.4 Reconvergent regions	121
3-6.5 Other heuristics for net selection in case analysis	121
3-7. Use of timing constraints	122
3-8. Implementation	122
<i>Chapter 4: Experimental Results</i>	123
4-1. Lower bound simulation	124
4-2. The initial upper bound on switching activity	126
4-3. Case analysis on nets sorted by decreasing fanout	127
4-3.1 FANOUT without learning for 1000 decisions	127
4-3.2 FANOUT without learning for 10000 decisions	131
4-3.3 FANOUT with learning for 1000 decisions	133
4-3.4 FANOUT with learning for 10000 decisions	134
4-4. Case analysis on primary inputs sorted by decreasing fanout	135
4-4.1 PIFAN without learning for 1000 decisions	135
4-4.2 PIFAN with learning for 1000 decisions	135
4-5. Case analysis on closing nets and primary inputs sorted by decreasing fanout	136
4-5.1 RCVFAN without learning for 1000 decisions	136
4-5.2 RCVFAN with learning for 1000 decisions	136
4-6. Comparison of heuristics for net selection	137
4-7. Comparison of PCA and HPCA	139
4-8. Reaching the exact value	141
4-9. Use of timing constraints	142
4-10. Comparison with other methods	144
<i>Chapter 5: Parallel Case Analysis</i>	145
5-1. Algorithm for parallel case analysis	145
5-1.1 Static search-space-division algorithm	146
5-1.2 Dynamic search-space-division algorithm	147
5-2. Example of Parallel Case Analysis	151
5-3. Implementation	153
5-3.1 The decision tree data structure	155
5-3.2 Implementation layers	157
5-3.3 Master	158
5-3.4 Decision tree checkpoints	158

5-3.5 Slave	159
5-3.6 Network layer	159
5-3.7 Object to message translation	159
5-3.8 MIF export interface	160
5-3.9 File descriptors	160
5-4. Experimental results	161
5-5. Performance measures for parallel processing	162
5-5.1 Analysis on a small number of computers	164
5-5.2 Analysis on a small number of computers for a constant time	168
5-5.3 Analysis on a large number of computers	169
5-5.4 Depth of local exploration	171
5-6. Theoretical model	173
5-6.1 Properties necessary for the application of our parallel algorithm	173
5-6.2 Speedup	174
5-6.3 Conditions for the best performance	176
5-6.4 Theoretical analysis of our practical results on 87 machines	176
5-7. Parallel implementation - conclusions	178
Chapter 6: Voltage Drops in Power Busses	179
6-1. Switching Activity Profile Algorithm	179
6-2. Example of Switching Activity Profile Calculation	181
6-3. Implementation of Switching Activity Profile Analysis	183
6-4. Experimental results	184
6-4.1 Peak switching activity in ISCAS'85 circuits	185
6-4.2 Peak switching activity in the MCNC circuits	189
Chapter 7: Conclusions	194
7-1. Constraint system	195
7-2. Case analysis	195
7-2.1 Initial upper bound on switching activity	195
7-2.2 Case analysis algorithms	196
7-2.3 Heuristics for net selection	196
7-2.4 Parallel case analysis	196
7-3. Comparison with other methods	197
7-4. Contributions of this thesis	199
7-5. Future research	199
7-5.1 Circuit models	200
7-5.2 Order of decisions during case analysis	200

7-5.3 Conversion of switching activity into electric current	200
7-5.4 Uses of parallel case analysis	200
Index	201
References	211
Appendices	ccxxviii
<i>Appendix A: Technical Documentation and Utilization</i>	ccxxviii
A-1. Architecture	ccxxviii
A-1.1 Modules	ccxxviii
A-1.2 Revision Control	ccxxix
A-1.3 Benchmarks	ccxxx
A-1.4 Source Code	ccxxx
A-2. Modules	ccxxxi
A-2.1 Parser interface and the top level (ic)	ccxxxii
A-2.1.1 Shared part	ccxxxii
A-2.1.2 Switching activity analysis (PW)	ccxxxiii
A-2.2 Circuit representation (udm)	ccxxxix
A-2.2.1 Constraint system fixpoint	ccxl
A-2.2.2 The data structure representing a circuit net	ccxl
A-2.2.3 The data structure representing a gate	ccxlii
A-2.2.4 The data structure representing a circuit	ccxliii
A-2.3 Timing (ta)	ccxlv
A-2.4 Switching activity (pw)	ccxlvi
A-2.5 Topological analysis (stat)	ccxlvii
A-2.6 Case analysis (anal)	ccxlvii
A-2.7 Partitioning (part)	ccxlviii
A-2.8 Learning (learn)	ccxlviii
A-2.9 Reconvergence (reconv)	ccxlix
A-2.10 Parallel (network)	ccl
A-2.11 Auxiliary (else)	ccl
A-3. Utilization	ccli
A-3.1 Viewing the circuit netlist	ccli
A-3.2 Timing analysis	cclii
A-3.3 Switching activity analysis	cclii
A-3.4 Decision function	cclii
A-3.5 Case analysis	ccliii
A-3.6 Parallel case analysis	ccliii
A-3.7 Manual mode	ccliv
A-3.8 Various options	ccliv
<i>Appendix B: Conflict of N messages</i>	cclvi
<i>Appendix C: Decision Trees</i>	cclvii

Acknowledgments cclxvi
Curriculum vitae cclxvii

List of Tables

TABLE I:	Classification of verification methods	6
TABLE II:	ISCAS'85 circuits [BrgF85]	124
TABLE III:	Lower bounds	125
TABLE IV:	Initial upper bound on switching activity	126
TABLE V:	Comparison of lower bounds and the initial upper bounds	127
TABLE VI:	Case analysis on all nets sorted by fanout, 1000 decisions	128
TABLE VII:	Case analysis on all nets sorted by fanout, 10000 decisions	131
TABLE VIII:	Case analysis on all nets, fanout, learning, 1000 decisions	133
TABLE IX:	Case analysis on all nets, fanout, learning, 10000 decisions	134
TABLE X:	Case analysis on PIs sorted by fanout, 1000 decisions	135
TABLE XI:	Case analysis on PIs, fanout, with learning, 1000 decisions	135
TABLE XII:	Case analysis on closing nets, 1000 decisions	136
TABLE XIII:	Case analysis on closing nets of reconvergent regions, with learning, 1000 decisions	137
TABLE XIV:	Comparison of heuristics for net selection	138
TABLE XV:	Comparison of PCA and HPCA, 11 ISCAS'85 circuits in total	139
TABLE XVI:	PCA, 1000 decisions	140
TABLE XVII:	HPCA, 1000 decisions	140
TABLE XVIII:	PCA, 10000 decisions	140
TABLE XIX:	HPCA, 10000 decisions	140
TABLE XX:	SAV in MCNC circuits, PIFAN, no_progress=1000	141
TABLE XXI:	Comparison of lower bounds and the final upper bounds	142
TABLE XXIII:	Case analysis with timing constraints	143

TABLE XXII: Topological delay and lower bound circuit delay	143
TABLE XXIV: Network media throughput	149
TABLE XXV: Calibration table c1908, 1002 decisions	164
TABLE XXVI: Comparison of SPECint92 with obtained performance	165
TABLE XXVII: Analysis results - circuit c1908, 1002 decisions, 1 to 10 computers in parallel	166
TABLE XXVIII: Analysis results - circuit c1908, 30 minutes, 1 to 10 computers in parallel	168
TABLE XXIX: Analysis results - circuit c1908, $3 \cdot 10^6$ decisions, 87 computers in parallel	170
TABLE XXX: Topological data for ISCAS85_lsi10k benchmarks	185
TABLE XXXI: Initial, and simulation switching activity for ISCAS85_lsi10k benchmarks	185
TABLE XXXII: PSAV (Peak Current) in ISCAS85_lsi10k circuits with no_progress=10, heuristic FANOUT	186
TABLE XXXIII: PSAV (Peak Current) in ISCAS85_lsi10k circuits with no_progress=100, heuristic FANOUT	186
TABLE XXXIV: PSAV (Peak Current) in ISCAS85_lsi10k circuits with no_progress=1000, heuristic FANOUT	187
TABLE XXXV: PSAV (Peak Current) in ISCAS85_lsi10k circuits with no_progress=1000, heuristic PIFAN	187
TABLE XXXVI: Comparison of our results with [KrNH95]; PSAV using FANOUT, no_progress=1000	188
TABLE XXXVII: Topological data for MCNC benchmarks	189
TABLE XXXVIII: Initial and simulation switching activity for MCNC benchmarks	189
TABLE XXXIX: PSAV (Peak Current) in MCNC circuits with no_progress=10.....	190
TABLE XL: PSAV (Peak Current) in MCNC circuits with no_progress=100	190
TABLE XLI: PSAV (Peak Current) in mcnc circuits with no_progress=1000	191

TABLE XLII: PSAV (Peak Current) in mnc circuits with no_progress= 10^6 ..	191
TABLE XLIII: PSAV (Peak Current) in MCNC circuits with no_progress= 10^6 , heuristic PIFAN	192

List of Figures

FIGURE 1:	Onde abstraite	vii
FIGURE 2:	Le système de constraints	vii
FIGURE 3:	Circuit c17 - propagation nationale	viii
FIGURE 4:	Circuit c17 - profile d'activité de commutation	viii
FIGURE 5:	L'apprentissage sur valeurs Booléennes	ix
FIGURE 6:	General Design flow	2
FIGURE 7:	Design flow	4
FIGURE 8:	Task of formal verification	7
FIGURE 9:	Behavioral description	9
FIGURE 10:	RTL description	10
FIGURE 11:	Gate-level description	11
FIGURE 12:	Transistor level description	12
FIGURE 13:	Layout description	14
FIGURE 14:	Verification by Simulation	15
FIGURE 15:	Formal verification	15
FIGURE 16:	Synchronous sequential circuit	28
FIGURE 17:	Timing properties of the synchronous circuit	28
FIGURE 19:	Setup and hold time considering clock distribution delay	29
FIGURE 18:	Clock distribution delay	29
FIGURE 20:	CMOS gate power consumption	30
FIGURE 21:	Switching behavior of a synchronous circuit	31
FIGURE 22:	Gate delay models	32
FIGURE 23:	Circuit delay models	33

FIGURE 25: False path demonstration	34
FIGURE 24: Path and side inputs to the path	34
FIGURE 26: Spatial correlation	35
FIGURE 27: Temporal correlation	36
FIGURE 28: Independent and correlated gate delays	37
FIGURE 29: Glitching	38
FIGURE 30: CMOS inverter	38
FIGURE 31: Static sensitization criterion	42
FIGURE 32: Dynamic sensitization criterion	43
FIGURE 33: Viability sensitization criterion	44
FIGURE 34: Floating-mode sensitization criterion	45
FIGURE 35: The difference between the viability and floating-mode sensitization criteria	46
FIGURE 36: Lower bound sensitization criterion	46
FIGURE 37: Vigorous sensitization criterion	47
FIGURE 38: Categorization of sensitization criteria	48
FIGURE 39: General verification algorithm handling a set of paths at a time ..	49
FIGURE 40: Making a circuit fanout tree	50
FIGURE 41: Two-level circuit representing ENF	50
FIGURE 42: Moving inverters to the primary inputs	52
FIGURE 43: Fault injection	52
FIGURE 44: Transition delay validity as a minimal clock period	58
FIGURE 45: Signal probabilities - AND and XOR gate	61
FIGURE 46: Glitching sensitivity of 2-input AND and XOR gates	62
FIGURE 47: Probability waveform	63
FIGURE 48: Signal representation by histogram of number of transitions	66
FIGURE 49: Transition density waveforms and the gate model	66

FIGURE 50: Less pessimistic transition density	67
FIGURE 51: Activity waveforms	68
FIGURE 52: Capacitance feed-through effect	76
FIGURE 53: 2-input CMOS NAND gate	76
FIGURE 54: An uncertainty waveform	77
FIGURE 55: VLSI power supply network	79
FIGURE 56: Voltage drops in a power network tree structure	80
FIGURE 57: Abstract waveform	84
FIGURE 58: Set of transitions	84
FIGURE 59: Waveform classification and abstraction	86
FIGURE 60: Graphical representation of an abstract waveform	86
FIGURE 61: Local intersection	88
FIGURE 62: AND operation	89
FIGURE 63: Example of NOT operation	89
FIGURE 64: Gate network as a system of equations	91
FIGURE 65: Primary input abstract waveform	92
FIGURE 66: Identification of simple waveform with the largest number of transitions in an abstract waveform	93
FIGURE 67: Principle of case analysis	94
FIGURE 68: Case analysis on a single net	94
FIGURE 69: Case analysis on two nets	95
FIGURE 70: Initial state of the constraint system	95
FIGURE 71: Constraint system for A=01	95
FIGURE 72: Constraint system for A=01, B=11	96
FIGURE 73: Case analysis decision tree	96
FIGURE 74: Step of the frontier-type case analysis	97
FIGURE 75: Counting fanout	98

FIGURE 76:	Circuit c17; all gates have unit delay.	100
FIGURE 77:	Waveforms of c17 after initial forward propagation	101
FIGURE 78:	Decision tree for c17, analysis on net 2	101
FIGURE 79:	Decision tree for c17, analysis on nets 2 and 6	102
FIGURE 80:	Waveforms of c17 for assignments 2:c01, 6:c10	102
FIGURE 81:	Decision tree for c17, complete analysis	103
FIGURE 82:	Simple waveforms with transition count of 14	103
FIGURE 83:	Reconvergent stem and reconvergence gate	104
FIGURE 84:	Closing gate	104
FIGURE 85:	Reconvergent stem region	105
FIGURE 86:	Primary (prg) and secondary (srg) reconvergence gates	106
FIGURE 87:	Enlarged reconvergence regions	107
FIGURE 88:	Disjoint subnetworks driven by exit lines	107
FIGURE 89:	Consequence of exit line properties	108
FIGURE 90:	Dropping faults, method "2"	109
FIGURE 91:	Case analysis on overlapping reconvergent stem regions	111
FIGURE 93:	Finding reconvergence gates - TAG1	112
FIGURE 92:	Finding reconvergence gates - TAG0	112
FIGURE 94:	Identifying reconvergent regions	113
FIGURE 95:	Forest of reconvergent stem regions in c432	114
FIGURE 96:	Implication and learning	114
FIGURE 97:	Learning - abstract waveforms	116
FIGURE 98:	Learning - constraint B=c11	116
FIGURE 99:	Learning - effect of the learned implication	116
FIGURE 100:	Learning - propagation due to a reduced side input	117
FIGURE 101:	Learning along reconvergent regions	117
FIGURE 102:	Fanout influence	118

FIGURE 103: Fanout influence - constraint $A=c00$	118
FIGURE 104: Fanout influence - constraint $B=c00$	119
FIGURE 105: Fanout influence - constraint $B=c00$ when $S1=c11$	119
FIGURE 106: Dynamically created list of nets	120
FIGURE 107: Simulation of ISCAS'85 circuits	124
FIGURE 108: Lower bounds on delay and switching activity in c432	125
FIGURE 109: Lower bounds on delay and switching activity in c499	125
FIGURE 110: Lower bounds on delay and switching activity in c1908	126
FIGURE 111: Lower bounds on delay and switching activity in c6288	126
FIGURE 112: Lower bounds on delay and switching activity in c7552	126
FIGURE 113: Progress of case analysis, c17, fanout, up to 1000 decisions	128
FIGURE 114: Progress of case analysis, c432, fanout, up to 1000 decisions ...	128
FIGURE 115: Progress of case analysis, c499, fanout, up to 1000 decisions ...	128
FIGURE 116: Progress of case analysis, c880, fanout, up to 1000 decisions ...	129
FIGURE 117: Progress of case analysis, c1350, fanout, up to 1000 decisions .	129
FIGURE 118: Progress of case analysis, c1908, fanout, up to 1000 decisions .	129
FIGURE 119: Progress of case analysis, c2670, fanout, up to 1000 decisions .	129
FIGURE 120: Progress of case analysis, c3540, fanout, up to 1000 decisions .	129
FIGURE 121: Progress of case analysis, c5315, fanout, up to 1000 decisions .	130
FIGURE 122: Progress of case analysis, c6288, fanout, up to 1000 decisions .	130
FIGURE 123: Progress of case analysis, c7552, fanout, up to 1000 decisions .	130
FIGURE 124: Progress of case analysis, c499, fanout, up to 10000 decisions .	131
FIGURE 125: Progress of case analysis, c1908, fanout, up to 10000 decisions	132
FIGURE 126: Progress of case analysis, c2670, fanout, up to 10000 decisions	132
FIGURE 127: Progress of case analysis, c6288, fanout, up to 10000 decisions	132

FIGURE 128: Progress of case analysis, c1355, fanout, learning, 1000 decis.	133
FIGURE 129: Progress of case analysis, c2670, fanout, learning, 10k decis. ..	134
FIGURE 130: Comparison of heuristics in c1908 - 1002 decisions, CPU time	138
FIGURE 131: Parallel case analysis with static division of the search space ...	146
FIGURE 132: Decision tree - c1908, 1000 decisions	147
FIGURE 133: Dynamic parallel case analysis	148
FIGURE 134: Fail-safe dynamic parallel case analysis	150
FIGURE 137: Parallel analysis - c17, second decision analysis	151
FIGURE 135: Network architecture - 2 slaves	151
FIGURE 136: Parallel analysis - c17, first 4 paths	151
FIGURE 138: Parallel analysis - c17, after the second decision analysis	152
FIGURE 139: Parallel analysis - c17, the third decision analysis	152
FIGURE 140: Parallel analysis - c17, the fourth decision analysis	152
FIGURE 141: Parallel analysis - c17, execution traces	153
FIGURE 142: CA tree as a heap	155
FIGURE 143: Path implementation	155
FIGURE 144: Heap architecture	156
FIGURE 145: C++ architecture of parallel case analysis	157
FIGURE 146: Upper bound as a functions of the number of decisions - c1908, 1002 decisions, 1, 6 and 10 CPUs	166
FIGURE 147: Upper bound in real time - circuit c1908, 1002 decisions, 1, 2, 4, 10 CPUs	167
FIGURE 148: Performance of parallel case analysis - decisions per second, c1908, 1002 decisions, 1 to 10 CPUs	167
FIGURE 149: Upper bound as a function of the number of slaves - c1908, 30 minutes, 1 to 10 CPUs	169
FIGURE 150: Performance of parallel case analysis as a function of the number of slaves - c1908, 30 minutes, 1 to 10 CPUs	169

FIGURE 151: Upper bound as a function of the number of decisions - c1908, 3*10 ⁶ decisions, 87 CPUs	170
FIGURE 152: Depth of local decisions	171
FIGURE 153: Comparison of 1- and 2-level local decision analysis - number of decisions, c1908, 10000 decisions, 6CPUs	171
FIGURE 155: Comparison of 1- and 2-level local decision analysis - real time, c1908, 2 hours, 6 CPUs	172
FIGURE 154: Comparison of 1- and 2-level local decision analysis - real time, c1908, 10000 decisions, 6 CPUs	172
FIGURE 156: Execution trace of parallel algorithm	174
FIGURE 157: Calculated speedup	177
FIGURE 159: Waveforms and the current profile of c17 after initial forward propagation	181
FIGURE 158: Circuit c17; all gates have unit delay.	181
FIGURE 160: Decision tree for c17	182
FIGURE 161: Test vector c17 (net4=c10)	182
FIGURE 162: Switching activity profile c17	183
FIGURE 163: Architecture	ccxxviii
FIGURE 164: Directory tree	ccxxx
FIGURE 165: Directory tree - source files	ccxxx
FIGURE 166: Circuit netlist and corresponding C++ classes	ccxxxix
FIGURE 168: Abstract waveform - UDMwaveformSet	ccxli
FIGURE 167: Net - UDMwaveforms	ccxli
FIGURE 169: Gate - UDMgate	ccxlii
FIGURE 170: UDMgatem	ccxliii
FIGURE 171: Circuit - UDM	ccxliv
FIGURE 172: Timing class - TAwaveform	ccxlv
FIGURE 173: Transition set class - PWwaveform	ccxlvi
FIGURE 174: Decision tree for c499	cclvii

FIGURE 175: Decision tree for c432	cclviii
FIGURE 176: Decision tree for c880	cclix
FIGURE 177: Decision tree for c1355	cclx
FIGURE 178: Decision tree for c1908	cclxi
FIGURE 179: Decision tree for c3540	cclxi
FIGURE 180: Decision tree for c2670	cclxii
FIGURE 181: Decision tree for c5315	cclxiii
FIGURE 182: Decision tree for c17	cclxiii
FIGURE 183: Decision tree for c6288	cclxiv
FIGURE 184: Decision tree for c7552	cclxv

List of Symbols, Keywords, and Abbreviations

- Arc** The concept of timing arcs is a way to describe pin-to-pin delays of a gate. A typical timing library would have at least one arc per pin pair. Each arc can describe rise/fall interval (min/max) delay or have a power consumption information associated with it.
- ASIC** Application specific integrated circuit
- BDD** Binary decision diagram
- BIST** Built-in self test
- bug** Error causing faulty behavior, in hardware or software
- CMOS** Complementary metal-oxide semiconductor - a widely used technology to implement low-cost, medium-speed, mainly digital electronic circuits
- C++** C Plus Plus - a portable object oriented programming language
- DCL** Delay Calculation Language

DRC	Design rule check - generally a term for post-layout verification of geometrical objects which implement the circuit in form of PCBs or ASICs
DSM	Deep Sub-Micron - related to design flow for fabrication with feature size under one micro meter, typically under 0.25 or 0.18 micro meter.
ECD	Expected Current Distribution - time profile of current drawn by a circuit with substantial statistical information for each interval of discrete time
EDA	Electronic Design Automation
FANOUT	Heuristic for selection of nets and their order for case analysis based on fanout count.
FF	Flip-flop
Flops	Floating-point operations per second - a very rough measure of computational performance
FPGA	Field-programmable gate-array - a low-cost alternative to ASICs, especially for prototyping
FSM	Finite state machine
GaAs	Gallium arsenide - semiconductor material used for high-speed or special components
HPCA	Heap path case analysis - the most advanced case analysis algorithm proposed in this thesis
HPGL	Hewlet-Packard Graphical Language - a vector image description language used mainly in plotters

- IC** Integrated circuit
- IC InCore** Verilog and VHDL parser and in-memory circuit representation database from Functionality, Inc., Ottawa
- MHz** 10^6 Hertz, SI derived unit of frequency, $\text{Hz} = \text{s}^{-1}$
- MIF** Maker Interchangeable Format - cross platform interchangeable format of a commercial publishing system (Adobe FrameMaker)
- NPC** Non-deterministic polynomial complete - a class of problems in classification by asymptotic complexity. A problem from this class can be solved by a non-deterministic computer in polynomial time
- NPH** Non-deterministic polynomial hard - a class of problems in classification by asymptotic complexity. In many cases in VLSI, an optimization version of a decision NPC problem is NPH.
- PCA** Path Case Analysis - intermediate level case analysis proposed in this thesis
- PCB** Printed Circuit Board - a commonly used way to implement circuit interconnects and mechanical integration for integrated circuits in packages
- PIFAN** Heuristic for selection of nets and their order for case analysis based on fanout count on primary inputs.
- PSAV** Peak Switching Activity value - maximum switching activity over observed time. It is used to compute peak power bus current.

- RMS** Root Mean Square current - a way to express consumed power by a single value of drawn current. In general, $RMS I = \frac{\int_0^T |\vec{i}(t) \times \vec{u}(t)| dt}{\int_0^T |\vec{u}(t)| dt}$, where u is voltage, i is current, and T is clock period (or any interval of time we want to observe)
- RTL** Register Transfer Level - a way how to describe functionality of systems or algorithms with some information on how they will be implemented
- SAT** Satisfiability problem - a general form of SAT is commonly used as canonical formulation of non-polynomial complete problems
- SCA** Stack-based case analysis - the oldest and least powerful case analysis proposed in this thesis
- SDF** Standard Delay Format
- SPF** Standard Parasitic Format
- Si** Silicon - chemical element, one of several natural semiconductors
- SPECint** System Performance Evaluation Cooperative's benchmark, <http://open.specbench.org/>
- 3-D** Three dimensional - describing things in space (natural to human understanding)
- 2-D, 2.5-D** Two dimensional - describing things as planar. The 2.5-D means some parameters from 3-D are added to a 2-D model
- VHDL** VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

VLSI Very Large Scale Integration

To all people whose ideas and hard work drive our world ahead.

Preface

There is no need to remind people how important part of our life electronics has become. We consider many electronic gadgets a normal part of our lives, from computers, cellular phones, precisely controlled car engines to the convenience of an airplane landing through dense fog instead of being diverted to an airport 200 miles away. To keep up with ever increasing demand, the complexity of electronic systems grows, therefore coming up with a new idea naturally brings up several essential questions: How much would it cost? Will it work after I manufacture what you propose? Is it going to be reliable? Is it not going to be obsolete when it is ready?

The multi-billion industry that is trying to give some answers to all these questions is called Electronic Design Automation (EDA). Its primary task is to make life of designers easier, more efficient and more productive. This is accomplished mainly through design and synthesis (transforming formally described ideas into feasible-to-manufacture physical components), layout, verification (testing the result before the product is physically manufactured), manufacturing and testing of physical systems. Additional tools are design flow managers, library tools, waveform browsers, integration frameworks, etc.

The recently published bugs in the most widely used complex consumer electronic products such as microprocessors for personal computers show that it is not easy to verify complex designs in a short time [Bur95]. Actually, verification together with behavioral synthesis and low power design became the challenges of EDA in the 1990-ties and well into the 21st century.

This thesis tries to touch a small portion of what is necessary to *verify* in an electronic design. It studies the use of a constraint-resolution approach in the verification of power consumption of one of the widely used classes of circuits called CMOS.

CHAPTER 1

Introduction

Despite the predictions from the early middle of this century suggesting that a few hundreds or thousands of computers would be sufficient to provide all the computing power needed all around the world, the computer industry is still expanding. There exists sustained desire for more powerful processors, larger memories, physically smaller and less power-hungry devices. Even more, availability of unprecedented computational power permitted one to think about the use of computers in areas and applications never imaginable before, resulting in a demand for resources exceeding the capability of current technologies by several orders of magnitude.

Nevertheless, there exist several technologies allowing non-human execution of algorithms. Possible implementations range from ancient hydromechanical machines, through recent mechanical calculators, to future optical, biological or even quantum computers. However, the microelectronic semiconductor technology seems to be dominant these days because of a lot of experience, ease of integration, reasonable scalability and reliability. Even if it does not seem so today, the electronic technology has physical limits and approaches them fast. E.g., a chip of dimensions of 1 inch across containing a simple combinational logic circuit cannot operate at a frequency above 12GHz, because of the speed of propagation of signals which is inferior to the speed of light. It can be improved by smart organization, but the limits of the current technologies are even lower than that imposed by

the speed of propagation, hence we cannot expect digital electronic processors running at more than a few tens of GHz soon.

Some other technologies may take over in the future, but considering also the huge investments in microelectronic VLSI technologies, such a transition will neither be soon nor fast. Thus in parallel with development of new methods and technologies we are improving the existing ones. The need of higher speeds and more functionality is satisfied by further integration, pushing the technologies to the edge. The higher the integration and the smaller the dimensions, the less predictable and more costly the designs are, thus requiring more accurate pre-manufacturing verification. The following section explains what processes are involved in achieving the best performance in the shortest possible time with the least cost.

1-1. Design flow

Design flow is a sequence of processes involved in obtaining a correct high-performance system for a given specification quickly and with minimal cost.

1-1.1 General design flow

Building an electronic or any other system requires in general at least three steps (Figure 6). *Specification* defines what the system is supposed to do, what the inputs and outputs are. *Realization* means designing an equivalent physical system which performs the same or an acceptably similar function. *Test* is a process of verifying whether the actual product really does what is stated in the specification.

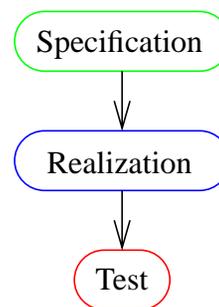


FIGURE 6: General Design flow

Such a design flow is more than sufficient for a simple system. For example, building a home library would involve specification of how many books and of what dimensions it should hold, cutting and gluing lumber to implement it as a physical wooden bookshelf and putting books there to test whether they really fit in. How are we sure the shelf would not collapse? The answer is we are NOT sure at all. We only hope (and know by experience) that the lumber and glue used are many times stronger than needed to hold the weight of the books.

1-1.2 Design flow in electronic industry

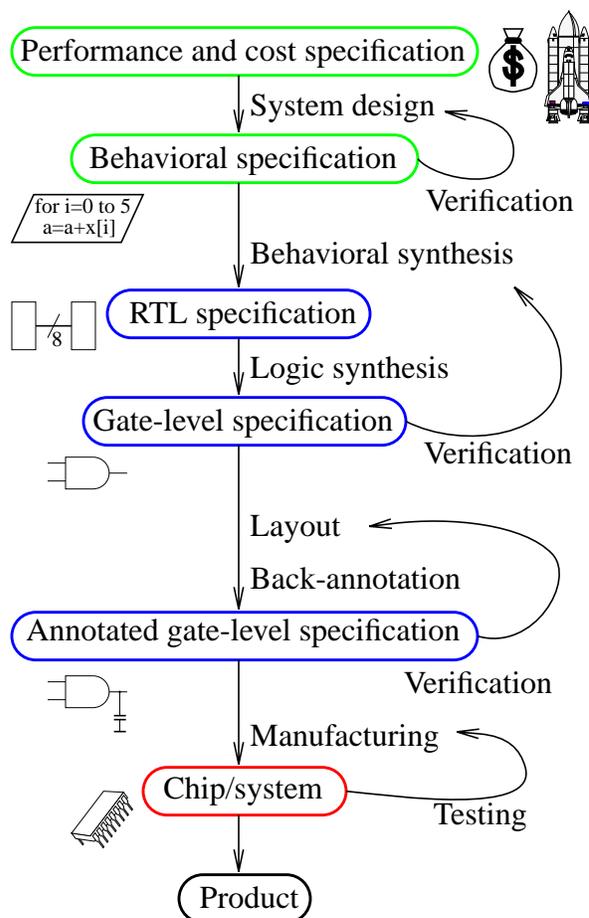
Electronics systems are much more complicated these days. No single person during his/her lifetime could retrace the true functionality of a contemporary microprocessor specified as a set of Boolean equations. The whole design process does not come for free either, but rather costs considerable amounts of money. In the market driven economy, the turnaround time of the design cycle from a specification to the final product (called *time to market*) is essential too. All that defines what a good design process is: producing the best system in terms of performance and features at minimal cost and in the shortest possible time.

A modern design process (Figure 7) of a digital electronic system follows a more complicated design flow than the one in Figure 6. The specification is done at several levels: *performance and cost specification* saying what a customer wants and can afford, *behavioral specification* usually in some high-level specification languages describing the desired behavior (algorithm) of the system. Then comes *validation* of the specification. Usually it is very difficult to verify performance and cost, but at least some properties of the functionality can be checked. Then the architecture and the interfaces are defined. The behavioral specification is transformed into a feasible-to-manufacture specification by synthesis, either manual, or partially or fully automated.

The current state-of-the-art automated synthesis proceeds in two steps, because algorithms and computational power which could do it in one step are not available. First the behavioral specification is transformed into a register transfer level specification (*RTL*) in a process called *behavioral synthesis*. Then this RTL model is transformed into a *gate-level specification*. This process is called *logic synthesis*.

Excluding possible implementations of our original specification in terms of a mechanical, analog, or in the future optical or even quantum systems, there are many gate-level systems satisfying the behavioral specification. Which one to choose is a sub-task of synthesis called *design-space exploration*. Unless an ideal synthesis tool is used, the gate-level description must be verified against the customer's or at least the behavioral specification. Once we have a satisfactory gate-level specification expressed as an oriented multi-graph (nodes are gates, edges are interconnections, the orientation is input to output), it is transformed into a multi-level

planar graph in a process called *layout*. Library cells are placed on what will become an IC chip, and interconnections are routed. The layout, also called *place & route* task, must produce something that chip manufacturers (called *foundries*) can physically produce at a price acceptable to the customer. Apart from *fea-*



If all tools were ideal, design would be a straightforward, not iterative process. Bugs found during verification return us one or more levels up.

FIGURE 7: Design flow

ture-line width there are many other limits on the geometry of a physical design such as the distances between wires, the shape of corners, etc., called *design rules*. Layout is always driven by design rules and a design rule check (*DRC*) is done after layout. Then the physical specification is verified as an analog circuit or more frequently as an *annotated gate-level* design. At this point in the design flow we can for the first time accurately verify timing (i.e., performance) and the consumed power. The foundry manufactures the circuits producing silicon, GaAs or other semiconductor chips, or multi-chip modules. Those are tested for structural defects relative to the gate-level netlist (structural faults) rather than for its function which is too costly.

The design flow is an iterative process. If verification at any stage identifies an error (a difference between the current and the higher-level specification), the designer or the automated tool returns to the previous step and chooses a different solution in the design space exploration. If no solution leading to a satisfiable design is found, it simply means that the specification is incompatible with the current technology and algorithms. For example, an 80-bit floating point unit with throughput of 1TFlops on 0.01 mm^2 of silicon and under \$1M is impossible now, but might be easy in 10 years. The problem of convergence of the design flow has become a hot topic in 1990-ties. The current logic delay-centric design flow needs too many iterations in deep submicron design (DSM) where interconnect delays are dominant over logic delays. Similar problems exists with power consumptions. The future probably belongs to synthesis tools that consider physical implementation early in the design flow and optimize for delay, power, and area at the same time [BrOt98].

We shall be mainly concerned with synchronous digital CMOS circuits. The reason is that it is the prevailing technology today and in the near future. Our models are intended to be generic, but when technology details are introduced, then CMOS is assumed. In the next section we look in detail at what verification is, and what the problems and the possible solutions are.

1-2. Verification of electronic circuits

Verification is a process of confirming certain properties on the subject of the investigation, a circuit in our case. In this section we see how verification can be classified and we look in detail at functional, timing, and power consumption verification.

1-2.1 Classification of verification methods

Verification methods for electronic digital circuits can be classified either by the properties they check, or by the circuit description they work with, or by the approach they take, as summarized in Table I.

TABLE I: Classification of verification methods

	Criteria of classification		
	Property to verify	Level of circuit description	Approach
Possibilities according to the classification criterion	Function	Behavioral	Simulation
	Testability	RTL	Formal
	Timing	Gate	
	Consumed power	Switch	
		Transistor	

1-2.1.1 Functional verification

Functional verification checks the function of the design. For example, a multiplier should perform function $c=a*b$ for any allowed values of inputs a and b . Note that it is not really important what the design is, whether a C-language code, an RTL structure, transistor-level description, a chip sitting on a testbench, or a human with a pencil and a piece of paper. The design is correct, if it is doing what the customer wants. But how can one specify what the customer wants? Usually by a

high-level description, such as our behavioral $c=a*b$ or few test cases, e.g. “please, design a multiplier that computes $2*3=6$ and $6*8=48$, I don’t care about the rest.”

Design is usually done in hierarchical steps from the behavioral specification down to a network of transistors. Verification can thus be done in each step too to discover problems as early as possible.

A design is considered correct, if the new synthesized lower-level specification *implies* the higher-level one.

The “implication” is used in the sense “[the design] does everything and maybe more”.

That “more” are the internal signals, intermediate states, etc. The implication is transitive, so the chain of implications from the transistor down to the behavioral level is the same as a direct implication. That would be the ideal case. In reality, many specifications are not complete [CerK95], making checking whether one design completely implies another one impossible. The methods of functional verification are discussed in detail in Section 1-2.1.11 on page 15.

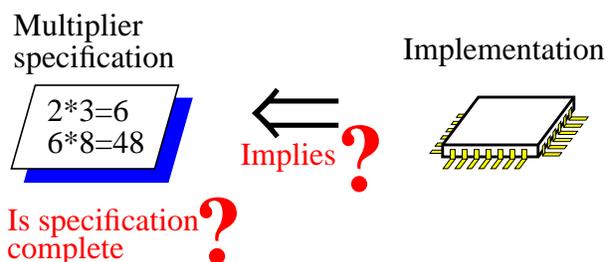


FIGURE 8: Task of formal verification

1-2.1.2 Testability

Once a circuit is physically manufactured, it must be tested. Testing can be very costly. An average ASIC/FPGA/memory tester costs around \$1M (or \$5k per channel), plus the cost of developing testbench programs, and the cost of the time spent by testing. Also, some circuits are easier to test than others. To save money, to increase reliability, and to speed up testing, designers employ *design for testability (DFT)*. It is a collection of design methods, which either add some circuitry (such as *BIST*, build-in self test), or use different implementations such as ones not having undetectable faults. Testability is beyond the scope of this thesis, however.

1-2.1.3 Timing verification

The performance of electronic products is often the major selling point. People upgrade computers after only few years of service, doubling the performance. Anybody can say that it is the new features and the functionality of the many software products that make it sell - but the features are enabled by the higher performance in all senses of that word. One of them is the raw speed expressed in either MHz, MIPS, SPECint or some better application-specific units.

Verification of timing properties of digital circuits is necessary either to find out whether a circuit meets the timing specification (like the system's clock frequency) or to inquire about the current performance of the system (the maximum clock frequency that may be applied). Timing verification is introduced in detail in Section 1-2.2 on page 17.

1-2.1.4 Verification of power consumption

With the diminishing physical dimensions and the need for portable electronic products with long battery life, the amount of power consumed by the products is more and more important. The average power drawn by circuits determines the duration of independent stand-alone use, or the dimensions and the weight of the batteries.

Another reason for the verification of power consumption is the functional correctness of the device with respect to the used technologies. The average power (to be defined later) consumed by a circuit should not exceed the maximum power the package is capable to dissipate, otherwise there is a danger of overheating. High peak power can cause voltage drops in power supply lines, leaving the circuit as if the power supply failed. Verification of power consumption is discussed in detail in Section 1-2.3 on page 20.

Next we shall see how different is the verification of a circuit described at the behavioral-level, RTL, gate or transistor level. Each has some advantages as well as disadvantages.

1-2.1.5 Verification properties on a behavioral model

A behavioral model of a circuit or a system is nothing but pure functionality, sometimes augmented with basic system requirements such as overall speed, power consumption and cost. Such a description is usually written in a formal language

```

for (i=0; i<10; i++)
k[i+1] = k[i]*1 + v
...

```

FIGURE 9: Behavioral description

(like that shown in Figure 9) based on the original description in plain English. E.g., “I want a SCSI-III interface with tag queue size of 32” means for a system designer to read through the SCSI-III standard definition and capture it in a formal language such as C++ or VHDL. There is no way to check whether he/she captured it correctly, but he/she can still verify many properties of his/her behavioral model¹. It can be, e.g., that no SCSI command is lost, or that the controller performs basic operations when connected to a behavioral model of a disk and that of a computer. The latter approach of trying few possible cases is called simulation [SaVi81, Pede84], the former (proving that no command can ever be lost) may be easier to achieve by formal methods [Schr97], we shall discuss both in Sections 1-2.1.10 and 1-2.1.11. No matter what approach is taken, the functionality can be checked using a behavioral description of the system.

Functionality of smaller blocks is easier to verify, but one should not forget to verify the compatibility of interfaces between partitions, compliance of behavior of each partition with the interface specification, compatibility of interfaces each with other, as well as realizability of interface specification [CerK95].

1. *Model versus description*: generally when talking about manufacturing, “description” is what specifies what to manufacture; “model” is referred to what is necessary to verify (simulate) what will be manufactured based on the “description.” In a modern design flow, verification and manufacturing are closely coupled; therefore, “description” and “model” are considered synonymous in this thesis.

Functionality can be checked at the behavioral level. How about performance and consumed power? Some estimates¹ are possible, but more often they impose constraints on the partitions of the system which are passed to behavioral synthesis. They can be left blank, i.e., we build a system and then see how good it is. Such an approach was common and acceptable in the early years of electronic design but not any more.

1-2.1.6 Verification of an RTL description

Behavioral synthesis, either manual or automated, translates a behavioral description into an RTL description under timing, power, and cost (area) constraints. An RTL description is similar to a behavioral one in the sense that it operates with

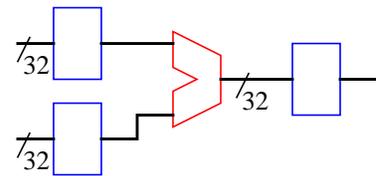


FIGURE 10: RTL description

objects at a higher level of abstraction, such as integers, enumerated types, sometimes floating-point numbers, and Booleans. However, the function is described by register transfers as the acronym RTL indicates. Each data type is assigned a fixed length in bits; the width of the data-paths, pipelines, how many functional units and how interconnected they are, all that is known. An example of RTL description is shown in Figure 10.

Functionality of an RTL description (model) is verified by either checking some properties of individual cases, or by checking that the whole RTL implies the behavioral model. E.g., the behavior of a multiplier is $c=a*b$, its implementation can be a 4-stage pipeline with 1000 bits of registers. We have to show that no matter in what order and what numbers are fed into the multiplier, it still holds that 4 clock cycles after the inputs a, b are entered at time t the result is $a*b$, i.e., $c[t+4]=a[t]*b[t]$. But it is not the same as the un-timed model $c=a*b$, which we

1. *Verification versus estimates*: it will be seen later in Section 2-2 on page 40 that some verification methods are capable of computing estimates, others are exact; estimates are easier to obtain, so in many parts of this introduction we encounter sentences like “at the gate level, estimation of power consumption is possible”

read as $c[t]=a[t]*b[t]!$ Yes, they are different, it actually means that also the other components (e.g., the unit that generates data for the multiplier, or the unit that uses its output) must be pipelined. If they are not or are pipelined with a different number of stages, a stall logic or buffers must be added ... and verified. How to verify such additions (called *glue logic*) precisely without building an RTL model of the whole system is beyond the scope of this thesis as well as the knowledge of the author.

An RTL model defines what data, how, and when moves around in the circuit. Having a good library of generic RTL building blocks with estimates on delay and consumed power, one can make very good predictions or optimizations based on performance (system clock), cost (number of units, i.e., area), and recently also consumed power. Yet, an RTL model alone is not sufficient to estimate consumed power as will be seen later in Section 1-2.3 on page 20, some information on input data is needed too. After design space exploration, the most promising RTL design is the input to logic synthesis. Logic synthesis produces a gate level description, which allows us to verify many interesting properties as outlined in the following section.

1-2.1.7 Verification at the gate level

Logic synthesis translates an RTL description into a gate-level description (model). Generic components such as adders of an arbitrary width are expressed as networks of gates (Figure 11). Gate-level models reflect

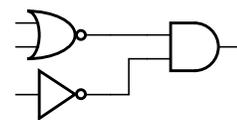


FIGURE 11: Gate-level description

exactly the Boolean equations that implement each synthesized block. Functionality may again be verified against a higher-level description. But the gate level representation is more accurate for computing test vectors (and eventually fault coverage), timing errors, cost, and power consumption estimation.

Since gates are generally the smallest units (*cells*) in the library, the timing information and other electrical characteristics can be precisely captured. Knowing how

a gate is interconnected with other gates allows to estimate the propagation time through the gate and the slew degradation on output signals. The delays on interconnections are not known yet, however, because the length and the spatial distribution of interconnections are not known. Till about early 1990s, the *interconnection delays* were neglected. Now, with higher integration (smaller gates, but more of them) the interconnect delay is changing from nearly negligible to the dominant delay in a circuit.

Power consumption can be estimated quite accurately at the gate level. As will be explained later in Section 2-1.9 on page 38, transitions, i.e., changes in the Boolean value of a signal, are the major cause of power consumption in CMOS gates. Knowing the input signals, their timing, and the paths along which the input transition will propagate, it is possible to estimate the power consumption reasonably well. More accurate estimates are possible at the transistor level as shown in the following two sections.

1-2.1.8 Verification at the transistor level

Technology mapping is a process of mapping a gate level description onto a particular technology library which is usually specific to each foundry. Many large design companies build their own libraries to reduce the dependence on foundries. The result of technology mapping is a transistor-level description. A transistor-level model views digital gates as an analog circuit, e.g., the circuit in

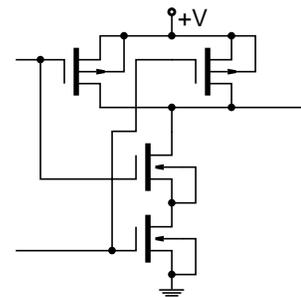


FIGURE 12: Transistor level

Figure 12 is a two-input CMOS technology AND gate. Analysis of the analog circuit can discover many problems in driving strength, timing, parasitic capacitance, resistance, and inductance. Yet, when working with some synthesis tools, there is no visible transistor-level description. The reason is that the RTL is mapped onto individual cells, that have functionality of less than one, one, or more gates. The foundry (owner of the library) provides only the names of the cells and many

parameters, such as area, timing information, power consumption indication, etc. The cell implementation is not public, neither is its layout (mask description).

Individual cells are always analyzed at the transistor level to assure functionality and to extract timing, power, and other electrical parameters. The process is called *cell characterization*, see e.g., [ChoS95]. The extracted parameters are exactly what the library vendor provides to designers and some more proprietary data such as the variations of the parameters due to the imperfections in the manufacturing process. Whenever needed the whole circuit or a block may have to be verified at the transistor level. This is the case when the cell library is inaccurate for the clock frequency used, incompletely characterized, or a special design is used, such as *dynamic logic*. (Any combinational transistor level circuit shown in this thesis will be *static CMOS logic* - it does not require any clock signal for its operation unlike dynamic logic [GoMa83]).

The enormous size of a transistor-level description for even a small circuit encourages the development of hybrid models - close to transistor-level in accuracy, yet benefiting from the gate-level model simplicity. Therefore, switch-level models and piece-wise linear signal representations have been introduced [Bryb87]. We can see the same phenomenon happening in verification at every level of abstraction - the engineers want to get the accuracy of lower levels with the lower complexity of the higher levels of abstraction.

What is missing for accurate timing and power consumption verification at the transistor level is the knowledge of interconnects, and thus the incurred capacitive load. Since the cell placement is not known, the crosstalk (coupling) between interconnections is not known either. Excluding the mask library for the individual cells, many electrical parameters become known only after layout, which we discuss next.

1-2.1.9 Verification at the layout (mask) level

The task of layout is to implement the transistor level description of a circuit as geometrical objects on a semiconductor die (Figure 13).

That makes the model the most accurate but also the most complex. Most major EDA companies now extract 2-D geometrical model for deriving parasitic capacitances and resistances. E.g., IBM [Hutt97] found out that in CMOS technology processes with the feature line

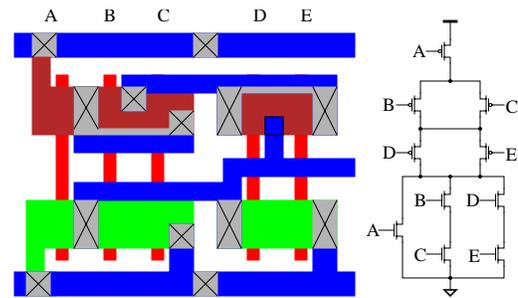


FIGURE 13: Layout description

width under $0.25 \mu\text{m}$, inductance becomes also important. Therefore, they are exploring 2.5-D models. Full 3-D is currently (as of early 1997) too costly, but some methods are starting to appear [DaSu97].

Functionality is rarely checked at the transistor level for common circuits. However, transistor models can be derived from masks and then compared with the transistor (or even higher-level) specification to verify the correctness of the layout tool. Timing can be checked very accurately [Syna97]. Power consumption too because resistances and capacitive loads due to interconnects are known.

Is layout the lowest abstraction level? Apparently not. The transistors are built of physical materials, which consist of molecules, molecules of atoms, atoms of quarks, Some tests at molecular level have been performed with alpha-particles interacting with molecules in the transistors of a memory cell [ZCMM96]. More verification below the transistor level can be expected in the future, especially if technology changes from electronic to optical or biological.

At the various levels of circuit description, two major approaches to verification can be taken: “evaluate the circuit model for a few important cases and hope the rest are ok” which is called simulation or methods based on mathematical defini-

tions, called formal methods. The major differences between the two approaches are the subject of the following two sections.

1-2.1.10 Verification by simulation

Simulation is a process of comparing circuit responses to inputs with the expected responses (Figure 14). It is fairly easy accomplished by evaluat-

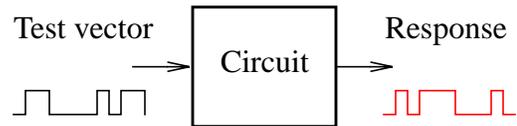


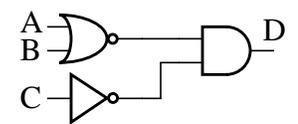
FIGURE 14: Verification by Simulation

ing the circuit simulation model for given inputs. Simulation models can be implemented in many ways. The most common implementation is a procedure written in C or assembler language that matches the API of the simulation tool. One circuit is typically described by a different model at each level of abstraction. There are many cases where simulation is the only feasible verification method. We shall find out more about the limitations and advantages of simulation in Section 2-2.1 on page 40. Currently (late 1998), simulation coexists with formal methods rather than being pushed out of the market. Formal methods are described in the next section.

1-2.1.11 Formal verification

Unlike simulation picking up few samples from the space of possible circuit inputs, *formal verification* deals with the whole space. Formal verification can be applied at many levels of abstraction, but even the state of the art implementations are more limited in capacity (number of gates, blocks) and speed than simulation. It seems that the right approach is to combine several formal methods [Sync98].

Formal methods exist for verification of functionality and timing, as well as power consumption. Generally,



⇓ Implies?

```
{
  if (a) return 0;
  if (b) return 0;
  if (c) return 0;
  return 1;
}
```

FIGURE 15: Formal verification

the term *formal verification* means “functional formal verification” (e.g., Figure 15). Timing and power formal verification methods are called *static methods*. The seemingly confusing term “a static method for verification of dynamic power in CMOS circuits” should be understood as a “static” (read formal) method for the verification of a circuit property called “dynamic power”. To estimate a property such as dynamic power (more in Section 2-1.9 on page 38), simulation would run several sample input patterns (called test vectors, or input vectors), measure the dynamic power and hope that the test vectors were typical. A static method does not require any hope. It computes the worst case. Moreover, it does it with complexity comparable to simulating only one test vector¹. **Most of this thesis is about a static verification method for dynamic power in CMOS circuits.**

Functional formal methods are of two major types: model checking and theorem proving. *Model checking* enumerates in an intelligent way all states of the sequential circuit (for definition see Section 2-1.1 on page 28). Its major problem is the enormous number of circuit states that must be verified, a problem called *state explosion*. Note that the number of states in which a binary system can be is two to the power of the number of the memory bits (a microprocessor with 10000 bits of memory in registers, pipeline buffers, etc., has potentially² 2^{10000} states). Therefore, a considerable effort is devoted to the state representation and related search heuristics. Among the popular applications is comparing a gate-level design to the behavioral specification or comparing two gate-level designs, e.g., an old reliable one with a new, modified one. This is often called *equivalence checking*. The term “model checking” itself is usually understood as checking of properties, such as, e.g., the output signals *ERROR* and *OK* in our design are never true at the same time. Equivalence checking is a special case of model checking³. Model checking verification tools are available on the market right now [Schr97, Youn96]

-
1. This is at the expense of accuracy. The static methods usually provide upper bounds (e.g., on delay, power consumptions). For best accuracy these methods become NPH; simulation is exponential in the number of gates (fixed gate delays) or impossible (interval gate delays).
 2. Not all states are reachable.

(*FormalCheck* from Lucent Technologies, *Design VERIFYer 2* and *Design INSIGHT* from Chrysalis, and several other tools from Abstract, Cadence, Synopsys, and many startups).

Model checking is the brute-force approach among formal methods. Generally it can find an error if there is any, but requires too many resources to complete the space search to give the answer “the property is satisfied.” It is relatively easy to implement, does not require experts¹ to run it and the more memory and CPU time it has, the better results it can produce. *Theorem proving* requires more user expertise and for some tasks needs far less CPU time. It tries to prove certain properties of signals or even that a design implies another one by mathematical theorem proving methods. There are several experimental systems developed at universities, e.g., [GoMe93], [Rues96]. One industrial player was Lambda from Abstract², Inc., [Youn96], and other companies probably attempt to use formal verification as well. The current state is that still too much user expertise is required to apply theorem proving in an industrial environment. We now examine the related subject of timing verification in the next section.

1-2.2 Timing verification

In this section we explain what is timing verification, where is it used in a modern design flow and what exactly is being verified.

Timing verification is a process of assuring that no timing constraint in the design is violated. A *timing constraint* is a subset of possible timing relations between signals and their events in the circuit, e.g., the data arrives on a flip-flop before the active clock edge minus the setup time. Satisfaction of timing constraints is

-
3. Connecting the inputs of the two circuits in parallel and adding comparators on the outputs, equivalence checking is the same as checking the property that the output of the comparator is always true.
 1. In comparison with theorem proving.
 2. Despite of small investments by Intel and Viewlogic (Synopsys) Abstract closed down in late 1998 because of insufficient revenue from its products.

required to ensure correct functionality and performance of the design. Using a design as a block in another design defines the boundary environment on inputs (timing on and drive strengths to the inputs of our design) and outputs (timing constraints and load on the outputs of our design). A timing verification tool analyzes these constraints, the circuit netlist, library cell, and additional information (like parasitics, load models, user-specified constraints or user-disabled constraints) to produce a list of timing violations. Modern timing tools work closely with synthesis tools, providing features like “how would the timing change if this cell were replaced by another one” [Synb97]. Each of the inputs to timing verification tools is equally important. University and research tools focus on new verification algorithms using a limited set of simple libraries, while commercial tools have strong library support and use stable, proven algorithms.

Timing verification can be done by either simulation or by static methods. It can be done at several levels of abstraction. In a typical design flow, the behavioral specification is verified for function. The behavioral-level timing models are quite inaccurate unless they were derived from an existing gate level model. Such models are used for design planning (deciding the location of every block on a chip) or budgeting (deciding the distribution of delays among all the blocks). RTL designs are verified for timing using library component delays. The timing information is further refined at the gate level by obtaining more accurate timing information about individual gates and cells. The most accurate verification can be done after layout when the interconnect delays are known, either at the transistor level or more often using a gate-level netlist annotated with delays extracted from the layout.

Both simulation and static methods can be used. The general idea applies to timing as well - simulation tests only few cases which are more or less representative; more sophisticated static methods test everything, but complexity might force them to be conservative (i.e., reporting worse delay than the real circuit would have).

The problem is to decide whether a given system meets the required timing specification (such as clock frequency) under all the operating modes and conditions. These include all possible inputs or input sequences, fabrication inaccuracy of components, etc. We are concerned here only with synchronous sequential circuits, because it is the most common way of implementing electronic systems today. The main difference between *synchronous* and *asynchronous* circuits is that in synchronous circuits signals may change only at precisely specified intervals derived from special signals called *clocks*. Synchronous circuits are smaller, easier to design and test while asynchronous are faster and consume less power for the same function.

A synchronous sequential circuit is an implementation of a finite state machine (FSM). It consists of a *combinational part* implementing the state and output transition functions and *flip-flops* (FF) implementing the state variables in a binary encoding. A flip-flop is a memory element which copies its input called *Data* (D) into its internal state on the active clock edge (input signal C). The internal state is viewed outside on the output called Q . For a given performance, the clock period is fixed. To assure that a synchronous sequential circuit works properly one must satisfy *setup* and *hold* time constraints on the inputs of the flip-flops. These depend on mutual timing relations between the arrival times of events on D and the events on C of FFs (the exact definition is given in Section 2-1.1 on page 28). The following issues are essential and must be considered while verifying the timing constraints:

- Circuit netlist,
- Library components (timing arcs, function, pin capacitances, ...),
- Gate delay model (interval gate delays, different for rising and falling transitions),
- Clock signal (skew, distribution line delays),
- Signal transitions on primary inputs,
- Component delay correlation, and

- Interconnect delays and capacitive load.

Because timing and power verification have much in common, an overview of existing timing verification methods is given in Section 2-2 on page 40. We briefly introduce verification of power consumption in the next section.

1-2.3 Verification of power consumption

In this section we show what is verified and when verification of power consumption is used in a modern design flow. Many terms used in this introduction are defined exactly in Chapter 2, Section 2-3 on page 60.

Circuits implemented in CMOS technology draw considerably higher amounts of current when switching (Section 2-1.9 on page 38). Therefore the power consumption of a circuit is closely related to its *switching activity*. Switching activity is the number of transitions in the circuit over certain period of time. It depends on circuit function, topology, and also on the signals on the circuit inputs.

Peak power (highest power consumption at any time) causes pulses in power buses leading to *voltage drops* and thus to malfunction of sub-circuits powered by them. *Average power* (e.g., per clock period) tells how much heat a package should dissipate to avoid overheating the chip inside the package.

In a similar way as in timing verification, power consumption is supplied as a synthesis constraint at behavioral level. Some methods for power verification at the behavioral level started to appear, more about it in Section 2-3.3 on page 68. At RTL, consumed power can be estimated more accurately. Knowing what happens on inputs to each RTL component, and how much power it would consume for certain signals, one can make estimates. How to estimate the power caused by a bus when we do not know which signals in the bus are switching? The answer is not easy but there exist methods addressing this problem, such as [LanR95]. But the mainstream and most

accurate power consumption verification is still done at the gate level. Knowing how many individual transitions on signals and when they occur in a circuit gives a very strong ground for estimating consumed power, both peak and average. Apparently, the most accurate is the transistor level model, where the consumed power is simply the product of the consumed instantaneous current and the power supply voltage. That allows for simple power/current profiling (power plotted against time). High complexity generally limits this approach to individual cells and is usually input-pattern dependent.

The most important issues which must be considered in verification of power consumption at the gate level are the following:

- Timing and functional properties (as mentioned in Section 1-2.2 on page 17 about timing verification),
- Transistor-level gate models.

In the next section we explain the ultimate goals of timing and power verification and summarize the contribution of this thesis.

1-3. The problem of timing and power verification and the contributions of this thesis

The ideal result of a verification procedure is a statement like “under all operating conditions the device can run with the clock frequency of nn Hertz, an additional constraint on timing of primary inputs is ..., it will consume less than mm watts, the peak power distribution being ...”. And if such a list of exact characteristics of a circuit were really possible to obtain, it would be preferable, for the algorithm to work the other way around: “Given all the functionality and technology constraints, synthesize a circuit that has the following timing and power properties ...”. **Constraint-driven synthesis producing a circuit 100% compliant with the constraints is the ultimate goal.** Note that no verification is needed in such case. How feasible the idea is remains to be seen.

Both timing and power verification are very complex problems. It was demonstrated in [DKMW94b] that already the simple task of event-driven simulation has exponential complexity. The decision problem whether a given circuit has delay more than a certain limit is an NPC (Non-Deterministic Polynomial Complete) problem. It was shown in [LaB94] how such a problem can be converted into a general satisfiability problem (SAT). The complementary problem of computing the exact delay is at least NPH (Non-Polynomial Hard), because it requires the application of the decision algorithm over a space that is exponential in the number of circuit components. Computation of switching activity has a similar complexity to delay calculation. Already the number of transitions on a net caused by a single transition on the inputs can be equal to the number of different paths through the net. The number of paths in a circuit can be exponential in the number of the gates. It was shown in [DKMW94b] how to construct circuits in which the number of transitions is exponential in the number of the gates.

Timing verification methods are reviewed in Section 2-2 on page 40 and power verification methods in Section 2-3 on page 60. Among all the methods presently available, the author considers the approach in [LBSV93] to be the most comprehensive framework for timing and power verification. The state of the art of power verification and design is represented by methods for low-power synthesis, e.g., [KKRV95, LanR95], and by static verification methods [Najm94, MDGK97].

1-3.1 Problem definition

This thesis addresses a much narrower problem than the entire field of power consumption verification. The most often used implementation of a finite state machine today is a synchronous sequential CMOS circuit. The most important component of consumed power is *dynamic power*, i.e., power consumed due to switching.

The problem is to compute how much power a circuit will consume, knowing its gate-level netlist and operating conditions, without manufacturing the circuit first. The essential data for obtaining dynamic power is the switching activity in the circuit. Switching activity can be computed in many ways, however, most methods only estimate it rather than bound it (see Section 2-3.2 on page 61 for detailed analysis). That leaves an uncertainty about what the peak power and average power can be in the worst case. High peak power could lead to circuit failure due to voltage drops in power busses. High average power could cause package overheating and metal migration. **In this thesis we compute an upper bound on switching activity in the combinational part of a synchronous sequential CMOS circuit.** That can be converted to dynamic power and voltage drops in power buses using technology libraries. We assume fixed gate delays, but an extension to interval delays is possible. How the search for an upper bound on switching activity is done is outlined in the next section.

1-3.2 Outline of the proposed solution to bounding switching activity

The approach described in our thesis is based on propagation of waveform sets. It views the circuit as a constraint system rather than a simple interconnection of Boolean operations. The constraint system is formed of constraints (gates, and operating conditions) and variables (circuit signals). At first glance, it might look not too far from a sophisticated simulation, but it is only the outer appearance. The first major difference is the underlying theory for building the constraint system and using its properties: A variable holds an abstracted set of signals on a net representing all possible behaviors of the circuit under a given set of constraints. The second major difference is that the gate constraints are full relational projections of type $terminal = f(other\ terminals)$ rather than the usual Boolean $output = f(inputs)$, where a *terminal* is either an output or an input of the gate. The method rapidly obtains an initial conservative upper bound on switching activity and then refines it by applying heuristics. We propose several heuristics, using fanout, reconvergent regions, and global learning. Several case analysis methods are also

tested, one of them also in a parallel implementation on a network of inexpensive computers.

1-3.3 Original contributions

The main scientific contribution of this thesis is the investigation of use of a constraint resolution method for the determination of an upper bound on switching activity. To the author's knowledge, it has not been published or announced before.

The contribution of the author alone:

- Developed and implemented the set arithmetic for constraint system variables for switching activity calculation (called the *PW* part of the implementation),
- Developed and implemented in C++ three case analysis algorithms (called SCA, PCA, HPCA),
- Developed an enhanced algorithm for reconvergent analysis based on the definitions in [MaaR90],
- Customized an existing global learning algorithm and implemented it in C++,
- Developed an algorithm and implemented a Perl prototype for a parallel version of the case analysis HPCA,
- Defined, developed, and implemented Postscript and Maker Interchangeable Format (MIF) graphical output interfaces for the power portion of the ICproject.

The author together with his research directors:

- Defined the theoretical basis of the constraint system for the verification of switching activity. It is based on the constraint system for timing verification [CerZ94],
- Investigated various properties of the constraint system,
- Implemented in VHDL a prototype of the constraint system for timing verification.

The author supervised undergraduate students in:

- Writing conversion scripts and programs for various benchmark formats in csh, awk, and C,
- Implementing MIF graphical interface for the timing version of the project in C++,
- Implementing an enhanced reconvergent region analysis algorithm in C++,
- Implementing a parallel version of case analysis HPCA in C++.

The author made use of the following existing code:

- IC InCore data structures from Functionality, Inc., provided by Nortel, Ottawa,
- Convertor from the IC InCore datastructures to a proprietary circuit database provided by the LASSO laboratory, University of Montreal. The source code module is referred in this thesis as “UDM”. UDM provides infrastructure such as, e.g., access to netlist or state-saving.

1-3.4 Plan of the thesis

Chapter 2 on Page 27 contains an overview and an analysis of existing timing and switching activity verification methods. We also define the terminology, indicate known problems, and refer to related subjects in the chapter.

Chapter 3 on Page 83 presents the proposed verification method. We give the theoretical basis of constraint resolution, and show how the circuit is translated into a constraint system. Then we explain how an upper bound on switching activity is tightened by case analysis and the associated heuristics. Global learning and reconvergence analysis algorithms are described in separate sections of that chapter.

Experimental results are summarized in Chapter 4 on Page 123. The chapter provides a description of the conducted experiments, their results in a tabular form, and an analysis of the results. The parallel implementation of the case analysis and

the results obtained are described in Chapter 5 on Page 145. The problem of chip reliability due to voltage drops in power busses and the computation of switching activity profiles is addressed in Chapter 6 on Page 179.

Conclusions based on the results and the overall experience with the development of the switching activity verification method are drawn in Chapter 7 on Page 194. Remaining problems with an indication of possible ways to resolve them as well as potential future developments of the proposed method are also listed in Chapter 7.

In Appendix A we describe the implementation of our switching activity verification method. It shows the basic architecture of C++ classes of the ICproject implementation, the data types used, and other enhancements such as the parallel case analysis algorithm and the graphical output interfaces. A brief user manual can be found in Section A-3 on page ccli.

CHAPTER 2

Verification

methods for timing

and switching

activity

In this chapter we present a survey of the published methods for the verification of timing properties and switching activity in synchronous sequential CMOS circuits. First we explain general issues, such as the modeling of digital circuits, the false path problem, and delay correlation. Next we survey verification methods that are applicable to both the timing and the switching activity verification methods, followed by methods specific to each of the two activities. A small section is also devoted to the conversion from switching activity measures to power consumption measures.

We are mainly concerned with synchronous digital CMOS circuits. The reason is that CMOS is the prevailing technology today. However, many results and methods can be generalized to other digital synchronous circuits. Our models are intended to be general, but when technology details are needed, then CMOS is assumed.

2-1. Modeling

This section defines the terminology used in the thesis, focuses on modeling of real circuits, and overviews the most important issues related to timing and power verification.

2-1.1 Timing and power properties of synchronous circuits

A typical *synchronous sequential circuit* consists of a combinational part and edge-triggered flip-flops such as shown in Figure 16. The *combinational circuit* combines the primary inputs (PI)

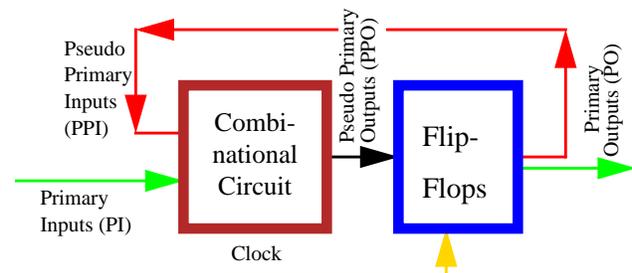


FIGURE 16: Synchronous sequential circuit

and the FF outputs (PPI¹) and produces the inputs of flip-flops (PPO). On the active edge of the clock signal the value of PPO is memorized in the flip-flops and the resulting change on PPIs may cause many changes on the internal nodes of the combinational circuit. The basic timing properties which characterize synchronous circuits are the following:

- The delay of the combinational circuit (CC_{delay})
- The clock to output delay of the flip-flops ($CtoQ$)
- The *setup* and *hold* times of the flip-flops
- Delay in the clock distribution lines

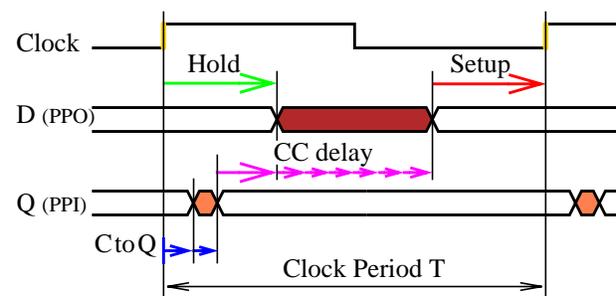


FIGURE 17: Timing properties of the synchronous circuit

1. Pseudo primary inputs could also be called “secondary inputs”. The reason why the term “PPI” is used in literature is probably that PPI are no different that primary inputs in a sense that are also driven by flip-flops. The only difference is that those flip-flops are not part of the circuit that we are investigating.

- The clock period T

The relationship between the signals as shown in Figure 17 can be described by two inequalities. Satisfaction of the inequalities guarantees proper functioning with respect to the timing properties.

- $CtoQ_{min} + CCdelay_{min} > Hold$
- $CtoQ_{max} + CCdelay_{max} + Setup < T$

The model in Figure 17 is a simplified model which does not consider the delay in clock lines or the variations in the clock frequency (clock jitter). A typical clock distribution network is shown in Figure 18. The clock

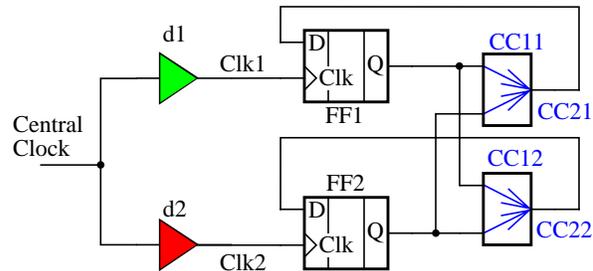


FIGURE 18: Clock distribution delay

signal $Clk1$ of the first flip-flop is delayed by the clock distribution buffers by delay $d1$. The second clock $Clk2$ is delayed by $d2$, both $d1$ and $d2$ are considered to be within some intervals of uncertainty.

The verification of the hold time $Hold1$ on the first flip-flop ($FF1$) with respect to the combinational delay $CC11$, and the verification of the setup time $Setup2$ on the second flip-flop ($FF2$) with respect to the delay $CC21$ can be done as shown in Figure 19. Two more checks (setup and hold along $CC21$),

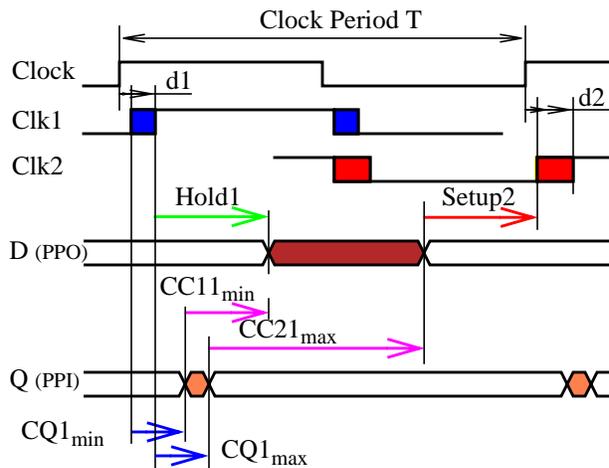


FIGURE 19: Setup and hold time considering clock distribution delay

must be performed on FF1. All the needed setup and hold checks on FF1 are summarized in the following expressions. Note that the hold check along CC11 is independent of the clock network delay $d1$, therefore min/max is not considered

- $d1 + CQ1_{min} + CC11_{min} > Hold1 + d1$
- $d2_{min} + CQ2_{min} + CC21_{min} > Hold1 + d1_{max}$
- $d1_{max} + CQ1_{max} + CC11_{max} + Setup1 < T + d1_{min}$
- $d2_{max} + CQ2_{max} + CC21_{max} + Setup1 < T + d1_{min}$

From the point of view of the power consumed by the circuit, the only important thing is how much current each gate draws from the power buses. In the prevailing CMOS technology, the current waveform is not

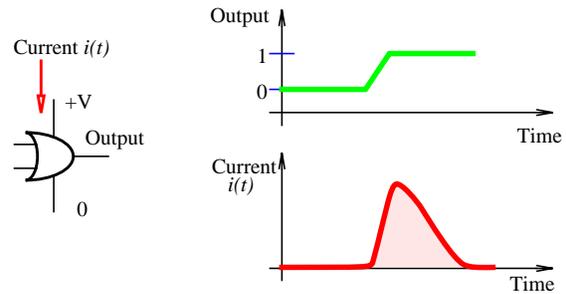


FIGURE 20: CMOS gate power consumption

constant but depends on the behavior of the gate. A significant portion of energy is consumed when the gate undergoes a transition on its output, e.g., as shown in Figure 20. Thus a very important factor is the *switching activity* of the circuit, i.e., the number and the distribution in time of signal transitions. For a general synchronous circuit (an implementation of an FSM), most of the transitions occur on the internal nodes of the combinational circuit as shown in Figure 21.

Gate g draws current $i_g(t)$, and the current drawn by the whole circuit is

$$i(t) = \sum_{\text{all } g} i_g(t). \text{ Instantaneous power drawn by the circuit is then } P(t) = V(t) i(t),$$

where $V(t)$ is the power supply voltage. *Average power* over the i -th clock period T

$$\text{is } P_{av_i} = \frac{1}{T} \int_{iT}^{(i+1)T} P(t) dt.$$

When a sequence of input vectors is assigned to the PPIs then the average over n of clock periods is called the *average power*,

$$Pa = \sum_{i=0}^{n-1} Pav_i. \text{ When the}$$

vectors on PPIs are assumed independent and the worst case is used, then such Pav_i is called *peak power* $Pav-max$, because a real sequential circuit cannot consume more power than $Pav-max$ during any clock period. In this thesis we compute an upper bound on peak power.

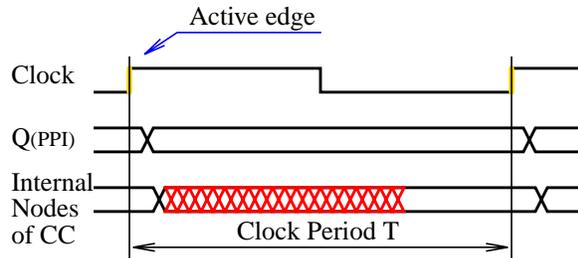


FIGURE 21: Switching behavior of a synchronous circuit

2-1.2 Delay representation

The most commonly used gate-delay models are as follows:

- Fixed delay - d
- Bounded delay - $[0, D]$
- Interval delay - $[d, D]$

The most natural is the *fixed delay*, it corresponds best to the physical reality. If a delay is considered as the separation of two events, it is accurately expressible (in the macro world) after both the events have happened. However, the fixed value assumes certain exact conditions of temperature, power supply voltage, etc. To describe a set of all possible values it is convenient to use an interval. *Bounded delay* sets only the upper bound of the interval, while the most general interval delay specifies both bounds.

Interval delay is interpreted as “any value from the interval”, i.e., the verification algorithms must evaluate all possible cases. In general, the number of such cases is infinite (assuming dense time).

2-1.3 Gate delay models

A gate is a component performing a logic function. The behavior is such that an analogue output waveform is a function of all the input wave-

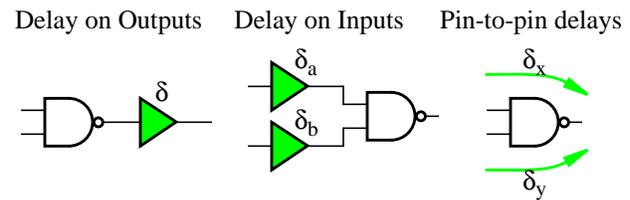


FIGURE 22: Gate delay models

forms. At a higher level of abstraction, transitions of digital signals are used instead of continuous waveforms and a propagation delay is thus introduced. The models used in practice are shown in Figure 22 from the simplest one to the most general one.

Delay-on-Inputs is often used to model the *Pin-to-Pin* delays for single-output gates when the same rising and falling delays are used¹. *Rising (falling)* delay is the propagation delay of a transition which causes a rising (falling) transition on the output. All the delay parameters can be specified as fixed, bounded or interval.

2-1.4 Combinational circuit delay model

A *combinational circuit* is modeled by a directed acyclic graph (DAG). Gates are represented by nodes, and their interconnections by edges. When dealing with combinational circuits only, pseudo primary inputs are referred to as the primary inputs (similarly for the outputs).

The *topological delay* of a circuit is simply measured by the lengths of the paths through the circuit using the gate delay metric. The minimum d_{min} (maximum d_{max}) topological delay is the length of the shortest (longest) path from any primary input to any primary output.

1. Otherwise the delay of a gate such as XOR cannot be captured correctly. The restriction disappears if the gate function is such that we can determine the polarity (rise/fall) of the output transition from the polarity of the input transition alone.

The *exact circuit delay* (or actual or real delay) indicates how long it takes for a transition on a primary input to propagate to a primary output. The shortest (resp. longest) such time is the minimum (resp. maximum) exact delay. Considering all of the possible instances of the circuit at the same time (e.g., using sets or intervals of values to model the gate delays), both the minimum and the maximum exact delays are sets of values, usually simplified to intervals. Then the minimum exact delay is described as $[d_{emin}, D_{emin}]$, and the maximum exact delay as $[d_{emax}, D_{emax}]$. In most cases it is not necessary to specify the delay in such detail, therefore the exact circuit delay is described only by $[d_{emin}, D_{emax}]$. Because the maximum exact delay is used more often than the minimum exact delay, it is in literature sometimes referred to as the “exact delay” in the literature.

The exact delay depends on the function of the circuit as well as on the specific operating conditions on the primary inputs. The following models are considered

(Figure 23):

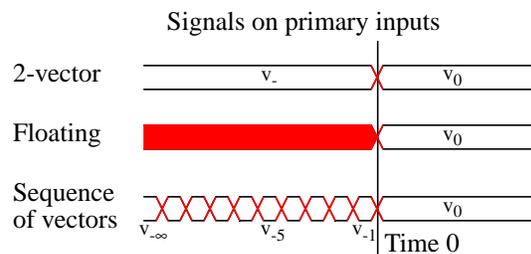


FIGURE 23: Circuit delay models

- *Two-vector delay* (or *transition delay*) model. The primary inputs of the combinational circuit under the two-vector delay model are assigned one value (vector) at time $-\infty$ and the second one at time 0.
- *Sequence of vectors*. There is a sequence of arbitrary vectors with the first one applied at time $-\infty$ and the last one at time 0.
- *Floating-mode delay model*. The primary inputs are in an unknown state from time $-\infty$ to time 0 and then stable at a defined value. The model also assumes that all the internal signals have unknown values prior time 0.

The difference between either the exact circuit delay or an estimated one and the specified delay for the circuit is called timing slack or simply *slack*. It is a measure

of how the implementation satisfies the specification. It is also very often used during optimization, where it indicates the liberty of modifying the timing parameters, e.g., by replacing the modules by cheaper and slower ones. Slack can be defined along circuit paths in a similar way to topological delay.

2-1.5 The false path problem

Apart from many difficulties with modeling of circuits using the different delay models, an important issue is the so-called *false path problem*.

Consider the circuit in Figure 24. A *path* in a combinational circuit is defined by the path in the corresponding DAG. A path is a sequence of nodes and gates such that the output of a gate is connected to the following node n_i and that one is an input to the subsequent gate g_i , etc., e.g. $P=\{PI, g_0, n_1, g_1, n_2, \dots, g_k, PO\}$.

The *side inputs to a path* $P=\{PI, g_0, n_1, g_1, n_2, \dots, g_i, n_i, \dots, g_k, PO\}$ at gate g_i are all inputs to this gate except n_i . The *side inputs to a path*

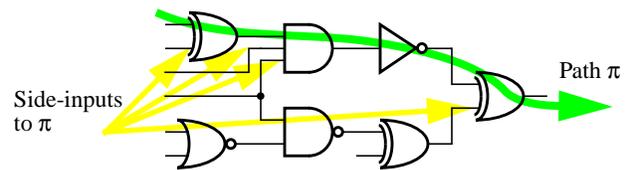


FIGURE 24: Path and side inputs to the path

are the side inputs to the path at every gate on the path. A side input has a *controlling value* if that value controls the gate output value (e.g., value 0 for an AND gate and 1 for an OR gate). A *non-controlling value* allows the other inputs to influence the gate output.

The topological delay gives an upper bound on the maximum exact circuit delay ($D_{emax} \leq d_{tmax}$) and a lower bound on the minimum circuit

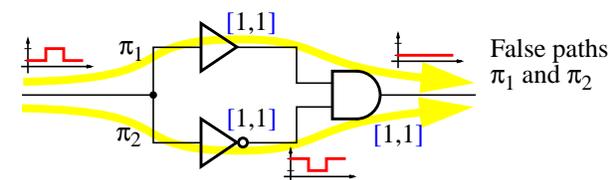


FIGURE 25: False path demonstration

delay ($d_{emin} \geq d_{min}$), but these may be very loose bounds. In real circuits there are many (topologically) long paths through which a transition on a primary input can-

not propagate to the primary output and consequently the actual delay becomes lower than the topological one. An example is shown in Figure 25. The circuit output is at stable 0 which corresponds to an actual delay of zero. But the topological delay is 2, assuming that zero-width glitches do not propagate (there are two false paths each of length 2).

A *false path* can be defined as a path through which a transition on a primary input cannot propagate to a primary output (under the specified delay model). Thus the exact circuit delay is determined purely by the longest path(s) which is (are) not false.

The cause of the existence of a false path lies in the topological structure and function of the circuit. It is due to the existence of reconvergent paths.

Topological and the exact delays may differ substantially in real circuits. For instance, circuit c1908 from the ISCAS-85 testability benchmark suite has the actual delay of about 93% of the topological delay using fixed gate delays of one unit of time.

2-1.6 Spatial and temporal correlation

The false path problem described in the previous section is the result of *spatial correlation*, which means that in the space of Boolean values (where each circuit signal is one dimension) the signals are correlated, i.e., they cannot have any combination of Boolean values because they are derived one from the other (Figure 26).

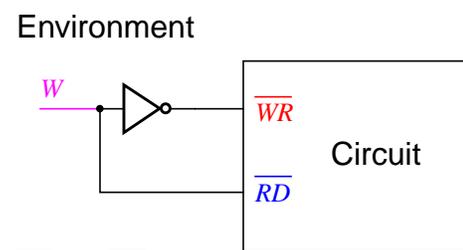


FIGURE 26: Spatial correlation

The inputs to a combinational circuit are driven by the external environment. If these inputs depend on the output of the circuit like in a synchronous sequential circuit,

the input pattern to the combinational circuit at time T_i depends on the input pattern at time T_{i-1} . This is called *temporal correlation* of input patterns (Figure 27).

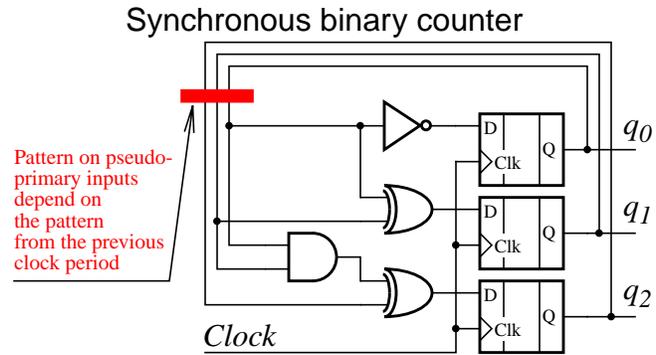


FIGURE 27: Temporal correlation

The temporal and spatial correlations are real phenomena occurring in the circuits because of the circuit function and topology. Some verification methods can handle spatial and temporal correlation which allows them to compute more realistic power estimates. In the next section we show yet another type of correlation.

2-1.7 Component parameter correlation

As much as the signals can be correlated in both values and time, the circuit component parameters can be correlated too. Component parameter correlation is not a phenomenon which occurs in real circuits. A circuit, at one precise moment in time has a unique value of each parameter. However, those parameters can change rapidly (such as temperature) or may differ for each manufactured instance of the circuit (such as power supply voltage) or are different for each instance of a component of the same type in the circuit due to manufacturing imperfections (such as the gate delays). To obtain more realistic results when applying a verification method to a circuit described by the manufacturer's specification (i.e., intervals of parameter values) rather than on measured parameters for each circuit instance and precise time instant, component parameter correlation can be introduced into the verification method.

Interval values of some parameters (like gate delays) are used to model the groups of instances of components. However parameters of gates on a single semiconductor wafer are not independent. For instance, all gates in a circuit can have delays

guaranteed to fall into e.g. [1.0ns, 2.0ns]. But the manufacturing process for gates on the same wafer may cause only about 10% difference from a typical value which is in the interval, e.g., [1.1ns, 1.9ns]. Thus for an instance of the circuit, all gate delays may fit well into, e.g., [1.3ns, 1.4ns], and thus the initial assumption [1.0ns, 2.0ns] was too pessimistic. Assume gates g_0, g_1, \dots, g_n . The delay of each gate g_i is modeled by an interval $[d_i, D_i]$ which is a subinterval of $[m_i, M_i]$. The $[m_i, M_i]$ is the (uncorrected) interval delay of the gate. Let K be the correlation coefficient, $0 \leq K \leq 1$. Complete independence of the gate delays is expressed by $K=0$, total correlation as $K=1$. Then for a value $[d_0, D_0]$ of the delay of gate g_0 the delay of the other gates can be modeled by $[d_i, D_i]$,

$$d_i = m_i + (d_0 - m_0) \frac{M_i - m_i}{M_0 - m_0} K \quad \text{and similarly} \quad D_i = M_i + (D_0 - M_0) \frac{M_i - m_i}{M_0 - m_0} K,$$

$i > 0$. A more general model for correlation with gates g_0, g_1, \dots, g_p can be modeled

$$\text{by } d_i = m_i + (M_i - m_i) \left(\prod_{j=0}^p \frac{d_j - m_j}{M_j - m_j} K_{ji} \right)^{\frac{1}{p}} \quad \text{and similarly for } D_i.$$

The introduction of the correlation of gate delays simply makes certain combinations of their values impossible (because they are not realizable), as shown for a two-component circuit in Figure 28. A *non-correlated*

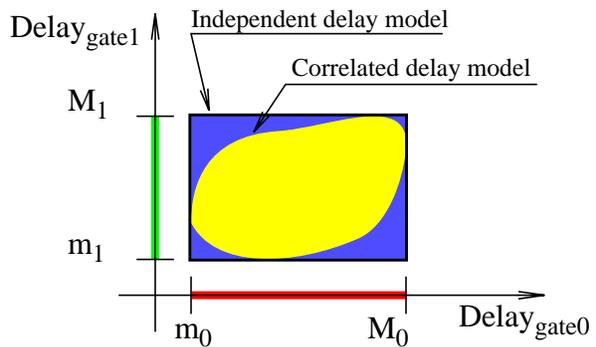


FIGURE 28: Independent and correlated gate delays

model assumes the whole square region delimited by the range of the delay of each component while a *correlated* delay considers only a subspace of the square region. Generally, an n -dimensional space is used for n correlated gates.

2-1.8 Toggle power

A *glitch* in a digital circuit is a temporary transition of a signal to a Boolean value opposite to the current value and then back (static hazard) or making one more transition (*dynamic glitch*).

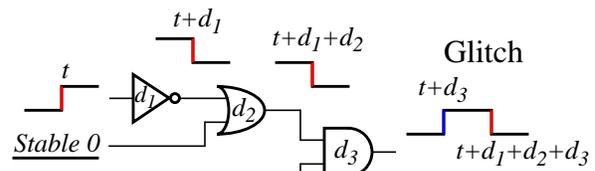


FIGURE 29: Glitching

In a combinational part of a synchronous circuit the final value is the value on the pseudo primary outputs at the end of the clock period, therefore only up to 1 transition on each signal is needed for the circuit functionality. All the other transitions are caused by the circuit implementation. It is not a design error as for the function, but in CMOS circuits each transition draws power, thus increasing the power consumption of the circuit. The power consumed by glitches is called *toggle power*. A typical toggle power is about 20% of the total power, but, e.g., in a combinational adder it can be up to 70% of the total power [SGDK92].

Most modern power verification methods consider toggle power, while the older ones based on signal probabilities do not [Najm94].

2-1.9 Conversion from switching activity to power

Whenever power verification is done at the level of logic transitions rather than at the level of currents and voltages as in analog simulation, the result (i.e., the switching activity) must be converted to the real consumed power.

A CMOS gate consumes power almost exclusively while switching. The reason is that, to produce a transition on the output, one of the output transistors closes while the other one opens. While they are both simultaneously (partially) open, some current flows directly from the power line to ground, sometimes called the *class-A*

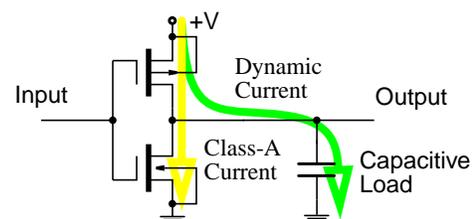


FIGURE 30: CMOS inverter

current flows directly from the power line to ground, sometimes called the *class-A*

current or power supply *crowbarring* or simply *short-circuit current*. The capacitances of wires and of the gate inputs driven by the gate form the *capacitive load*. These capacitances are charged by the transistors of the gate, thus drawing *dynamic current* (Figure 30).

The total power a CMOS gate consumes [DevM95] is given as

$$P = \frac{1}{2}CV^2fN + Q_{SC}VfN + I_{leak}V,$$

where P is the total power, C is the capacitive load on the output, f is the clock frequency, N is the switching activity (number of transitions per clock cycle), Q_{SC} is the quantity of charge carried by the short-circuit current per transition, V is the power supply voltage, I_{leak} is the leakage current. The first term is the power consumed to charge and discharge the capacitive load, the second one is caused by the short-circuit current, and the third one is the *static leakage*. Simplified models of CMOS gates are as follows:

- One unit of energy per transition,
- Current pulse per transition,
- Complex current waveform per transition or sequence of transitions.

The simplest model assumes that a gate consumes a certain amount of energy

$$E_{pulse} = \int_{-\infty}^{+\infty} V i(t) dt = 1$$

to create a transition on its output. Such a model cannot capture the power-current profile quite accurately. Closer to reality is a model which uses current waveforms. There, a gate undergoing a transition on the output draws a current waveform of a certain shape, like the one shown in Figure 20. However, the shape of the current waveform depends on many factors, e.g., which input causes the output transition, whether there are multiple transitions on the inputs, the mutual position of transitions on the inputs, and the capacitive load on the output. For example, some current is drawn even when a pulse appears on a gate input and is absorbed by the inertia of the gate (i.e., the capacitive loads).

Conversion of switching activity to power is a quite straightforward task, knowing the switching activity and the gate power model. In the simplest case of the unit of energy per transition model, the power consumed by a gate in a clock cycle is a product of the switching activity N and the unit of energy E_{pulse} . In general, N is either calculated or measured, and E_{pulse} is taken from the gate library. A practical implementation can use, e.g., a look-up table for E_{pulse} with indices being the capacitive load on the gate output, the number of simultaneous transitions on the gate inputs, and the distribution (proximity) of transitions on the gate inputs, etc.

This chapter summarized the most important issues in both timing and power verification. In the next chapter we will review those verification methods that can be applied in both timing and power verification or are specific to timing verification.

2-2. Universal¹ and timing-specific verification methods

As the problem of calculating the exact circuit delay and power consumption of a circuit is known to be at least NPH (Section 1-3.), many methods are being developed to trade off solution accuracy for computational speed. This section surveys the solution methods and algorithms that can be found in the literature.

2-2.1 Exhaustive simulation

Exhaustive simulation tries to test the circuit behavior for all possible inputs and under all possible conditions. It always gives the exact delay.

It is an ideal method for small circuits with fixed gate delays. The algorithm is obvious - for each pair of input vectors do the following:

1. Apply the first vector to the inputs.
2. Let the circuit stabilize (meaning wait for topological delay or longer).

1. Timing and power

3. Apply the second vector and measure the time of the last transition on the primary outputs.

The only advantage of the algorithm is its simplicity. As it tests all the possibilities it has many drawbacks as shown below:

- *Circuit size* (number of inputs). For a circuit having n inputs, $O(2^{n+1})$ cases must be simulated.
- *Interval delays*. The method completely fails to deal with gate delays ranging in a real number interval as each gate introduces an infinite number of cases to assess. The only remedy is to lower the resolution in the interval. Still the number of cases can be enormous - for m values inside each interval and o gates in the circuit the number of possible cases is $2^{n+1} \times o^m$. For example, consider a small circuit with $n=32$ inputs, $o=2000$ gates and interval delays represented by $m=10$ discrete points. The number of cases is 8.8×10^{42} - by evaluating one case per *Ips*, the entire evaluation would take 2.8×10^{24} years.

Random vector and/or delay generation can be used to acquire results within one's lifetime. But the method loses its greatest advantage - the result is no longer the exact delay. It is only a lower bound, unless one of the measured delays is equal to the topological delay, only then it is exact.

2-2.2 Path-oriented methods

Some of the heuristics which try to compute an upper bound on the circuit delay transfer the problem of eliminating false paths to the problem of satisfying local conditions, using the so-called *sensitization criteria*.

A sensitization criterion helps us to decide whether a given gate can propagate a transition on its input to the output (i.e., is sensitized) by defining local conditions on the gate's inputs and output. If the sensitization criterion is satisfied for all the gates on a path, the path is said to be *sensitizable* under the given criterion.

The following definitions of several sensitization criteria are taken from [SiSa93]. The notation used to describe the criteria is mainly based on that report. The following symbols are used:

- X - stable but unknown value (stable 1 or stable 0),
- C - changing value. A node has value C at time t when it either is X or it undergoes a transition at t ,
- c - controlling value (e.g., 1 for an OR gate),
- c' - non-controlling value (e.g., 0 for an OR gate).

Let g be a gate on a path $P = \{PI, \dots, u, g, v, \dots, PO\}$, u is an input to gate g and v is the output of g . All other inputs to g are denoted s - side inputs of u with respect to v . The event which propagates through P arrives at node u at time $\tau(u)$. The path is said to be sensitizable if each gate on it is sensitizable. The *maximal circuit delay* is the length of the longest sensitizable path.

2-2.2.1 Static sensitization

The criterion is depicted in Figure 31. *Static sensitization* neglects the dynamic behavior of the circuit. The criterion is evaluated when all nodes have stable values. The gate g is said to be sensitizable if all the side inputs s have non-controlling values (when the circuit settled down), e.g., s_1 and s_2 in Figure 31.

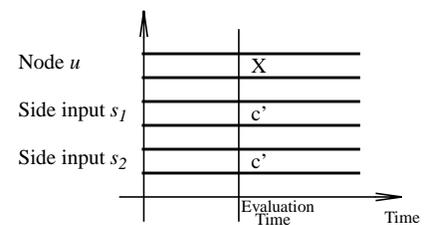


FIGURE 31: Static sensitization criterion

The search algorithm for the *longest statically sensitizable path* consists of identifying all the paths from PIs to POs, and for each path doing the following: try to assign all side inputs of each gate to the non-controlling values. Such an assignment may induce values on PIs (or be inconsistent). The path is sensitizable if

there exists a vector on PIs for which all side inputs of all gates on the path have non-controlling values.

Unfortunately the static sensitization criterion is too far from modeling the actual circuit behavior. Some statically sensitizable paths may not propagate events from PIs (e.g., both paths in Figure 25). Even worse, some paths that are not statically sensitizable can propagate transitions due to static and dynamic hazards that create temporary sensitization [SiSa93].

Static sensitization is of no interest as it either overestimates or underestimates the circuit delay.

2-2.2.2 Dynamic sensitization

The *dynamic sensitization* criterion conditions [MGBr89] are shown in Figure 32. Gate g is dynamically sensitizable if all its side inputs s have non-controlling values at the arrival time of the transition on node u .

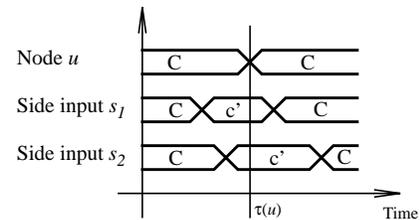


FIGURE 32: Dynamic sensitization criterion

The algorithm for circuit delay computation using the dynamic sensitization criterion is not straightforward.

The dynamic sensitization as defined in Figure 32 may overestimate circuit delay as it does not explicitly require a transition on node u at time $\tau(u)$. Therefore, the following condition must be added:

$$(Eq. 1) \quad \lim_{t \rightarrow \tau(u)^-} u(t) \neq \lim_{t \rightarrow \tau(u)^+} u(t)$$

The dynamic sensitization criterion gives then the exact delay for fixed gate delays but may underestimate the circuit delay under interval or bounded gate delays. The

dynamic sensitization criterion does not satisfy the monotone speedup property (see the definition in Section 2-2.2.3).

2-2.2.3 Viability sensitization

The *viability sensitization* criterion conditions [MGBr89] are presented in Figure 33. A gate is *viable* if each of the side inputs satisfies one of the following conditions:

- The side input has a non-controlling value at time $\tau(u)$, e.g., node s_1 in Figure 33.
- The side input has a stable time greater than or equal to $\tau(u)$. It means that it has either a stable value or undergoes an arbitrary number of transitions with the last one at time $\tau(u)$ or after, e.g., node s_2 in Figure 33.

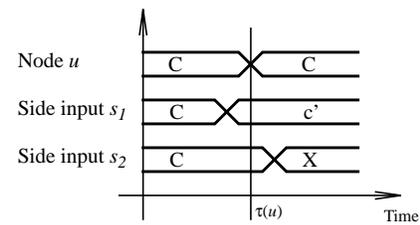


FIGURE 33: Viability sensitization criterion

The algorithm for circuit delay computation using viability criterion is again not straightforward [MGBr89].

Viability sensitization has been proposed as a tight upper bound on the circuit delay under bounded gate delays. It also satisfies the *monotone speedup property*. It means that the circuit delay reported for a circuit with a certain gate delay assignment can never be smaller than one reported for the same circuit with each gate delay equal or smaller.

2-2.2.4 Floating-mode sensitization

The *floating-mode sensitization* criterion conditions [DeKM93] are shown in Figure 34. Gate g is *sensitizable under the floating mode criterion* if one of the following conditions holds:

- Node u has a stable controlling value with stable time $\tau(u)$ and the following is true for the side inputs:

- A side input has a stable non-controlling value at the time $\tau(u)$, e.g., a node s_1 in Figure 34a.

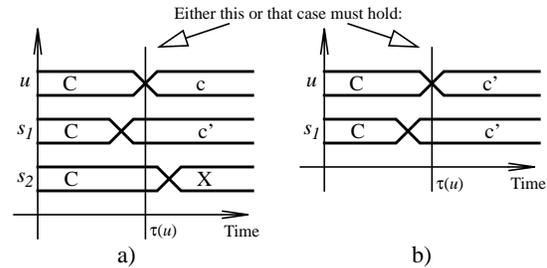


FIGURE 34: Floating-mode sensitization criterion

- A side input has a stable time greater than or equal to $\tau(u)$. It means it has either stable value or undergoes an arbitrary number of transitions with the last one at time $\tau(u)$ or later, e.g., node s_2 in Figure 34a.
- Node u has a stable non-controlling value with stable time $\tau(u)$ and the following is true for the side inputs:
 - A side input has a stable non-controlling value at time $\tau(u)$, e.g., node s_1 in Figure 34b.

The algorithm for circuit delay computation using the floating-mode criterion is again not straightforward [MGBr89].

Floating-mode sensitization never underestimates circuit delay. However it may overestimate it. In [SiSa93] the authors state that the viability and the floating-mode criteria report the same circuit delay. They also show a circuit containing a path which is not sensitizable under floating-mode but may actually propagate a transition (and also that path is viable). However, in that case there exists another path of the same length which is sensitizable under the floating-mode criterion and thus the circuit delay is the same under both criteria.

The *difference between the viability and the floating-mode criteria* is shown in Figure 35. The gate g is not sensitized under the floating-mode criterion (but is under viability) when there is a non-controlling value on node u and all the side inputs reach stable values later than the time of the transition on u , i.e. $\tau(u)$.

The situation in Figure 35 means that the transition on the output of gate g at $\tau(u)$ or later may be caused by either:

- the transition on u at $\tau(u)$ while s_I has a non-controlling value at that time, or
- the transition on s_I at $\tau(u)$ or later as u has a non-controlling value after $\tau(u)$.

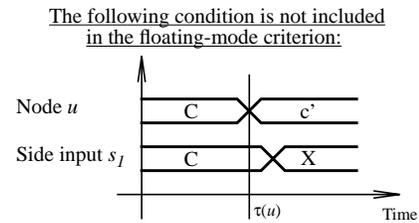


FIGURE 35: The difference between the viability and floating-mode sensitization criteria

The first one of the two cases is covered by another rule of the floating-mode sensitization criterion - node s_I in Figure 34b. In the second case, by swapping u and s_I there is a transition on s_I and its side input u has a non-controlling value. Thus the path $P_1 = \{\dots, u, g, v, \dots\}$ is not sensitizable under floating-mode, but another $P_2 = \{\dots, s_I, g, v, \dots\}$ is. Therefore, the circuit delay reported under viability sensitization and the one obtained under floating-mode sensitization are the same. The fact that under the viability criterion the node u can be changing (denoted by C) after time $\tau(u)$ while under the floating-mode criterion it has to be stable (X) is not of great importance, as any later transition on u after $\tau(u)$ will be confronted with the criterion at that later time (and recognized as influencing the circuit delay).

2-2.2.5 Lower bound sensitization

The *lower bound sensitization* criterion has been originally introduced by [SiSa93]. It is really not obvious to the author of this thesis why such criterion guarantees computing a lower bound on the circuit delay. No rigorous proof is given in [SiSa93]. It was only explained that this criterion eliminates path sensitization caused by glitches. The criterion is presented here only to keep the survey complete. The conditions required by the lower bound criterion are shown in Figure 36. Gate g is *sensitizable under the lower bound criterion* if one of the following holds:

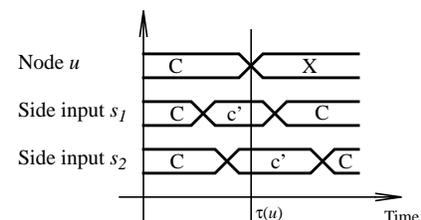


FIGURE 36: Lower bound sensitization criterion

- There is an effective change of value on node u at time $\tau(u)$ and after that it remains stable.
- A side input has a non-controlling value at time $\tau(u)$ like nodes s_1 or s_2 in Figure 36.

2-2.2.6 Vigorous sensitization

In [ChA93] the authors propose the so-called *vigorous sensitization criterion*. Gate g is vigorously sensitizable if it complies with the following conditions:

- When u has a non-controlling value all its side inputs have non-controlling values too.
- When u has a controlling value, no side input that arrives earlier than u has a controlling value.

The first condition is shown on the left-hand side of Figure 37. The second condition says that the side inputs either arrive earlier and have non-controlling values (s_1 on right-hand side of Figure 37) or arrive later and we do not care about their value (s_2 on right-hand side of Figure 37).

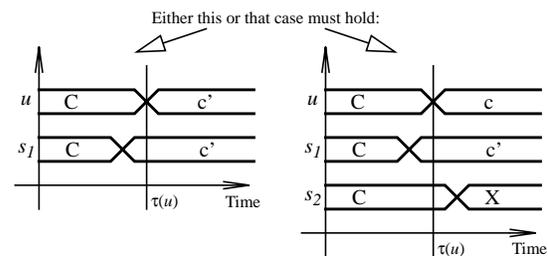


FIGURE 37: Vigorous sensitization criterion

If you swap the left- and right-hand sides of Figure 37 and compare them with Figure 34 you can conclude that they are the same. It seems that there is no difference between the vigorous and floating-mode criterion.

2-2.2.7 Comparison of several sensitization criteria

Finally, in [SiSa93] the authors compare the circuit delays calculated by different sensitization criteria as shown in Figure 38. The circuit delays are labeled using the following sensitization criteria:

- L - lower bound
- D - dynamic
- E - exact delay (i.e., the actual delay of the circuit, not a criterion)
- F - floating-mode
- V - viability

The delay computed using dynamic sensitization is the exact circuit delay for fixed gate delays. The floating-mode and viability circuit delays are the same for bounded gate delays. For interval and bounded delays the floating/viability criterion provides an upper bound.

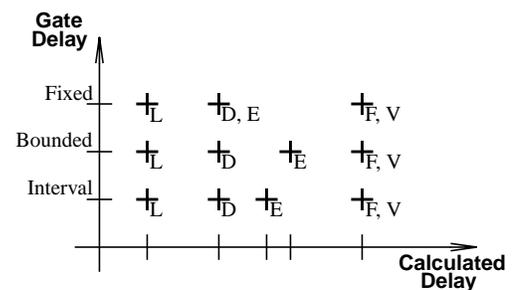


FIGURE 38: Categorization of sensitization criteria

2-2.3 Algorithms for solving the false path problem

This section briefly presents some algorithms or at least their basic ideas for solving the false path problem. The algorithms shown here use the dynamic, viability and floating mode sensitization criteria. Also, one example is based on an ATPG (Automated Test Pattern Generation) algorithm. The algorithm for static sensitization was outlined in Section 2-2.2.1.

2-2.3.1 An algorithm for computing the longest viable path

An algorithm for computing the longest sensitizable path and its detailed refinement for the viability sensitization criterion is given in [MGBr89].

The general algorithm handling a set of paths at a time is depicted in Figure 39. The set of partial true paths from PI is maintained. At each step a path from the list is selected and if its last node is a PO then it is a full true path and its length is recorded. If not, any of its already unexamined fanouts is chosen. If the sensitization function for such a possible path extension is satisfied then the new extended path is added to the list of true partial paths.

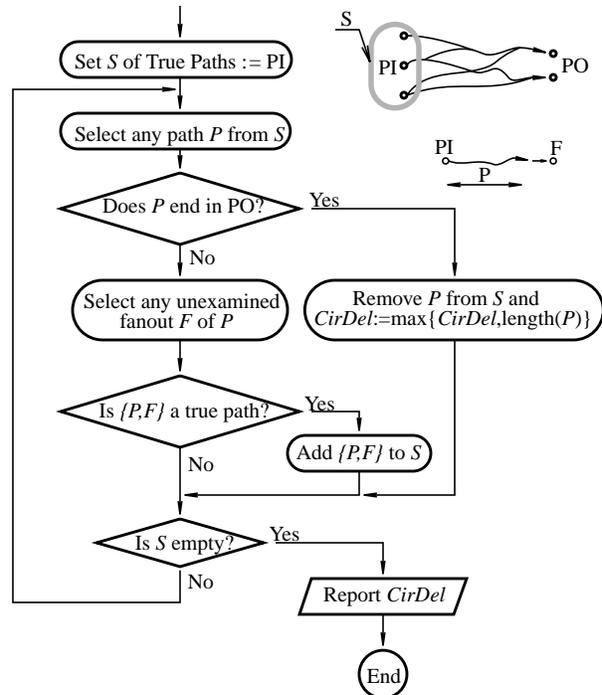


FIGURE 39: General verification algorithm handling a set of paths at a time

The *best-first* version of the algorithm always takes the best possible extension of the path and if the extended path is true then it is inserted to the list of partial paths. The algorithm terminates when the first path reaches a primary output, because the path-building method assures that no other true path is longer. The process of selection of the best (longest) extension of the partial paths requires very expensive search.

The *depth-first* version keeps the partial true paths on a LIFO stack. If a path reaches PO its length is compared with the longest path found so far. The algorithm terminates when the stack is empty.

The problem is to define the decision function which distinguishes the false and the true paths. The function is used locally for each gate on the path but generally will depend on the transitive fanin of the currently investigated gate. The function

and its effective recursive application is the core of the algorithm and can be found in [MGBr89].

2-2.3.2 An algorithm based on ATPG

A different method for floating-mode circuit delay computation can be found in [AMR93, DeKM93]. It benefits from the existing implementation of an ATPG algorithm. It first transforms the circuit and then applies known test generation procedures. The answer to the question whether there is a possible timing violation at time δ or later on the outputs of a circuit is *NO* if no test is generated for the modified circuit. The problem is only what kind of transformation to do and what faults to test for.

The theoretical background is given in [DeKM93]. It deals with a set of paths rather than one path at a time. The most important result is that it intro-

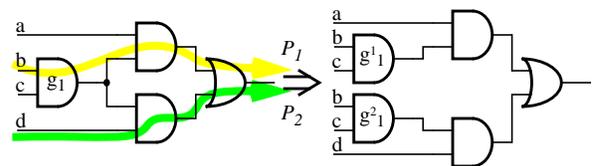


FIGURE 40: Making a circuit fanout tree

duces a relationship between the truth or falsity of a set of paths and the testability of a multifault in the equivalent normal form (ENF) representation of the circuit. The algorithm is called *timed test generation* and the associated calculus *timed D-calculus*. The ENF is a two-level circuit obtained from the original one by first replicating shared parts to make it fanout-tree (each gate drives not more than one gate input) as shown in Figure 40. Then, inverters are pushed to the primary inputs using the de Morgan's law (see Figure 42 on page 52). Once this is done, the inverter-free tree circuit is changed to its two-level representation, e.g., Figure 41. The path information is retained in the ENF as each primary input (literal of ENF) corresponds to a path in the original circuit (demonstrated on paths P_1 and P_2 in Figures 40 and 41).

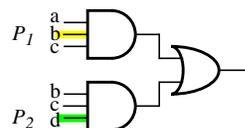


FIGURE 41: Two-level circuit representing ENF

Due to the replication the resulting circuit can be exponential in the size of the original one. Therefore, the algorithm itself uses only the properties derived using ENF, but the computation is done on the original circuit.

The presentation in [DeKM93] deals with rising transitions on PO only, since the falling ones are a dual case. It shows that for a path to be the cause of the latest rising transition (and thus the path determining the circuit delay), there must exist a cube in ENF with a literal causing the final value 1 on PO and that there must be no other cube with a literal causing a rising transition earlier. The authors define a criterion to decide whether a set of paths contains a true path. Finally, the conclusion is that a set of paths contains a true path for a rising transition if and only if there is a test for a multifault stuck-at-0 on all literals corresponding to the paths from the set.

The *timed test generation* algorithm performs the test for a multifault on the original circuit. The evaluation of a gate output is done by *timed D-calculus* which is very similar to the well-known propagation of a Boolean fault value in ATPG. The difference is that here the different branches of a net with fanout greater than 1 may evaluate to different values. Another difference is that timing information is used for the evaluation of D (the error value).

The algorithm based on timed test generation is described in [DeKM93] which also contains experimental results for some circuits. For instance, the c1908 circuit from the ISCAS-85 benchmark suit took 3675 CPU seconds on a SUN-4 and the topological delay of 34 was reduced to 31 (91%). This data is for the reader to be able to compare it with another implementation described at the end of this chapter.

In [AMR93], the authors remove the following drawbacks of the timed test generation:

- The modifications to the ATPG algorithm needed for timed test generation are non-trivial.

- The complexity of evaluation of an error value on each gate output is enlarged by taking into account the timing information.
- The error value is computed for each fanout edge separately which again hurts the performance.

The algorithm they propose solves the same problem but more efficiently. The input to the algorithm is a combinational circuit C and the allowed

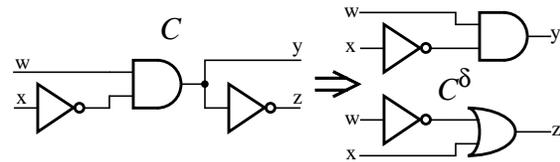


FIGURE 42: Moving inverters to the primary inputs

circuit delay δ . A *circuit transformation* is the first step of the algorithm. Inverters are pushed to the primary inputs of the circuit as indicated in Figure 42. For each gate, the inverters are moved from the gate's output to the gate inputs. When this is not possible the gate is duplicated. This is iteratively applied to all gates, starting from the primary outputs. The new circuit is denoted C^δ .

In the second step, *test pattern generation* is performed on the modified circuit. Multiple stuck-at-0 faults are injected on such fanout branches of PIs (after inverters if there are any) that are the first edges of paths longer than δ (a set of such nodes is denoted δ_leaves) as shown in Figure 43. If no test is found then the same is repeated for the multiple stuck-at-1 faults. If again no test can be generated then the circuit delay is smaller than δ .

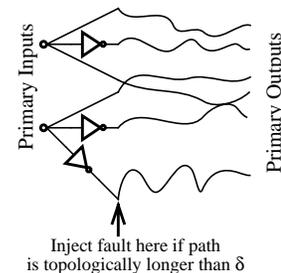


FIGURE 43: Fault injection

As for the complexity of the algorithm, it is determined by the ATPG algorithm assuming that the modified circuit C^δ is not more than twice the size of the original one (each gate may be duplicated at most once).

The actual implementation was done for fixed and bounded delays, and under the floating-mode model. The authors claim that it is delay model independent, how-

ever. For comparison with [DeKM93], the CPU time required for analyzing the c1908 circuit from the ISCAS-85 benchmarks was 800 CPU seconds on a Sparc 2 (of which 780 seconds for ATPG). The preprocessing enlarged the circuit from 550 to 1222 gates. The circuit (topological) delay of 34 was lowered down to 31. Assuming that Sparc2 is about 2x faster than Sun4 the algorithm provides about 2x speedup compared to [DeKM93] (discussed earlier in Section 2-2.3.2).

2-2.4 Optimization-based methods

2-2.4.1 An algorithm for computing the exact circuit delay

The algorithm as well as its theoretical background can be found in [LaB94].

The authors introduce a formalization of the various delay models and then use the more general interval gate delays. The behavior of a circuit is modeled by *Timed Boolean Functions* (TBF): $B^n(t) \rightarrow B(t)$, where $B(t)$ is the so called waveform space - collection of mappings from reals into Booleans, $R \rightarrow B$, $B = \{0,1\}$. The algorithm itself - briefly described - consists of two steps:

1. The input vectors which sensitize the longest sensitizable paths are found.
2. Each gate is assigned a delay (within its min/max interval) such that the paths found in the first step are sensitized.

The first step is what the already mentioned algorithms do only. The second step is added to ensure that all the sensitizable paths are realizable and that the path lengths are maximized.

Experimental results were presented for the ISCAS'85 benchmarks with interval gate delays and using the 2-vector transition delay model, executed on a DECstation 5000. For instance, the c1908 circuit with the topological delay of 41.1 had the circuit delay reduced to 27.2 (by 34%). Unfortunately, this is only an upper bound, because the optimization lasting for 12140 CPU seconds ran out of memory.

2-2.4.2 An algorithm based on constraint satisfaction

The method proposed in [CerZ94] describes a transformation of the set of constraints defining the exact circuit delay to a set of equations over the lattice of sets of abstract waveforms. The transformation of the network constraints is independent of the circuit delay model used, transition (delay by two-vectors) or floating-mode. This is determined by the initial constraint on the primary input waveforms. The user expresses the timing checks by constraints on the primary output waveforms. This is done by the complementary interval - to verify that circuit delay is smaller than t , the constraint “require transition in $[t, \infty]$ ” is added.

The algorithm tightens the constraint system. If it is found inconsistent then there is no transition on POs within the intervals specified by the user. If the constraint system is consistent then nothing can be concluded because the “require transition” constraint on the outputs is satisfied, but there might not be any transition in the real circuit. Such a false-negative answer is due to the abstraction introduced by the mapping and the approximate method used for the evaluation of the constraint system. The algorithm performs limited case analysis, tightens certain constraints and evaluates each possible case.

The method was originally implemented in VHDL which has the advantage of direct use of the circuit specification in VHDL. The results for the c1908 circuit from the ISCAS’85 benchmarks run on a Sparc10 are the following: the circuit topological delay was reduced from 400 down to 370 (reduction of 7.5%) in 60 CPU seconds under the floating-mode delay model with gate delays $[0, 10]$ and with user-guided case analysis. The CPU time was further improved in the C++ implementation.

2-2.4.3 Methods treating correlated component delays

A timing verification algorithm for correlated components was proposed in [SivS93]. The algorithm first selects the longest potentially sensitizable paths. Then for these paths, in a decreasing order of lengths, the sensitization expression

for the exact (dynamic) circuit delay is found. It consists of timed Boolean variables. The satisfiability of the expression induces a set of constraints on the gate delays. If an intersection of the space defined by that constraint system and the one defining the correlation is non-empty it means that there is a feasible delay assignment of gate delays. Consecutive optimizations select such a delay assignment for which the length of the currently evaluated path is maximal.

Generally, the timed Boolean expression would be $f(i_k[t_l])$, where i denotes a node, k ranges over the nodes on the path and l over the times the dynamic sensitization is checked at the given gate. Such an expression is hard to handle (the method similar to that reported in [LBSV93] must be used). The authors simplify the method by allowing only one primary input to change at a time. Thus $i_k[t_l]$ is not timed for all but one node, i.e., $i_k[t_l] = i_{k_l} \in \{0, 1\}$ for $k = k_l$. The only source of transitions in the circuit is node i_{k_l} , and only the paths reconverging on it must be examined (i.e., such a reconvergent side path brings one timed variable to the sensitization expression). Furthermore, the timed variables representing the reconvergent paths which are always longer (or always shorter) than the currently evaluated path over the whole gate parameter space are replaced by their values. Thus the only timed variables in the sensitization expression are those representing the reconvergent side paths which may be either longer or shorter than the current one.

The sensitization expression is transferred to the constraint system and checked for its realizability by finding a fixpoint. If realizable (has an intersection with the space defined by the correlation) the parameter assignment (circuit instance) for which the path delay is maximal is searched for. If the path's length is greater than the one found so far, the path is recorded including the parameter assignment and the test vector.

Experimental results are provided in [SivS93] for several circuits from the ISCAS'85 benchmark suit. Unfortunately, the c1908 which was used for comparison of the other algorithms is not included. Also, the definition of the parameter

subspace (delay correlation formula) for which the computation was done is missing. E.g., for ALU c5315 they reduced the delay down to 99% of the topological delay in 294 CPU minutes. For comparison, in [LBSV93] the authors reported delay reduction to 89% in 12 CPU minutes (under the true transition delay model but with no gate delay correlation). For some other circuits, the CPU time is usually more than four times greater than those reported in [LBSV93]. The CPU time is compared just to give a flavor of computational complexity because a direct comparison is not fair - the algorithm in [LBSV93] does not handle delay correlation.

The advantages and drawbacks of the method in [SivS93] can be summarized as follows:

Advantages:

- Computes the exact delay,
- Handles correlated gate delays,
- Identifies the longest true path(s) as well as the gate parameter assignment and a test vector.

Disadvantages:

- Does not provide the usual transition delay model as only one input is allowed to change at a time. That introduces certain inaccuracy to the solution. The extension of the method for the actual transition delay model would increase the time complexity (most likely exponentially),
- Enumerates paths which again increases the computation time.

Another version of the method for the verification of setup-time violations based on constraint resolution is presented in [AouC97]. The constraint system is used for representing the circuit including the notion of waveform classes as in

[CerZ94]. But the waveform representation is more refined to identify where the first and the last transitions in one clock period must occur. Component delay correlation is naturally included as an additional constraint to the system of constraints representing the circuit gates and input waveforms.

The floating mode delay model was enhanced in [SiSt97] to include delay correlation. The method also allows some incremental updates (of netlist) which makes it suitable for embedded timing engines (e.g., in optimization or routing tools).

2-2.5 Various problems and notes

2-2.5.1 Transition delay and the minimum clock period

When a combinational circuit is used in a synchronous sequential circuit as its combinational part, it is not obvious that the circuit transition delay can be used as the *minimum period* of the clock (even when neglecting clock skew and hold time). The problem as discussed in [DeKM93] is briefly described next.

The transition delay model assumes that two vectors are applied to PIs - the first one at time $-\infty$ and the second one at time 0 . This allows all the circuit nodes to stabilize before the second vector is applied. The delay reported as the circuit delay is the time of the latest transition on a PO (maximized over all possible pairs of vectors on PIs). Although at that time a stable value appears on all POs, this is not necessarily true on all internal nodes. The problem is that the sequential circuit clocked with the period equal to the transition delay of its combinational part may not work properly due to the interference of signal transitions from earlier clock cycles still propagating on internal nodes of the circuit.

Theorem 4.1 in [DKMW94b] says that the transition delay τ of a combinational circuit is a valid clock period only if it is greater than half of the topological delay ω , $\tau > \omega/2$. The proof can be summarized as follows:

Assume that vector v_{-1} is applied to the PIs at time $-\tau$ and that a second vector v_0 is applied at time 0 . Let v_{-1} cause an event e_{-1} still propagating somewhere in the circuit at time 0 when event e_0 caused by v_0 appears

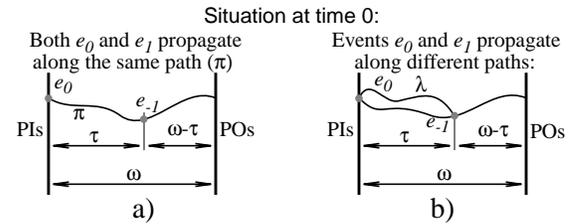


FIGURE 44: Transition delay validity as a minimal clock period

on PI. At time 0 , event e_{-1} has passed a partial path of length τ and it still has to travel for at most $\omega-\tau$ to reach a PO. Event e_0 may catch up with e_{-1} if it goes through length τ faster than e_{-1} reaches PO, written $\tau \leq \omega-\tau$. This cannot happen if both events propagate along the same path (Figure 44a). But event e_0 may catch e_{-1} propagating through a shorter path than e_{-1} , e.g., through a partial path of length λ , as shown in Figure 44b. In such a case λ must be shorter than τ to allow e_0 to catch up with e_{-1} , hence $\lambda \leq \tau$. As e_{-1} will reach PO after time $\omega-\tau$, event e_0 must reach it through λ before that time, $\lambda \leq \omega-\tau$. To avoid the interference, the reverse condition must hold true, i.e. $\tau \geq \lambda \geq \omega-\tau$, $\tau \geq \omega-\tau$, i.e., $\tau > \omega/2$.

2-2.5.2 Zero-width glitches

For instance, zero width glitches are caused on a 2-input AND gate by a rising and a falling transitions arriving at about the same time on the two inputs. Usually these are neglected, because such a glitch is absorbed by the inertial delay of the gate. However, simultaneous transitions on inputs may influence transition time (slew) on the output and increase the power consumption of the gate. An accurate timing analysis tool must account for transitions on gate inputs arriving simultaneously or within a narrow interval of time.

2-2.5.3 Standardization in delay and power calculation

Many problems in the electronics industry prompt many design automation vendors to develop new tools. Generally, each tool has different capabilities and thus for the same circuit it would calculate a different delay or power consumption which may cause many problems for designers. Therefore, in recent years there

has been an effort to provide a unique well interfaced method for delay and power calculation. Using the same gate and interconnect delay calculation allows timing verification tools to calculate the same circuit (topological) that is calculated by physical (placement, routing) tools. Furthermore, if all libraries are properly characterized by the library vendor (which is a basic requirement for static timing sign off) then the calculated delay is also very close to that of the manufactured circuit.

The main idea is to include the delay and power calculation for individual gates and the wire-load model (for interconnect delay calculation) in the library provided by the manufacturer. An EDA tool then uses that delay calculation for individual gates rather than only reading gate delays from the library. The major benefits are that the manufacturer prepares only one description for all EDA tools and the designer uses the same gate delay and power calculation method in all tools. The *Delay and Power Calculation Language* (DCL) described in [Broph97] is an attempt at standardization implemented by IBM which was later accepted as the IEEE standard.

To the author's knowledge there is no single widely accepted method for interconnect delay calculation. There exist several de-facto industrial and IEEE standard file formats for interconnect delay annotation (SDF) and parasitics extraction (e.g., SPEF). These allow EDA tools from different vendors to interchange annotated delays or RC-network equivalent of the interconnect network, but not necessarily to calculate the same delay. For example, the existing IEEE standard SDF 3.0 accurately defines delays, but does not define transition times, which will be included only in SDF 4.0. Similarly parasitics file formats describe networks of resistances and capacitances, but do not tell us how to compute delay along such networks. New formats are expected with RCL extraction of parasitics in deep sub-micron technologies.

2-3. Power verification methods

Power verification can be carried out at several levels. The most accurate (and the slowest) methods use transistor-level models, while currently the most common methods operate at the gate level. Methods operating at the RT and higher levels are starting to appear.

At any level (with more or less efficiency) one can use random simulation, probabilistic methods, and methods for computing upper bounds on power or switching activity.

2-3.1 Random simulation

A simple random simulation has many drawbacks as described in the section dealing with timing verification. Such a method can only compute a lower bound. Some effort has been made to bound the error of such methods.

Monte Carlo simulation is used in the McPower system [BNYT92] to estimate the average power. Based on experimental experience, it assumes that any random sample is a good representative of all possible vectors, and it uses statistical methods to compute how many input vectors are needed to obtain a certain standard deviation. Therefore, such a method is referred to as *statistical* [Najm94]. Despite the claims of “retaining the accuracy of deterministic simulation-based approaches” such methods do not guarantee any upper bound. The result is a triplet {a power value, a deviation *ERR* from the exact value, a probability that the error is smaller than *ERR*}. The approach cannot be used to compute the power profile of individual gates because it would require too many samples to obtain an acceptable error. In [XakN94], it was shown that many more vectors are needed to achieve the same relative error on the transition density of low-density nodes compared to high-density nodes. By using an absolute error instead of relative error the authors achieve considerable speedup with the same accuracy.

2-3.2 Probabilistic power estimation methods

Probabilistic switching activity/power estimation methods are pattern independent. Instead of simulating specific patterns, they compute how likely it is that a signal has a certain value or transition. The user supplies these probabilities for the primary inputs and they are then propagated through the circuit. A comprehensive overview and the related terminology appear in [Najm94] and in a shorter form also in [Najm95].

Signal probabilities describe the fraction of clock cycles in which signals have a stable value, e.g., $P_s(x)$ is the probability of value 1 on signal x . The gate models are very simple, for exam-

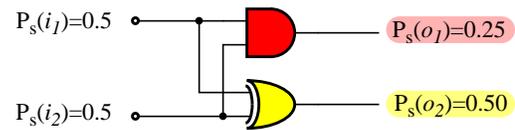


FIGURE 45: Signal probabilities - AND and XOR gate

ple, a 2-input delayless AND gate has $P_s(out) = P_s(i1) \times P_s(i2)$ as shown in Figure 45. Signal probabilities were used in [Ciri87].

Another definition uses *transition probability* $P_t(x)$ which for a delay-less gate is the number of clock cycles in which the steady-state value is different from the initial one divided by the total number of cycles. Under the assumption of signal values being independent between clock periods (*temporal independence*) and signals being independent of each other (*spatial independence*), transition probability can be computed from signal probability, $P_t(x) = 2P_s(x)P_s(\bar{x}) = 2P_s(x)(1-P_s(x))$. The

power consumed by a capacitive load is then $P_{av} = \frac{1}{2T} V^2 \sum_{i=1}^n C_i P_t(x_i)$, where T

is the clock period, V the power supply voltage, and C_i the capacitance at node x_i .

The disadvantage of such a method is that it does not consider gate delays, thus neglecting all glitches on the internal nodes of combinational circuits. Yet, glitches can typically account for 20%, and as much as 70% of the total power [Najm94]. The power caused by glitches is called *toggle power*. This disadvantage can be

resolved by power measures based on *transition density* [Najm93]. It is the average number of transitions on a node per unit of time, $D(x) = \lim_{t \rightarrow \infty} \frac{n_x(t)}{t}$, where n_x is the number of transitions on node x during the time interval t . The average con-

$$P_{av} = \frac{1}{2} V^2 \sum_{i=1}^n C_i D(x_i).$$

Computing functions for the propagation of transition/signal probabilities for more complex components can be based on the Shannon's expansion theorem

$$y = f(x_1, x_2, \dots, x_n) = x_i f(x_i = 1) + \bar{x}_i f(x_i = 0).$$

Using

simple rules for AND and OR gates,

$$P(y) = P(x_i)P(f(x_i = 1)) + P(\bar{x}_i)P(f(x_i = 0)).$$

An efficient implementation based on *binary decision diagrams* (BDD) is proposed in [Najm91]. However, density computation using the Boolean difference is valid only for a single transition on an input to the gate at a time. Therefore, in [ChRP94] the authors propose a method that accounts for simultaneous switching activity using *symbolic probabilities*. Another method using Boolean differences of higher orders is proposed in [MBOI95]. To handle unit delay gates, it defines *glitching sensitivity* as a fraction of the number of possible sequences of transitions on inputs to the gate where a glitch can occur (Figure 46). Experimental results are provided for several small circuits (up to 93 gates).

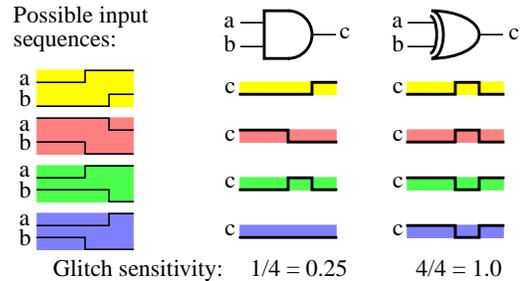


FIGURE 46: Glitching sensitivity of 2-input AND and XOR gates

A method which does not assume *temporal independence* is proposed in [NBYH90]. It uses probability waveforms. Such a waveform specifies the probability of signal values in

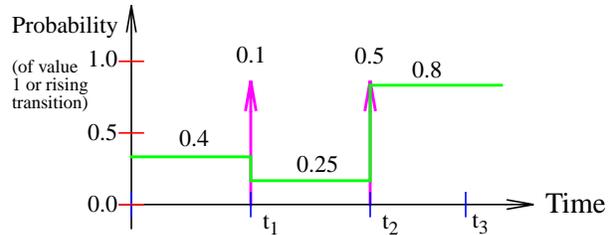


FIGURE 47: Probability waveform

time and the probability of transitions, as shown in Figure 47. User-specified input waveforms are propagated in an event-driven manner through the circuit. The probabilities of rising transitions may seem to be redundant and even incorrect in Figure 47: logically probability of a rising transition at t_1 is the probability of zero before t_1 multiplied by the probability of one after t_1 , $(1-0.4) \times 0.25 = 0.15$. The authors claim that by annotating a different probability (0.1) they can introduce temporal dependency.

Transition density and equilibrium signal probabilities are used in [Najm93]. The authors derive a gate model for transition density propagation based on Boolean difference and use BDD (binary decision diagrams) for implementation. However, it assumes spatial independence and thus cannot cope with toggle power. The authors claim that for sufficiently large circuits and a “good” choice of module partitioning, the local reconvergent regions are mostly within a partition. The results of computing the transition density on the ISCAS-85 benchmarks have errors ranging from -41% to +30% compared to a simulation of 1000 transitions on every input node (on the average).

The concept of *transition density* was revised in [Kapo94]. The authors give examples of two small reconvergent circuits where the original method (which does not handle reconvergence) in one case overestimates transition density by 100% and in the other underestimates it by 50%. The authors use the same definition for computing the transition density based on Boolean difference, but its implementation based on BDDs is more efficient. They define a new BDD operation called “DIFFERENCE”. The more efficient implementation allows to compute Boolean differ-

ence for larger subcircuits. They partition the circuit into macrogates - small partitions with as many correlated nodes inside one partition as possible. Each partition has a single output and limited number of inputs, which may be the primary inputs or the outputs of other partitions. Unfortunately, from the brief presentation in the paper, I could not understand how exactly it is done and why such partitioning is optimal. The algorithm may be of interest within our method.

A probabilistic method presented in [ChRC97] shows that switching activity can be very sensitive to the activity on certain inputs. In other words, a small change in the activity on these few inputs causes a big change in the switching activity in the circuit as a whole.

A more expensive technique than [Najm93] which can handle both temporal and spatial correlation is proposed in [GDKW92]. It is again based on BDDs. The values of each signal in time are denoted by a separate variable, $x_i(1), x_i(2), \dots$, defining thus a sequence of values on signal x_i . All internal nodes and outputs are functions of the user-specified sequences of values on primary inputs. The probability of a transition on node x_i from $x_i(k)$ to $x_i(k+1)$ is the probability of the Boolean difference of two consecutive values being 1, i.e., $x_i(k) \oplus x_i(k+1) = 1$. The total number of transitions per clock cycle is the sum of the probabilities of all transitions on a signal. Dividing it by the clock period yields the transition density. The size of the corresponding Boolean expressions is enormous, and thus even some moderately-sized circuits (those with many internal glitches) cannot be handled, in spite of the efficient BDD representation. Another accurate, but very expensive method based on BDDs was proposed in [TsPD93].

All the above mentioned methods analyze combinational circuits only. They assume that the incoming input vectors are independent of each other. They thus make the assumption that all states of the sequential circuits have an equal probability of occurrence. The examples shown in [MoDL94] reported errors of up to 56% if the sequential behavior is not considered. Let P_{in} be the vector of probabil-

ities on inputs to the combinational circuit, P_{out} the vector of probabilities on the state variables, and F the mapping of probabilities expressing the Boolean function of the combinational circuit. The method computes a greatest fixpoint of the equation $P_{out}=F(P_{in})$. The authors can obtain results within 3% of the exact probabilities computed by Chapman-Kolmogorov equations. “Exact” here means the probabilities that take into account the sequential behavior, which of course does not guarantee the exact power estimate.

In [NaGH95], it is shown that $P_{out}=F(P_{in})$ may not have a unique solution. Unlike the previously mentioned methods, the method does not need the assumption of a *Markov FSM* (the next state depends on the present state only, not the past ones). It simulates the circuit with user-specified inputs to the FSM, collects statistics, and stops after a number of cycles determined from the maximum allowed statistical error specified by the user. Basically, it is an extension of the method in [BNYT92] for sequential circuits.

A new analytical model for handling spatio-temporal correlation was proposed in [MaMP94]. Sequential behavior is modeled by a one-lag Markov chain. That means that the next state depends on only one previous state, no other history is taken into account. Signal correlation is limited to pair-wise dependence expressed by conditional probabilities. It is very memory expensive even if a BDD representation is used.

2-3.2.1 Interval gate delays and power analysis

The switching activity of a circuit can vary substantially with the delay assignment to the individual gates. Therefore, it is important to consider interval gate delays.

A method for computing a loose upper bound and a heuristic-based estimate of transition density was proposed in [NajZ95].

The authors give an example of a

circuit where the transition density on a node changes by three orders of magnitude under different gate-delay assignments. Such a result is not surprising, because the node where the phenomenon was observed is the exit line of 6 reconvergent regions¹. When the delays along reconvergent paths differ by less than the inertial delay of the closing gate, then transitions on the output of such a gate cancel out. However, a slight change of the delays in the reconvergent paths can cause the gate to propagate all the transitions. The ratio of the transition density for any two assignments of gate delays is then bounded only by the topology and the function of the circuit. A straightforward example with the possible ratio of infinity is shown in Figure 25 on page 34.

The upper bound computed in [NajZ95] neglects function and assumes that every transition propagates through all gates on a path to the output. Each signal is represented as a histogram of possible transitions within a time interval (Figure 48). Such a simplification allows the authors to use interval delays as shown in Figure 49. The number of

transitions N_{tr} within an interval is bounded by the inertial delay of the gate: $N_{tr56} \leq (t_6 - t_5)/d_i$, where d_i is the inertial delay of the gate.

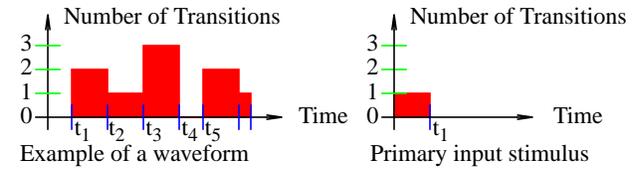


FIGURE 48: Signal representation by histogram of number of transitions

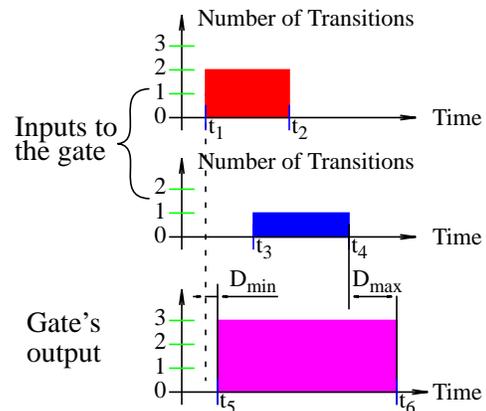


FIGURE 49: Transition density waveforms and the gate model

1. To be exact, 6 secondary reconvergences of one reconvergent region. Note that a secondary reconvergence can be viewed as a “small local reconvergent region” inside another reconvergent region.

The upper bounds on transition density in the ISCAS'85 circuits computed using such methods are between 140% and 281% of the lower bounds

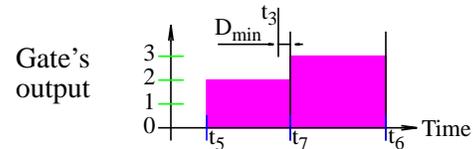


FIGURE 50: Less pessimistic transition density

obtained by simulation. Actually the waveforms as they are defined in the paper can accommodate even more accurate gate models than proposed, but at the expense of an increased number of intervals. It is clear that there can only be two transitions on the output of the gate between t_5 and $t_7=t_3+D_{min}$, as shown in Figure 50. Before t_3 there were 2 transitions on the first input and none on the second input, thus on the output there cannot be more than two transitions before t_7 (t_3 delayed by the minimum gate delay D_{min}).

The authors propose heuristics based on the idea published in [MBOI95]. They assume an equal distribution of transitions on gate inputs and thus gates with controlling values statistically propagate only some transitions. For AND and OR gates it is 0.75, because 3 out of 4 input combinations of transitions cause a transition on the output. If they assumed equal probability of static signal values, the ratio would be 6/16.

A method based on a similar idea was proposed in [VaSM93]. It neglects the gate function and can handle interval delays. Waveforms are represented at the level of intervals of uncertainty where transitions can occur; separate intervals are used for rising and for falling transitions, called *activity waveforms*. An activity waveform of a rising transition is a Boolean representation of a presence of a rising transition, has two values, present or not present. But it also respects the finite time of the rising transition, which makes it a piece-wise linear signal (Figure 51).

The current corresponding to an activity waveform is derived by fitting triangular current pulses into the activity waveform with respect to the inertial delay of the gate. For simplicity, such a

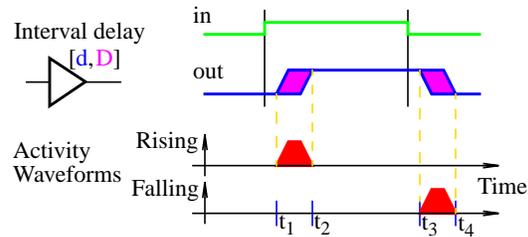


FIGURE 51: Activity waveforms

set of pulses is then approximated by one square pulse of current of the corresponding amplitude. A heuristic based on the fact that two subsequent gates on the same path cannot switch at the same time (due to the same event) was proposed to make the estimate of the power supply current less pessimistic.

The effect on accuracy of using different gate delay models for power estimation in sequential circuit was analyzed in [HsRP97]. The authors compared the accuracy of estimation of a lower bound on peak power in sequential circuits using the zero delay model, the unit delay model, a model that calculates delays from fanout, and a VLSI library delay model. They concluded that when the switching activity is low, then any non-zero delay model is sufficient, thus using a simpler gate delay model results in shorter CPU time without any loss of accuracy. When the switching activity is high, then the estimates differ considerably for different delay models.

2-3.3 High-level approaches to power verification

This section focuses on the optimization of gate level networks, power estimation at a higher-than-gate level, and the design for low power consumption. A survey of optimization techniques targeting low power VLSI circuits [DevM95] serves as the basis for our review.

2-3.3.1 High-level power analysis

The only accurate estimate of dissipated power can be obtained at low levels, knowing the transistor or at least the annotated gate-level description. Reasonable power models can be obtained if there exists some knowledge of the final imple-

mentation, such as restrictions on libraries (e.g., limits on gate sizes, use of components of regular structure) and design rules (e.g., maximum fanout). A power estimate can be obtained by estimating the total capacitance charged when a module is activated, or knowing the power consumption of individual modules, or by using a simple count of the components of the circuit. A list of references can be found in [DevM95].

A method estimating circuit area and power described by Boolean equations was presented in [NeNa97]. The authors transform multi-output Boolean equations into single output gates. Capacitance and power is then estimated from the gate count.

Architecture-level analysis called the *dual bit type* method (DBT) was proposed in [LanR95]. It attempts to eliminate a major disadvantage of the previously published architectural methods which estimate power by simple counting of modules, gates, or transistors within a module. Such an estimate can be expressed as a function of the word length (bus width) N as $P = C_u N^2 V^2 f$, where P is the total power, C_u the capacitance coefficient, V the power supply voltage and f the clock frequency. The DBT method employs the typical approach of architecture-level methods: A component model is assigned parameters obtained by simulation. The use of these parameters (mainly C_u) results in a more precise power estimate during the analysis itself. To achieve higher accuracy, DTB uses a vector of effective capacitances rather than a single value, each characterizing one type of the components used. E.g., one constant can describe output buffers, another one a 2-D computational matrix, etc. The effective capacitance of the module is then $C_u = C_0 N_0 + C_1 N_1^2 + \dots$, where N_i is the number of cells characterized by the capacitance constant C_i found in the module. However, a major improvement is obtained by dividing signals into two types: low- and high-order signals; therefore, the name “dual bit type”. It comes from the understanding that low-order bits switch very often and almost randomly while high-order (and sign) bits less often. A bit vector is described by a template, separating the sign, the low- and the high-

order bits. For each group, a different distribution of transition activity is used. Multi-function units such as ALUs are taken care of by having a unique set of capacitances for each function. These capacitances are switched in or out by the values on the control lines of the unit. The component parameters are extracted by model fitting techniques from simulation results. The power analysis proceeds as follows: First, an RTL description of the circuit is analyzed and modules are assigned appropriate coefficients from tables. The user supplies data (e.g., a program) and the control vectors for which the power analysis should be done. Signal activities are derived by a functional simulation of the RTL netlist. Based on the derived statistics, the modules are then decomposed into data and sign regions. Using the capacitance coefficients and the transition activity, the effective capacitance of each module is computed.

The theoretical limits on power savings by encoding bus signals were derived in [RaSH97]. The authors studied entropy (information content) of the transferred symbols and analyzed various encoding schemes.

One can also improve performance by combining a low-level and a high-level method to achieve a reasonable balance between accuracy and CPU time. In [BMMP97] simulation at high-level runs by default and when needed the low-level simulation is started. The start and stop criteria are derived using statistical estimates.

2-3.3.2 Optimization of circuits for low power at the circuit level

Two basic techniques can be adopted for optimization of circuits at the transistor level - transistor structures of complex gates and transistor sizing. More complicated gates than simple AND or OR can be implemented in many ways depending on the interconnection of the transistors. The structural modifications of the gate can be done considering power, delay, or both at the same time. Also, the order of the gate inputs is significant because different inputs may have different delays.

Transistor sizing is a technique which replaces fast transistors by slower, less power consuming transistors. The choice of the transistors which will be replaced by slower ones is done within the *slack* of the individual gates to avoid a change of the timing properties of the circuit. It is not as simple as it seems, because even if the circuit delay is not changed, the modified timing can introduce some internal glitches leading to higher power consumption. The timing method for computing the slack must be very carefully selected.

The method described in [BHMS94] operates at the gate level and assumes that multiple implementations of the gates exist as *library modules*. The decision on which one to choose is based on the arrival times on gate inputs and the slack time. The delay of a gate can be increased up to its available slack. From the description in the paper, it is not clear whether the slack computation is based on the knowledge of the exact delay. Experimental results for circuits of sizes ranging from 96 to 306 gates show power savings between 0.5% and 38%.

A transistor sizing tool was proposed in [DuNR94]. It uses a global image of the circuit in terms of cost functions based on the topological delay, area and capacitance, to minimize all delay, area and power consumption.

2-3.3.3 Optimization of circuits for low power at the logic level

Don't-care optimization relies on the fact that there exist many implementations of an incompletely specified logic function.

Boolean networks with zero delays were analyzed in [ImaP94]. The main idea is a local modification of the internal Boolean functions to lower the power of a block without increasing the power consumption of other blocks. Experimental results achieved about a 10% improvement on small circuits, but the method does not handle toggle power which can introduce errors several times greater. Neither spatial nor temporal dependences were considered. The algorithm first identifies Boolean variables that are don't-care (redundant). Then the least power-consuming implementation of the mod-

ule is chosen - the function of the module is implemented in such way that both the sum of the products of the fanin of each input and its switching activity are minimal. The switching activity on the inputs is obtained from the modules which generate them.

Switching activity can also be reduced by *path balancing*. Balancing a path delay is done by inserting delay buffers in short paths so as to equalize delays. It is a question of compromise, because the buffers will consume additional power. For more details see [LemS94].

Power consumption can also be reduced by minimizing the number of drivers (gate outputs) in the circuit. Boolean expressions can be factored to reduce the number of literals and thus the number of gates to implement the expressions. E.g., $ac + ad + bc + bd$ implying 5 drivers can be expressed as $(a + b)(c + d)$ implying 3 drivers. For more details see [BrRS87, RoyP92]. An optimization targeting a specifically XOR circuits was described in [NaLi97]. It Optimizes XOR Boolean trees for minimum power. The authors achieved a considerable improvement compared to SIS. An optimization of AND Boolean trees was presented in [ZhWo97]. Multi-input gates are translated into two-input gates and then the Boolean tree is optimized for minimal transition probability on internal nodes of the tree.

Once a target technology is chosen, the optimized Boolean expressions are mapped onto technological library components to minimize area, delay, and power. Modern technology mapping methods use *graph covering algorithms*. For references see [DevM95].

Not only Boolean expressions can be optimized for low power but also the *encoding of states* has strong influence on the power consumption of the circuit. If, for instance, a transition between two states of an FSM occurs very often, it is preferable that the two states be encoded with the least Hamming distance to minimize the number of transitions on the flip-flops [RoyP92]. However, one must also consider

that the combinational logic for such an encoding might be larger [TPCD94]. A reduction of the *switching activity on buses* and thus in the whole data path logic can also be achieved by a smart encoding. One of the techniques in [StaB94] transfers either the data or its complement, whichever is closer in the Hamming metric to the previous value, on the bus. One bit is added to the bus to indicate whether it transfers the data or its complement. An *arithmetic unit* based on an encoding different from the usual 2's complement encoding was proposed in [Chre95].

Retiming repositions flip-flops in a sequential circuit to minimize the clock period [LeRS83] or power dissipation [MoDG93].

Power can be also saved by switching unused blocks off. It is the same technique as used at a high level by portable devices but applied at the circuit level. E.g., a notebook computer *switches off power* of the hard disk when there is no request for a certain period of time. Behavioral analysis can identify blocks of gates/registers which either are not used or need not be updated in the given clock cycle. In the former case the power is switched off, in the latter case the clocks are gated.

Switching off power of some components can be done on-line. In [DevM95], this is demonstrated on an example of the arithmetic operation “greater”. The authors compare the most significant bits (which costs 1 XNOR gate only) and if they differ, the comparator does not receive the other bits, thus eliminating many transitions. References to several other algorithms identifying subcircuits which can be gated or switched off also appear in [DevM95].

2-3.3.4 High level power optimization

High level behavioral descriptions are usually specified using data and control flow graphs (CDFG). A basic optimization is the reduction of the number of control steps using slower and less power-consuming blocks to achieve the same performance. This is a trade-off between reducing the power (by slowing down the clock) and increasing the power due to additional capacitances [CPMR95].

Once a CDFG is optimized, then if there exists a choice the components of various delay/power ratios can be selected to implement the operations [GoOC94]. The allocation of registers and processing units can influence the amount of data transferred along data paths, and thus reduce the power consumption due to data transfers [RagJ95].

2-3.3.5 System and software level power optimization

The issue of power consumed by a processor executing a certain program became especially important for embedded computers. A method minimizing power consumed by a processor executing a program was presented in [MonD95]. For existing CPUs the power consumed while executing individual instructions or their sequences can be measured [TiMW94b] and later used to optimize the program for low power consumption [TiMW94a]. It is also clear that the choice of the algorithm influences both the runtime and the consumed power. Code optimization such as register allocation and the order of operations is very important too [OngY94]. It has been shown that the order of operations has no great impact for a general purpose processor but can be important on a small DSP processor [DevM95].

2-3.3.6 Synthesis for low power

Behavioral synthesis from a VHDL subset targeting low switching activity was proposed in [KKRV95]. The authors use a profiling tool to simulate the data flow graph with user-specified input patterns. The synthesizer is supported by a library of RTL components parametrized by area, delay, and average intrinsic switching activity. The parameters are obtained by simulating several layout-level instances of the components of different word sizes. *Least square regression* is used to find a general equation for an n -bit component. In a similar way, by long simulations, the *intrinsic switching activity* is obtained. It expresses how many circuit nodes would switch when there is an event on the input. The profiler examines the flow graph and determines how many times each node of the graph was executed, how many

times each edge was traversed, and how many times the value on each edge changed. The annotated graph is then synthesized.

The synthesizer performs scheduling, register optimization, interconnect optimization and generates the control logic. A set of valid schedules is determined. For each generated schedule, the switching activity, the minimum clock period, and the average power are estimated. The paper describes in detail the algorithm for estimating switching activity.

In [CGSS97], the authors proposed a method suitable for the synthesis of sequential circuits for low power. It takes into account input-pattern dependency.

2-3.4 Low-level approaches to power verification

This section deals with gate models and methods for power estimation at the transistor level.

The simplest and the most accurate way is a Spice simulation using exact input waveforms which results in a complete distribution of electric current in the transistor network. This is then easy to convert into power consumption or any kind of current profiling. There are several drawbacks, however:

- Spice simulations are slow,
- The method cannot handle interval delays,
- It cannot handle sets of input waveforms.

A model was proposed in [LiLS94] to estimate power dissipation. On several benchmark circuits of sizes from 6 to 209 gates, it achieved 10% accuracy compared

to Spice, and it was about two orders of magnitude faster. The method addresses two major phenomena - charging of the internal capacitance without a change on the output, and input to output *capacitance feed-through* (Figure 52).

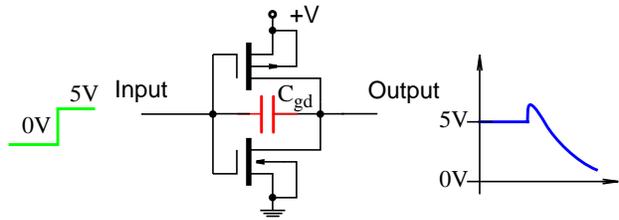


FIGURE 52: Capacitance feed-through effect

The effect of internal and coupling capacitances is captured by a *state transition graph for power estimation* (STGPE). The STGPE for a 2-input (resp. m -input) NAND gate has three ($m+1$) states. The states are

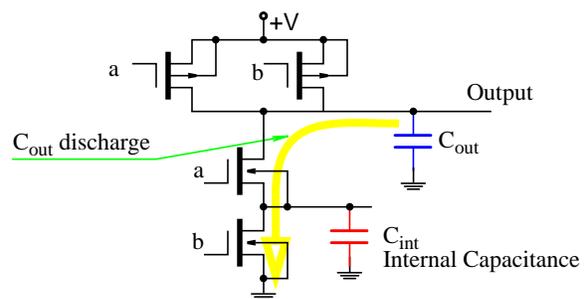


FIGURE 53: 2-input CMOS NAND gate

expressed by the following state variables: the voltages on the input and the output capacitances, each 0 or 1. For example, in Figure 53, the state ($output=0$, $internal=1$) does not exist as the output discharging path passes through the internal node (Figure 53). Therefore, the 2-input NAND gate has only 3 states, even when it is characterized by 2 state variables. Each transition of the graph is assigned an input pattern which may cause it and a Spice-derived energy consumption caused by the transition. The algorithm then finds the STMP edge activity from the input signal probabilities under the temporal independence assumption and computes the power consumption.

A transistor-level model of a *CMOS inverter* needed for computing the supply current and delay presented in [NabR92] is 3 orders of magnitude faster than HSpice simulation and achieves accuracy to within 12% of HSpice. The inverter model was used for power estimation of combinational circuits in [NabR94]. It accounts for relative times of transitions on inputs. Static CMOS gates are collapsed to inverters by replacing groups of transistors by equivalent transistors.

A current model suitable for accurate simulation was proposed in [WaFF94]. It takes into account the capacitor charging current, the short-circuit current, and toggle power. The authors express toggle power by the power consumed by charging internal capacitors without an observable transition on the output.

An analysis at an even lower level was presented in [ChaS94]. The model of a NMOS transistor includes such details as voltages, output conductances, high-frequency transmittances, and noise, thus allowing for more accurate estimation of the consumed power.

2-3.5 Pattern independent methods for computing an upper bound on power dissipation

These methods try to find upper limits on switching activity, voltage drops, and the average power dissipation. Most of the methods already mentioned are either input-pattern dependent (like simulation) or compute only an estimate. A different approach is needed to obtain some bounds - to be able to guarantee that for the given libraries and operating conditions, a manufactured chip would never exceed certain power parameters.

2-3.5.1 Uncertainty waveforms and partial input enumeration

An input-pattern-independent method based on *uncertainty waveforms* was reported in [KrNH92, KNYH93, KrNH95]. This method is the closest one

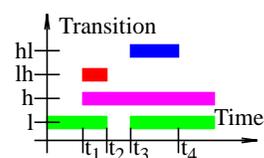


FIGURE 54: An uncertainty waveform

to our approach. There, every circuit node carries sets of intervals during which a certain transition/value activity may occur. It distinguishes stable zero (l), stable one (h), rising (lh), and falling (hl) activities. E.g., the waveform shown in Figure 54 represents a signal which is 0 till t_1 , changes to 1 during $[t_1, t_2]$, stays at 1 till t_3 , and can change back to 0 during $[t_3, t_4]$ or stay at 1. Such a representation can easily accommodate sets of waveforms. Initially, every primary input is assigned the set of all 4 possible transitions at time 0 (the interval $[0,0]$). The tran-

sition sets are propagated only forward towards the primary outputs. To keep the computational complexity manageable, there is a limit on the number of intervals within each waveform. If during the propagation this limit is exceeded then the two closest intervals are merged. A case analysis on primary inputs only (referred to as MCA, Multi-Cone Analysis) is used to improve the initial loose bound by enforcing some spatial correlation. In [KrNH92], the computed waveform sets are converted to current waveforms and the *peak current* is reported.

In [KNYH93], the capability of the case analysis is extended to more than one circuit node at a time (*Partial Input Enumeration, PIE*). Three heuristics are provided for the selection of the circuit inputs where the analysis is performed: **H1static** - analysis on each input node alone is done, the improvement measured, and the nodes are ordered by the improvement; **H1dynamic** uses H1static after each enumeration to select the next node; in **H2** the list of input nodes is ordered according to the largest number of gates reachable from the given node.

The most recent and most comprehensive work [KrNH95] adds the calculation of the *maximum voltage drops* in the power and ground busses, and modifies the heuristic H2 to use a list of nodes ordered by decreasing fanout.

2-3.5.2 Power estimation based on constraint resolution

This is the place where the method we propose belongs to. It is based on an idea published in [CerZ94], which describes a circuit as a system of inequalities (constraints). For more details see Section 3-1 on page 83.

Computing power dissipated by individual gates and the whole circuit is very important, but not sufficient to guarantee the proper operation of a chip. In Section 2-3.7, we discuss a few methods for the analysis of current distribution on a chip. But before that, we briefly summarize in the next section the power-related issues specific to sequential circuits.

2-3.6 Power estimation in sequential circuits

An existing power verification method for combinational circuits can be used for verifying sequential circuits under the assumption that the states of the FSM are distributed randomly, i.e., assuming no temporal correlation between successive patterns on the pseudo primary inputs. For more accurate estimation, temporal correlation must be used.

In [ChouR96] the authors explain the existence of *near-closed sets*. A near-closed set is a set of FSM states into and from which it is very unlikely to get during random simulation. The authors discuss how the near-closed sets influence the results of statistical power estimation methods and provide a method on how to identify and use these sets in a Monte Carlo analysis of power estimation.

A method that focuses on large sequential circuits was described in [KoNa97]. It is a statistical method returning a bound on the transition activity and a level of confidence, i.e., the probability that the upper bound is correct.

2-3.7 Methods analyzing current distributions on a chip

2-3.7.1 Voltage drops

One of the major causes of malfunction of digital integrated circuits due to high power consumption are *voltage drops* on power buses.

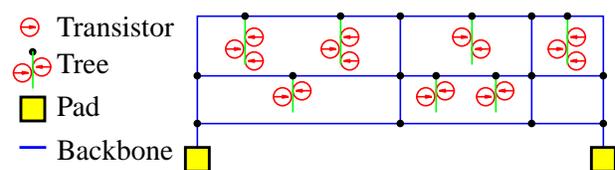


FIGURE 55: VLSI power supply network

A voltage drop is caused by a current pulse in the non-zero resistance power distribution network. A typical *power bus* network used in VLSI circuits consists of backbones with attached tree networks (Figure 55). An example of the resistance, current and corresponding voltage drop distributions in a tree structure is shown in Figure 56.

A method for computing voltage drops along individual segments of the network was proposed in [StaH90]. Knowing the network structure, the resistances of the individual

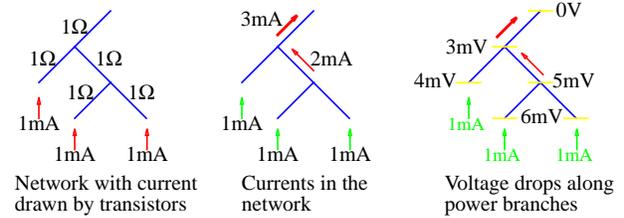


FIGURE 56: Voltage drops in a power network tree structure

branches, and the current drawn by the individual transistors, the method partitions the network into tree sub-structures, simple loops, transistors in series, and the remaining network. The regular structures are solved separately using dedicated algorithms. The remaining network is described by a system of equations and solved using sparse-matrix techniques.

2-3.7.2 Current distributions

Not all switching activity verification methods are able to provide a current profile, i.e., the current drawn by the circuit and the individual gates and blocks as a function of time. The current distribution can provide clues of problems such as the voltage drops in power busses discussed in the previous section, and capacitance coupling or noise coupling with an analogue part of the circuit. A method which computes *Expected Current Distributions (ECDs)* was presented in [CipR96]. The authors assume a sequential circuit without temporal correlation and with known switching activity obtained by a probabilistic method. They use a gate current pulse model to compute ECD for a gate and provide a framework for combining gates and blocks together to compute the overall ECD. ECD is not a simple current profile, as it also contains statistical information about the current drawn in each interval of discrete time, such as variance, covariance, etc. This enables the method to obtain RMS power for each gate comparable in accuracy to a statistical method (simulations run over many clock cycles), but about 100 times faster.

2-4. Literature survey - conclusions

The most important modeling characteristics in timing and power verifications methods are the following:

- Interval pin-to-pin gate delay,
- Component parameter correlation.

A major problem for both timing and power verification techniques is to determine whether transitions can propagate from inputs to outputs. They do not propagate all the way to POs along so called false paths. The problem of delay calculation is to find the arrival time of the latest transition on the circuit outputs. The problem of power verification in CMOS circuits is to find how many transitions and when they can occur in the whole circuit. The major source of dissipated power is the charging of both the internal and the external (wires and inputs of connected gates) capacitances.

Methods for timing verification can be based on the topological delay (found in many commercial design systems¹) or on simulation. The former is too pessimistic, while the latter is input-pattern dependent. An exact method has been proposed of far lower complexity than previous ones, yet it still cannot handle large circuits (note that the problem is NPH). Therefore, many more or less accurate heuristics have been proposed. In most cases they trade off accuracy for computational speed. There exist very few accurate methods treating component delay correlation.

Most of the existing power verification methods are based on probabilistic analysis. However, in principle, that does imply the computation of an estimate rather

1. Among the most difficult problems in commercial static timing verification tools are the support of many libraries, many clocks in the design, loops in combinational logic, and designs with both level- and edge sensitive latches. These, however, are beyond the scope of this thesis. In general, a sophisticated algorithm can be applied to translate all the timing checks into many one clock-two flip-flop setup/hold checks.

than an upper bound on power dissipation. Some of the simple methods even do not consider toggle power, as they use zero gate delay models. Input-pattern independent methods for computing upper bounds at the gate level are based on propagation of waveform sets. Power estimation methods at higher levels of abstraction and direct high-level synthesis targeting low-power start to appear.

Dependence between states of flip-flops in subsequent clock cycles in sequential circuits (temporal dependence) is a problem in both timing and power verification. Several timing methods handling temporal correlation have been proposed. Most of the existing power estimation methods assume temporal independence.

CHAPTER 3

Bounding switching activity using constraint resolution

In this chapter we describe the core of the switching activity estimation method based on constraint resolution. We show how to build the constraint system from a circuit and how to use it for computing an upper bound on the switching activity in the combinational part of a synchronous sequential circuit. The case analysis procedure as a way to tighten the initial loose upper bound is presented next, followed by the heuristics involved in the case analysis and other enhancements to the method such as learning and reconvergence analysis.

3-1. Circuit modeling - the constraint system

In this section we explain the abstract waveform representation used to preserve information about the occurrence of signal transitions (Figure 57).

3-1.1 Waveform classification

A *real waveform* is a mapping of real numbers (time) to Boolean, $rw: R \rightarrow B$ (similarly as in [LaB94]). The collection of such mappings forms the *space of real waveforms* RW . We operate in discrete time, a unit

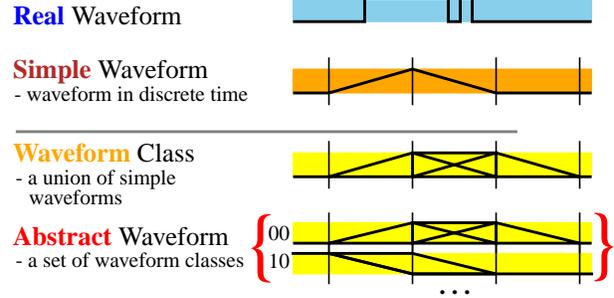


FIGURE 57: Abstract waveform

of which corresponds to the smallest resolvable interval of real time relative to the circuit technology. The intervals $[r_i, r_{i+1})$ are denoted by integer numbers $i \in \mathbb{N}$ such that $R = \bigcup_{i \in \mathbb{N}} [r_i, r_{i+1})$, $r_i < r_{i+1}$, and r_i, r_{i+1} are rational numbers. Individual transitions occur inside these intervals. Multiple transitions within an interval cannot be distinguished.

The shape of a real waveform rw inside the intervals is thus abstracted and is described by the initial and final values only, $rw(r_i), rw(r_{i+1})$. Such a couple of Boolean values is a *transition* t , defined as

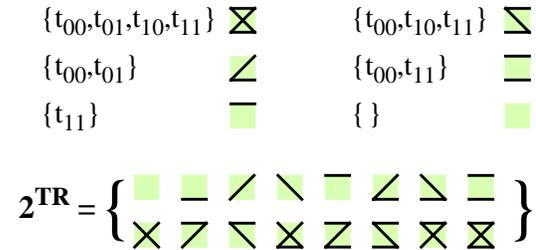


FIGURE 58: Set of transitions

$t \in TR = \{0, 1\} \times \{0, 1\}$, where TR denotes a *transition space*. The couples $(0,0)$ and $(1,1)$ represent stable values 0 and 1 , respectively. In general, we denote a transition $t = (x, y)$, $x, y \in \{0, 1\}$, as t_{xy} . Boolean operations over transition space are defined over the initial and final values, e.g., the AND operation is t_{vx} and $t_{yz} = t_{(v \text{ and } y)(x \text{ and } z)}$, similarly for OR and NOT.

The above definition of a transition space by Cartesian product of Booleans exactly corresponds to the *P-set* defined in [Haye86], our $TR = B_2^2$. A set of transitions that describes uncertainty (caused by presence of more than one simple

waveform) corresponds to the *U-set*. Our method thus combines P-sets and U-sets similarly to many logic simulators. Unlike some simulators that define value sets in ad hoc fashion and potentially incompletely, our transition space is fully defined. As shown in the previous paragraph, we derive the operations on transitions from the basic set $\{0, 1\}$ as required by [Haye86]. Similarly for transition sets (see DEF 7 on page 89). Deriving operations for transitions sets was straightforward since we consider the simplest basic set of $\{0,1\}$. For larger basic sets (necessary to include drive strength or tristate drivers) a rigorous algebraic framework such as [Haye86] is needed. Then our transition set becomes a value of a multivalued logic thus making our method very generic.

A *simple waveform* is a mapping from integers to transition space, $w: N \rightarrow TR$, and the discretized waveform space W is thus a collection of such mappings which represent continuous waveforms, $(w(\tau) = t_{xy}) \Rightarrow (w(\tau - 1) = t_{vx}) \wedge (w(\tau + 1) = t_{yz})$, $v, x, y, z \in \{0, 1\}$. Let $w_1, w_2 \in W$ and $w_i(\tau) \in TR$ be the transition of w_i in the interval $[r_\tau, r_{\tau+1})$. Boolean operations on simple waveforms are then defined as follows:

- NOT: $not(w_1)(\tau) = not(w_1(\tau))$
- OP: $(w_1 \text{ op } w_2)(\tau) = w_1(\tau) \text{ op } w_2(\tau)$, where *op* stands for any dyadic boolean operator, such as for example OR:
- OR: $(w_1 + w_2)(\tau) = w_1(\tau) + w_2(\tau)$
- DELAY by a fixed value $d \in N: delay_d(w_1(\tau)) = w_1(\tau - d)$

Waveforms belong to one of four class types depending on their initial ($\tau \rightarrow -\infty$) and final ($\tau \rightarrow \infty$) values [CerZ94], as illustrated in Figure 59:

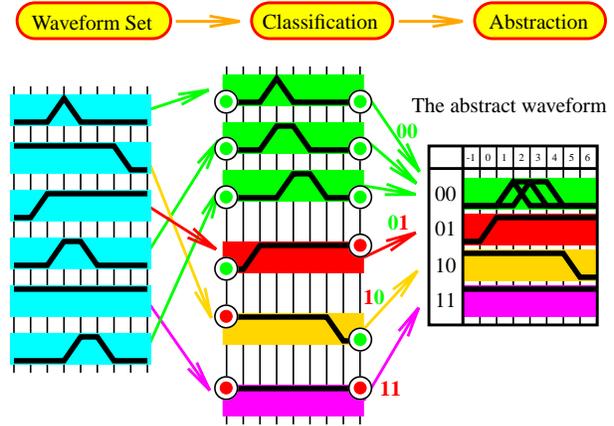


FIGURE 59: Waveform classification and abstraction

$$\text{Class type } xy \in C = \{0, 1\} \times \{0, 1\} \tag{DEF 1}$$

A *waveform class* wc_{xy} is an abstracted¹ set of simple waveforms of the same type xy , obtained by representing this set through the union of the transitions on the member waveforms at each time interval τ .

Finally, an *abstract waveform* aw (Figure 60) is a set of (up to four) waveform classes (at most one per class xy). Then $AW \subset C \times N \times 2^{TR}$ denotes the *abstract waveform space*, where 2^{TR} is the space of sets

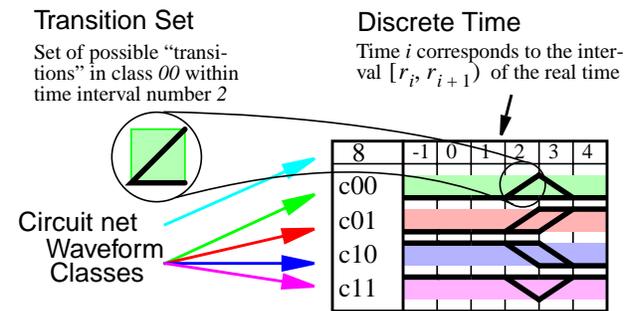


FIGURE 60: Graphical representation of an abstract waveform

of transitions, C space of class types, and N is discrete time. In other words, an abstract waveform is a set of transition sets; there is one transition set per interval of discrete time and class type.

1. *Abstracted*, because it contains less information than a pure set of waveforms would. It is the envelope over such a set.

It is important to ensure that the operations performed on these waveforms preserve *class validity*. A class is valid if it is continuous, i.e., the end point of every transition in interval i is a starting point of a transition in interval $i+1$, and vice versa. For example, the waveforms in Figure 60 are continuous, however if the class 01 in the figure contained a falling transition at $\tau=1$, it would not be continuous.

3-1.2 Operations over space of abstract waveforms

We have defined the operations intersection \cap , union \cup , AND, OR, NOT and DELAY over the abstract waveform space. The *union* of two transition sets is a standard set union. Let A and B be the transition sets, t a transition.

$$A \cup B = \{t: t \in A \vee t \in B\} \quad (\text{DEF 2})$$

The *union of two abstract waveforms* is defined as a union of the transition sets, separately for each τ and waveform class. Let X_{cd} , Y_{ef} be two waveform classes of type cd , and ef .

$$\begin{aligned} X_{cd} \cup Y_{ef} &= \emptyset \quad \text{when } c \neq e \vee d \neq f \\ X_{cd} \cup Y_{cd} &= \{\forall \tau \in N: X_{cd}(\tau) \cup Y_{cd}(\tau)\} \end{aligned} \quad (\text{DEF 3})$$

The definition of *intersection* of abstract waveforms must be based on the simple waveforms represented by the abstract waveform. The reason is that transition-set intersection done locally for each $\tau \in N$ could produce an invalid waveform. Simply put, such an operation would not be closed over the space of valid abstract waveforms.

The *intersection* of two transition sets is a standard set intersection. Let A and B be the transition sets, t a transition.

$$A \cap B = \{t: t \in A \wedge t \in B\} \quad (\text{DEF 4})$$

The *local intersection* \cap_L of abstract waveforms is a pure set intersection for each class and time interval τ . Let X_{cd} and Y_{ef} be two waveform classes of type cd and ef , respectively:

$$\begin{aligned} X_{cd} \cap_L Y_{ef} &= \emptyset \quad \text{when } c \neq e \vee d \neq f \\ X_{cd} \cap_L Y_{cd} &= \{\forall \tau \in N: X_{cd}(\tau) \cap Y_{cd}(\tau)\} \end{aligned} \quad (\text{DEF 5})$$

The local intersection can produce an invalid waveform even when both operands are valid as shown in Figure 61.

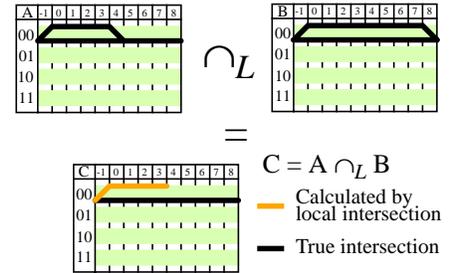


FIGURE 61: Local intersection

The *intersection of two abstract waveforms* is thus defined as the union of all simple waveforms contained in all the waveforms

being intersected. Thus the intersection of two abstract waveforms X_{cd} , Y_{ef} is

$$\begin{aligned} X_{cd} \cap Y_{ef} &= \emptyset \quad \text{when } c \neq e \vee d \neq f \\ X_{cd} \cap Y_{cd} &= \left\{ \bigcup_{w \in X_{cd} \wedge w \in Y_{cd}} w \right\} \end{aligned} \quad (\text{DEF 6})$$

where w is a simple waveform. Such a general definition is very hard to implement because it requires enumerating all simple waveforms in the waveform class. However, we are employing intersections only under limited conditions where Definitions 5 and 6 are equivalent. The intersection is used to compute conjunction of constraints, i.e., to update variables (Figure 61). Thus it is used only on abstract waveforms which are the result of operations preserving validity (\cup , AND, OR, XOR - will be defined later). The equivalence of Definitions 5 and 6 is preserved also when the case analysis algorithm (Section 3-2) imposes additional constraints because only complete waveform classes are being removed¹.

(DEF 7) The AND operation

Let X, Y be abstract waveforms, X_{ab} the waveform class of X of type ab , $X_{ab}(\tau)$ the transition set of X_{ab} at τ . Similarly for Y_{de} and $Y_{de}(\tau)$. The AND of abstract waveforms X and Y is defined by the Boolean AND function of the individual transitions as follows:

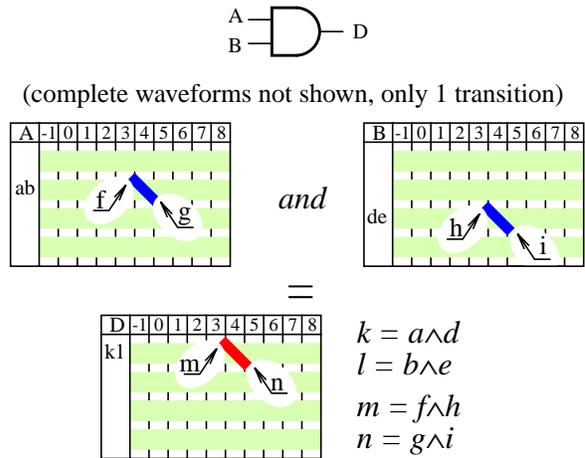


FIGURE 62: AND operation

$Z = X \text{ and } Y$ such that $Z_{kl} = \bigcup_{k = a \wedge d, l = b \wedge e} X_{ab} \text{ and } Y_{de}$, where at each τ , the

AND operation of transition sets is performed:

$(X_{ab} \text{ and } Y_{de})(\tau) = \bigcup_{\substack{t_{ij} \in X_{ab}(\tau) \\ t_{kl} \in Y_{de}(\tau)}} t_{ij} \text{ and } t_{kl}$. The operations OR, NOT and XOR are

defined in a similar way.

The transition analysis to be described in Section 3-1.3 views a gate as a relation between the signals on its terminals. The usual gate functions (AND, OR, NOT, DELAY) relating outputs to

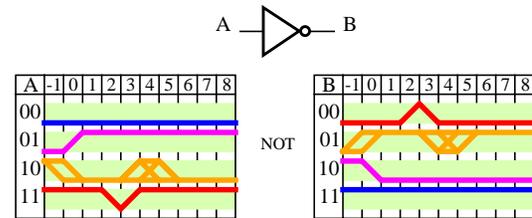


FIGURE 63: Example of NOT operation

inputs are used for forward propagation of waveforms. To propagate the effect of reduced waveform sets (due to case analysis) from outputs of gates to their inputs, we use partial inverse functions [Lhom93] illustrated below with AND^{-1} .

1. A simple but not necessarily brief proof can be based on proving that the resulting intersection (DEF 5) contains at least all individual transitions that (DEF 6) requires. Note that intersection does not generate transitions, only removes. It is obvious that fanout (duplication of abstract waveforms) preserves validity. Then it is sufficient to prove that removing one or more waveform classes during the case analysis on one or more nets of a reconvergent region results in a valid abstract waveform on the stem. E.g., the proof of continuity would show that for each transition t_{xy} at time τ there is a transition t_{yz} at $\tau+1$ on the stem.

(DEF 8) Partial inverse function AND^{-1} over abstract waveforms

The partial inverse function returns an abstract waveform on an input as a function of the other input(s) and the output waveform.

$X = Z \text{ and}^{-1} Y$ is such that $X_{ab} = \bigcup_{k=a \wedge d, l=b \wedge e} Z_{kl} \text{ and}^{-1} Y_{de}$, where at each τ

the AND^{-1} operation over transition sets is performed:

$(Z_{kl} \text{ and}^{-1} Y_{de})(\tau) = Z_{kl}(\tau) \text{ and}^{-1} Y_{de}(\tau)$. The latter is defined as

$Z_{kl}(\tau) \text{ and}^{-1} Y_{de}(\tau) = \{ t \in X_{ab}(\tau) \mid \exists t_1 \in Y_{de}(\tau) \exists t_2 \in Z_{kl}(\tau) (t \text{ and } t_1 = t_2) \}$.

(DEF 9) The DELAY operation

In this thesis we consider fixed delays. Let $\text{delay}(d)$ represent the DELAY operation of d time intervals and X an abstract waveform. Then $\text{delay}(d)(X)(\tau) = X(\tau - d)$ and $\text{delay}^{-1}(d)(X)(\tau) = \text{delay}(-d)(X)(\tau) = X(\tau + d)$.

All the above functions preserve class validity and always include at least all the simple waveforms the circuit can produce on every node, given a set of waveforms on its inputs.

The above definitions for two input variables can be generalized to M -input gates, but their implementation would be quite complex.

Presently, we employ structural gate models using two-input functions. Delays appear at the outputs of gates. Interconnect delays can be included in the gate if it drives only one gate input (fanout of one). They can be modeled by inserting a buffer (delay function in the constraint system) on interconnects with fanout greater than one.

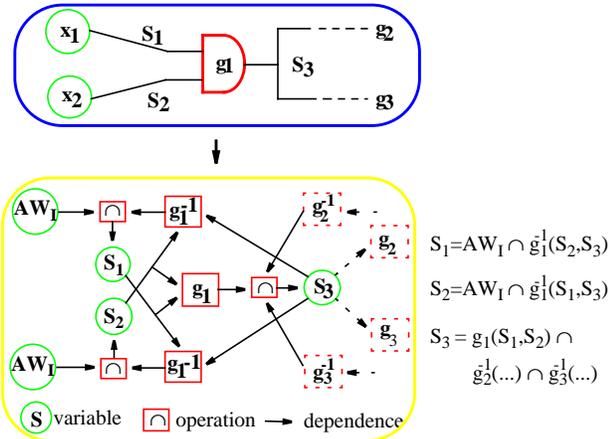


FIGURE 64: Gate network as a system of equations

It follows that every gate is effectively described by the forward and the partial inverse functions which introduce relations between the waveforms on the circuit nodes connected to the gate terminals. We can thus view the network as a system of node equations over abstract waveforms as shown in Figure 64.

3-1.3 Power analysis

The objective of a pattern-independent power analysis method is to find a (tight) upper bound on the average power dissipated by the circuit under a two-vector input stimulus. As a by-product we may determine the pair of vectors which yield that power. In our analysis we approximate power by the total number of switching transitions that occur in the circuit over a clock period. The transitions could be easily converted to power using the method in [KrNH94, NBYH90].

To obtain a tight upper bound, we must find the smallest possible set of abstract waveforms containing all possible waveforms in the circuit resulting from all possible primary input stimuli. The resulting abstract waveforms provide detailed information about the possible switching activity in the circuit.

We begin by assigning a complete set of two-vector input waveforms to primary inputs, let them propagate to primary outputs, and calculate the resulting switching activity estimate. Since the number of transitions in the constraint system is always a conservative upper bound over all possible input vectors, we approach the exact solution (the highest switching activity for a single pair of input vectors) by case analysis which proceeds by constraining some nets to a single waveform class at a time. The power estimate is then the maximum value obtained over all such cases. Since many constraints can be applied simultaneously to different nets, the result can be an inconsistent constraint system. This is detected by noticing that the abstract waveform becomes empty on a node and it triggers backtracking in the case analysis. The method is described in more detail in the Section 3-2.

3-1.3.1 Initial propagation and estimate

Initially, all nets carry the unknown waveform represented by the largest valid abstract waveform (all possible transitions in every interval). Then, primary inputs are assigned

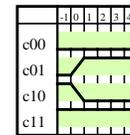


FIGURE 65: Primary input abstract waveform

the two-vector abstract waveforms consisting of classes *stable 00*, *stable 11*, *01* at time 0 , and *10* at time 0 (Figure 65). The initial propagation of the waveforms produces a very pessimistic estimate, since the waveforms on gate inputs are considered independent and waveforms of the same class are merged on circuit nodes. The resulting abstract waveform on a node is thus an upper envelope over the possible simple waveforms, i.e., it contains all the real transitions and, usually, some that are artificially created by the method.

3-1.3.2 Transition counting

The number of transitions occurring on a circuit node is taken as the maximum over its four waveform classes. The number of transitions within a class is obtained from a simple waveform containing the largest number of transitions contained in the class. We developed an exact linear-time algorithm to extract such a worst-case waveform.

An example is shown in Figure 66. The transition counting was considerably enhanced by the extraction of consistent sets of worst switching activity waveforms in [Shen96].

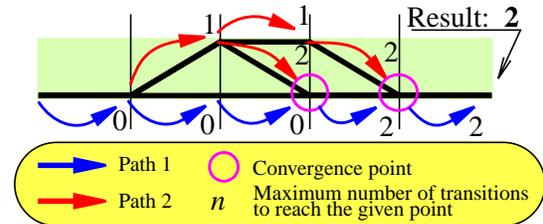


FIGURE 66: Identification of simple waveform with the largest number of transitions in an abstract waveform

The decomposition of a waveform class into all possible simple waveforms amounts to a traversal of a binary tree, a level of which corresponds to the time interval τ and the tree branches to the Boolean values. A path from the root to a leaf identifies a simple waveform that could be contained in the class. Exponential complexity of such a traversal is avoided by realizing that a waveform has to have the value 0 or 1 at any time. The tree can thus be folded onto itself so that, at any level, it has at most two nodes. The graph is then traversed by following two paths, at each level choosing the direction of a transition if such exists. The nodes with the same signal value (up to two, for each 0 and 1 values) are merged, assigning the resulting node the maximum number of transitions amongst the folded paths. The traversal is thus achieved in a time linear in the number of time intervals.

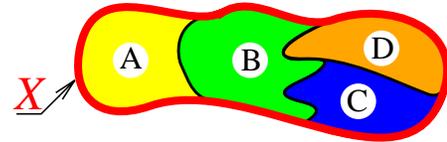
3-2. Case analysis

In this section we show how case analysis on waveform classes can be used to obtain a tighter upper bound on switching activity.

3-2.1 Principles of case analysis

The state of the constraint system before starting the case analysis describes all possible waveforms for all possible pairs of input vectors. This space is further refined by case analysis to identify the subspace with the test vector having the highest number of transitions.

Generally speaking, *case analysis (CA)* proves the satisfaction of a property by the satisfaction of all of its independent parts. The property X should be completely covered by the subcases,



The whole Property X is satisfied if all of its particular cases A , B , C and D are satisfied.

FIGURE 67: Principle of case analysis

$X = A \wedge B \wedge C \wedge D$, i.e., the X is satisfied if all subcases are satisfied.

A further requirement is that all the subcases are independent of each other. In the case of the signal representation by abstract waveforms, the waveform classes which distinguish waveforms by their initial and final values are the simplest criteria for dividing the verification problem into sub-cases. Since in the real circuit the value of a net is always unique¹ (either 0 or 1), then any two classes are independent of each other.

3-2.2 Case analysis on waveform classes

The finest resolution which can be achieved considering 4 classes is the division of waveforms found on a net into 4 cases as shown in Figure 68. The notation of $c00$, $c01$, $c10$, $c11$ has the meaning *class 00, 01, 10, 11*.

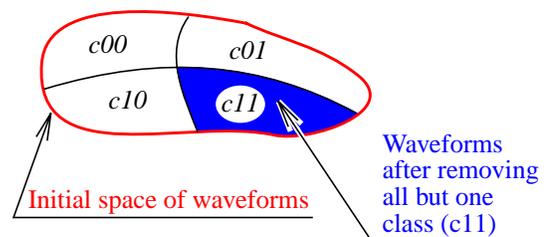


FIGURE 68: Case analysis on a single net

When more nets are involved, we also use the following notation: $a01$ is class 01 on net a .

1. Either 0 , 1 , Z , etc. - as many values we want to care about - but never two or more of them at the same time on the same net.

A further refinement is possible by continuing the case analysis on another net. The finest case is obtained by splitting of waveforms on the second net again into four classes, thus introducing 4 new cases for each of the previous ones, as depicted in Figure 69.

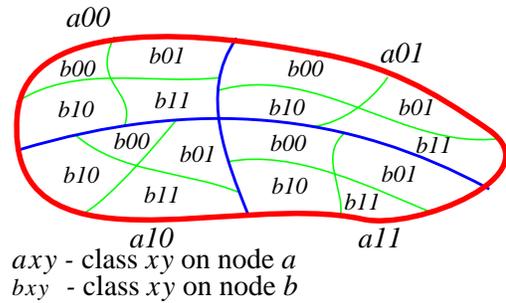


FIGURE 69: Case analysis on two nets

3-2.3 Case analysis in a circuit

Consider the example of a circuit which consists of a single AND gate. The circuit has two primary inputs and one primary output. The result of the forward propagation is shown in Figure 70. These waveforms represent the whole space of possible waveforms - the complete oval in Figure 70 or Figure 71.

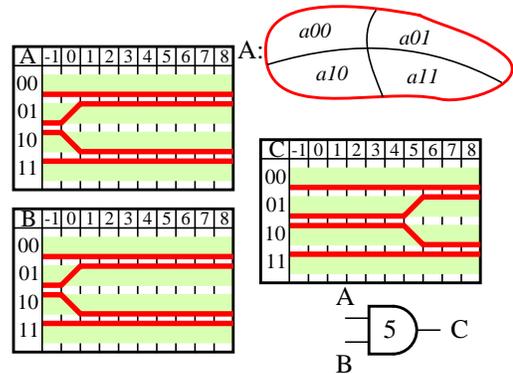


FIGURE 70: Initial state of the constraint system

The particular case with limited waveforms on a single net (A) is shown in Figure 71. The case analysis is usually further extended by limiting waveform sets on other nets. Once each net bears only exact waveforms rather than sets of waveforms, the propagation of waveforms is equivalent to circuit simulation. This is the case in Figure 72 when the

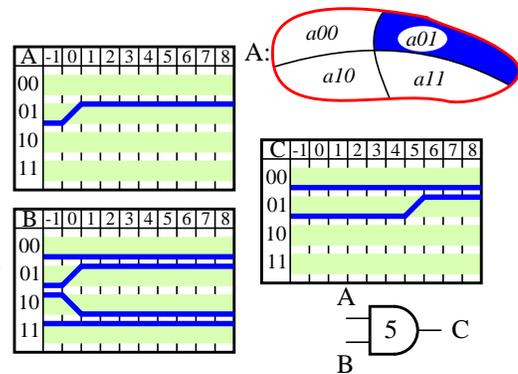


FIGURE 71: Constraint system for A=01

waveform set on net B is also limited to an exact waveform. Since all the primary inputs and thus all the circuit nets carry exact waveforms, the result corresponds to a simulation using the specific input waveform.

The case analysis is used to tighten the upper bound on the power consumption. The more we progress with the case analysis, the tighter the bound becomes, up to a simulation which gives a lower bound. It follows that case analysis provides a compromise between verification by simulation and the propagation of all possible waveforms.

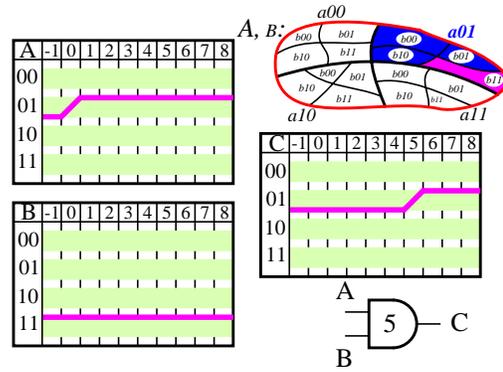


FIGURE 72: Constraint system for A=01, B=11

3-2.4 Decision tree

The overall behavior of the case analysis can be depicted by a *decision tree*, as illustrated in Figure 73. There, a node represents the state of the circuit (i.e., abstract waveforms, one per circuit net), and an edge corresponds to the addition of a new constraint which causes the circuit state to change. Each node is

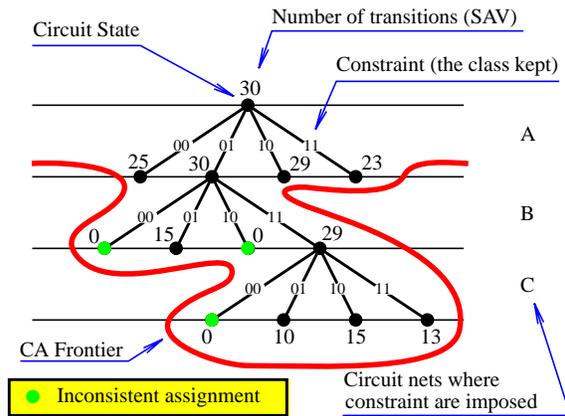


FIGURE 73: Case analysis decision tree

characterized by a *switching activity value* (SAV) which is an upper bound on the number of transitions as computed with the constraints leading to the node. The top node is the state before the case analysis begins. The process of extending the decision tree in one node by one branch is called a *decision*. Splitting the abstract waveform on a net into at most 4 waveform classes represents making up to four decisions. Making a decision implies recomputing the waveforms in the constraint system; therefore, we use the number of decisions as a measure of the amount of work spent by a processor on the case analysis. *Decision analysis n-levels deep* extends the decision tree from a node by up to $4 + 4^2 + \dots + 4^n = \frac{4^{n+1} - 4}{3}$ decisions. *Case analysis n-level-deep* is an n-level deep decision analysis on the root of the decision tree.

The overall upper bound on switching activity is the maximum over all the leaves of the decision tree. When waveforms on all circuit nodes are reduced to simple waveforms, the case corresponds to a simulation and the transition count is exact for those specific reduced waveforms.

3-2.5 Case analysis algorithms

Three forms of case analysis were developed. The simplest one [ZCSR95] is called the *stack-based case analysis (SCA)*, since it uses a stack to store paths in the decision tree for backtracking during the search.

The algorithm is a pure depth-first search through the decision tree. It is very efficient for small circuits, however, for large ones it rebuilds the decision tree too often. The problem can be overcome by a method inspired by the A^* search method from AI that combines depth-first and breadth-first search. It is referred to as the *frontier case analysis* or *PCA, path case analysis* (Figure 74). The method is also similar to the partial enumeration described in [KNYH93]. It main-

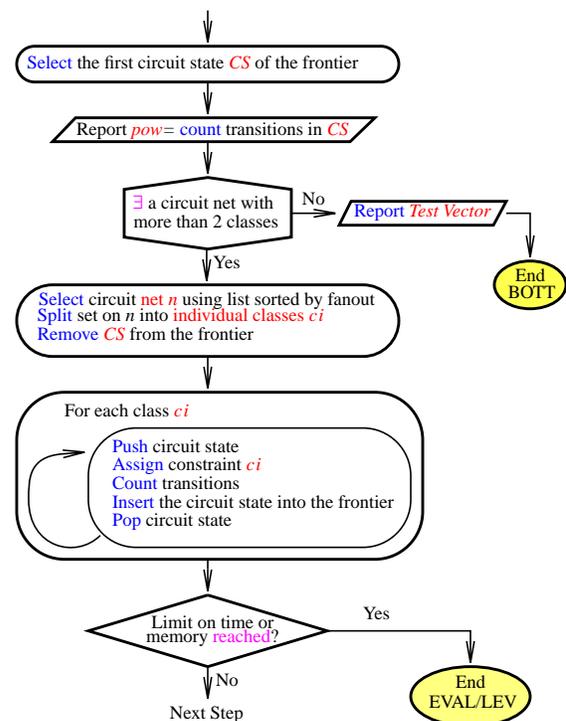


FIGURE 74: Step of the frontier-type case analysis

tains a frontier of the current leaf nodes of the decision tree, sorted by the number of transitions. Since the upper bound on the number of transitions is the maximum over all cases, the algorithm always extends the decision tree from the node with the highest number of transitions. The union of the constraints corresponding to the leaves completely covers the original abstract waveform on a circuit net which ensures the completeness of the case analysis. When the number of transitions is

the same for more than one leaf node, the deeper one with more constraints is kept at the head of the frontier list, giving preference to depth-first exploration that leads faster to the identification of a pair of input vectors that produces the highest switching activity.

The most advanced case analysis is *HPCA*, which stands for *heap path case analysis*. It is based on the same algorithm as the PCA, but it represents the decision tree in a heap-like structure. This is useful when operating on a large circuit and with a large number of decisions. The main features of the heap path case analysis (HPCA) are as follows:

- HPCA uses a heap rather than a list of paths; therefore, it has linear rather than exponential memory requirements in terms of the number of decisions,
- HPCA allows a variable number of decisions when extending a path in the decision tree,
- HPCA performs check-pointing, hence it can recover after a process/computer crash,
- HPCA has the capability to analyze one circuit on many machines in parallel (up to 50000; 90 were tested).

3-2.6 Net selection for case analysis

The order of nets in which the decisions are made is very important. It is similar to ordering variables in BDDs. The entire section Section 3-6 is devoted to heuristics for finding the most efficient net ordering. The simplest one is ordering by decreasing fanout. It is assumed that the

larger fanout the higher influence of the node's waveforms on the other nets in the circuit. As inverters and buffers introduce no pessimism to the waveform propagation, they are ignored when determining fanout as shown in Figure 75.

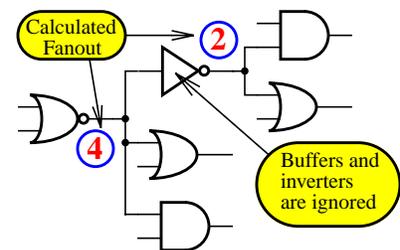


FIGURE 75: Counting fanout

3-2.7 Properties of the constraint system and of the case analysis

In this section we show why the constraint system actually computes an upper bound on the switching activity and that the case analysis tightens the upper bound and converges towards the identification of a test vector with the highest switching activity.

Conservatism (*upper bound*) of the constraint assignment operation is guaranteed by the structure of the constraint system. All gate operations are non-decreasing in the sense that they do not lose any transition. This is assured by the gate functions that generate a transition on the output if at least one combination of input transitions can cause it. Similarly for the partial inverse operations: only those transitions are removed that cannot produce a transition on the output when combined with any waveform on the other inputs.

Because all constraints are conjunctive (must hold at the same time), the value of each waveform variable is computed as an intersection of all the contributing projections and itself (Figure 64). This guarantees *monotonicity* (non-increasing) of the constraint system evaluation (which is done by computing the greatest fixpoint). Therefore, anytime a new constraint is added, the greatest fixpoint calculation converges.

Convergence of the case analysis is also easy to show. In each step, it chooses a circuit net with at least 2 waveform classes, splits them in the four individual classes and replaces the tree node by a subtree. No transitions are added, but only removed, therefore the constraint assignment is monotonically non-increasing. The proof that the case analysis *identifies one of the test vectors*, can be done using a counter-example: If it did not identify a test vector, there must be at least one circuit input with more than two classes. Such a net may be split further; therefore, the case analysis is not yet finished. Once each circuit input has only one class, either *00*, *11*, *01* or *10*, it corresponds to a simulation for 2 vectors. The class name identifies the pair of bits (the first one till time t and second after time t) of the vector on that input. The convergence is further accelerated by choosing the deci-

sion node deeper in the decision tree (i.e., with more nets already constrained to a single class) among those with the same SAV (switching activity value). Case analysis can also be performed on individual transitions, rather than classes. Any subset of simple waveforms which forms a *valid* (see Section 3-1) class can be used.

The case analysis can be performed in parallel on a number of processors, however, in this case we have to show that there exist *independent subspaces* in the whole search space. Such subspaces do exist and they are any two sub-trees of the decision tree which differ in the decisions on at least one net. The proof is as follows: Recall that decisions are made on waveform classes. Since waveform classes are distinguished by the initial and final values, no real waveform can be in 2 different classes. Since any actual behavior of the real circuit (or simulation for 1 pair of input vectors) assigns only one real waveform to each net, there is only one waveform class on each net in our model for such a case. The class on the net where we made the decisions then determines which one, and the only one, of the subspaces this simulation falls into.

3-3. Example

The case analysis is illustrated in Figure 76 on circuit *c17* from the ISCAS85 benchmarks circuits [BrgF85, Brya89]. The *c17* is small enough to allow a complete overview of the procedure.

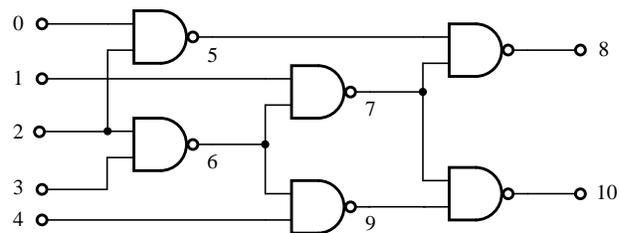


FIGURE 76: Circuit c17; all gates have unit delay.

The primary inputs are assigned the two-vector waveform sets and these are then propagated through the network. The resulting abstract waveforms on the nodes of the circuit are shown in Figure 77. The maximum number of transitions on each net as

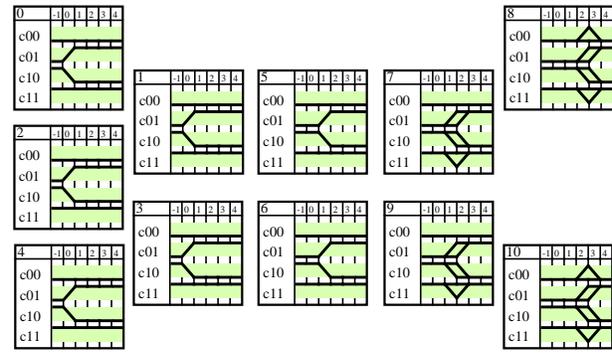


FIGURE 77: Waveforms of c17 after initial forward propagation

computed by our algorithm (Section 3-1.3.2) is: two transitions on nets 7, 8, 9 and 10 due to the potential static hazards in classes 00 and 11, and at most one transition on all other nets. The maximum number of transitions in the circuit (including the primary inputs) is thus 15.

Case analysis is now used to tighten this upper bound. From the state of the circuit after the initial propagation (Figure 77), the waveform classes on the net with the highest

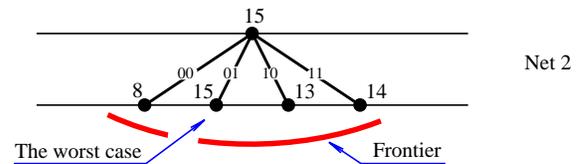


FIGURE 78: Decision tree for c17, analysis on net 2

fanout (e.g., net 2) are enumerated. We constrain the waveforms on net 2 to one class at a time, re-compute the abstract waveforms so that all constraints are satisfied, count the transitions, and return the circuit to the state as it was after the initial forward propagation. The number of transitions for those four cases is as follows, expressed as follows: $net:class \rightarrow transitions$: $2:c00 \rightarrow 8$, $2:c01 \rightarrow 15$, $2:c10 \rightarrow 13$, $2:c11 \rightarrow 14$. The four cases form the *frontier* of the case analysis. The cases cover all possible input patterns, thus giving an upper bound on the number of transitions. The upper bound is determined by the assignment resulting in the highest number of transitions ($c01$ with transitions). The situation is depicted by the decision tree in Figure 78.

The list of nets for the case analysis is calculated before the case analysis starts or dynamically on the fly using heuristics looking for the nets with the highest “influence” in the circuit, i.e., making a decision there elimi-

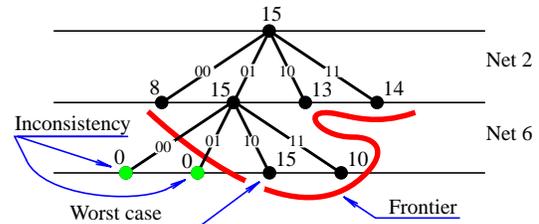


FIGURE 79: Decision tree for c17, analysis on nets 2 and 6

nates more transitions than on other nets. In this example we are using the heuristic *FANOUT* which orders nets statically according to decreasing fanout. In *c17*, the nets 2, 6 and 7 have fanout of two, and all others have fanout of one. The list of nets for case analysis can thus be 2, 6, 7, 0, 1, 3, 4.

Since the maximum transition count along the frontier is still equal to 15, we extend the worst case node by adding constraints on the next circuit net on the list, i.e., net 6. Again, we enumerate all classes on that net.

We obtain two cases, $6:c10 \rightarrow 15$ and $6:c11 \rightarrow 10$ (Figure 79). The new globally worst case becomes the simultaneous assignment of *c01* on 2 and *c10* on 6. Its waveforms are shown in Figure 80.

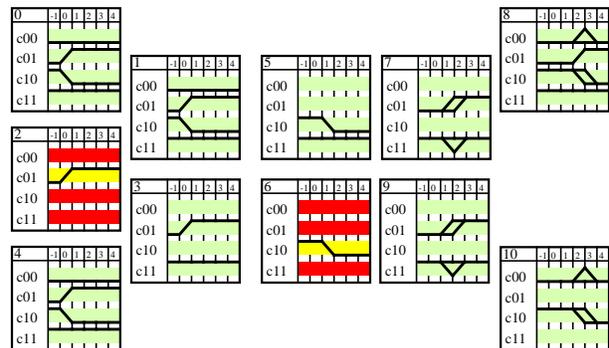


FIGURE 80: Waveforms of c17 for assignments 2:c01, 6:c10

The case analysis continues until either all circuit nets carry only a single waveform or the limit on CPU time or memory is reached. The first condition applies in this example. The entire verification is described by the decision tree shown in Figure 81. After constraining circuit net 4 to the

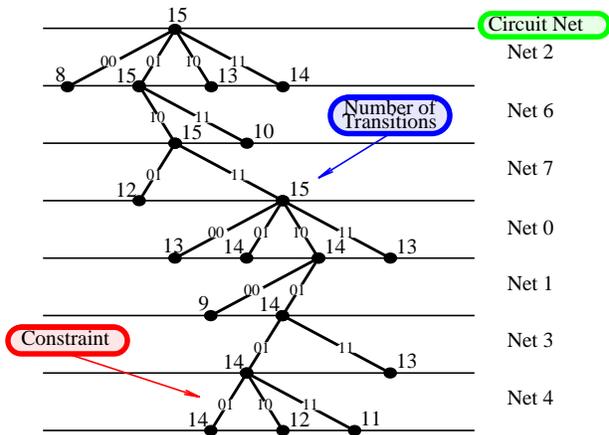


FIGURE 81: Decision tree for c17, complete analysis

individual classes, the case 4:c01→14 is one of the worst ones on the frontier. As all circuit nets now carry only simple waveforms, no further refinement is possible. Figure 82 shows the resulting worst-case waveforms.

A lower bound on the number of transitions can be obtained by simulation. If the number of primary inputs is less than about 10, then exhaustive simulation can be used. In the case of c17 we of course obtained 14 transitions. There are two pairs of input vectors that yield 14 transitions in the circuit, one of them was identified by the case analysis.

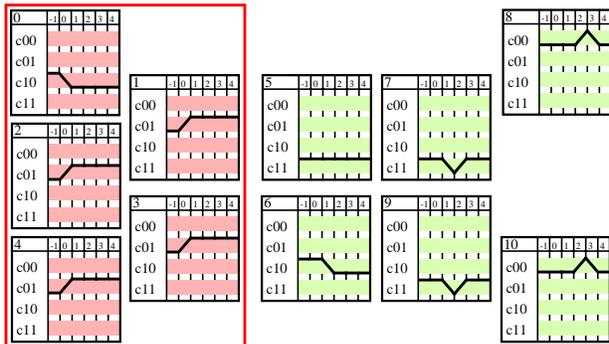


FIGURE 82: Simple waveforms with transition count of 14

In the previous two sections we showed the principle of the case analysis procedure and demonstrated it on a small example. In the following sections we analyze some circuit properties which can be used for guiding the case analysis procedure, starting with reconvergence analysis in the next section.

3-4. Reconvergence analysis

In this section we introduce a formal framework for the analysis of reconvergent regions based on work of F. Maamari [MaaR90]. We then we present a modified algorithm for identifying reconvergent regions and outline how knowing these regions may help in the case analysis.

3-4.1 Introduction

Reconvergence in combinational circuits is a form of internal spatial correlation. Reference [MaaR87] is the best introduction into the analysis of reconvergent regions. The authors define the basic terminology (with some terms redefined in more recent papers) as follows:

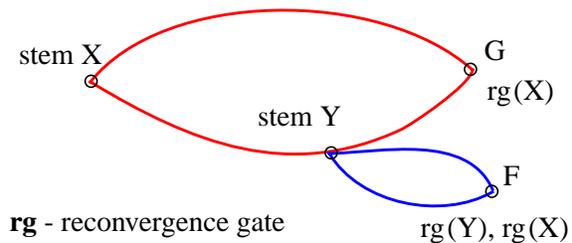


FIGURE 83: Reconvergent stem and reconvergence gate

- X is a *reconvergent stem* and G is its *reconvergence gate* if there are two or more interconnection-disjoint paths between X and G (Figure 83). Note that the gate and its output are not distinguished in the paper, i.e., the definition does not account for multi-output components.
- If Y is a stem on a path between X and G (Figure 83), then all reconvergence gates of Y are also reconvergence gates of stem X .
- If reconvergence gate G of stem X does not drive another reconvergence gate of X then it is a *closing reconvergence gate* of X ($cg(X)$ in Figure 84).

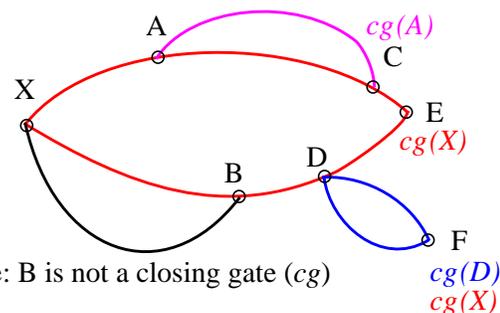


FIGURE 84: Closing gate

- A *reconvergent stem region*

(or reconvergent region) is composed of all interconnections (called also lines) that are on the path from the stem to a reconvergence gate (Figure 85). Note that a net = several lines, i.e., branches.

- *Exit lines* are lines through which the signal can exit the region. They are of two types: 1) branches of stems belonging to the region, 2) outputs of closing reconvergence gates. Note that no two exit lines of one reconvergent region fan out to the same primary output.

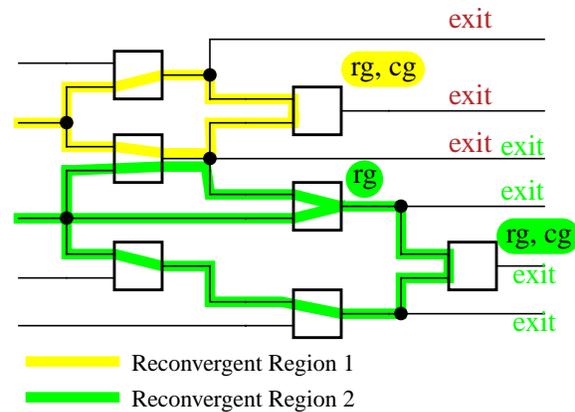


FIGURE 85: Reconvergent stem region

3-4.1.1 Use of reconvergent regions in fault analysis

The property the authors use in fault analysis is that if a stem fault is detectable on the exit line, and if the exit line is critical (has a sensitizable path to a PO, primary output), then a fault on the stem is detectable on the PO if it is detectable on the exit line. The consequences: stem faults need not to be simulated beyond their reconvergent stem regions.

3-4.1.2 Complexity of fault analysis and region overlap

Region overlap of a net is defined as the number of regions the net is a member of. For ISCAS85 circuits the highest overlap ranges from 34 in c880 to 845 in c6288. Average (per net) overlap ranges from 6.8 in c5315 to 184.2 in c6288. Since - contrary to our approach - they have to simulate (up to) every stem fault with respect to its stem region, the overlap determines in how many simulations the net will be involved, therefore the CPU time of the fault analysis. The maximum time to fault-

simulate all reconvergent stems is $T = \sum_{i=1}^N tR_i$, where N is the number of lines, t

is the time required to process a fault on a line, R_i is the overlap on line i . Using the average overlap \bar{R} , $T = Nt\bar{R}$.

Notes about circuits: c499 and c1355 are functionally equivalent, but XOR gates are expanded into their AND-OR structural equivalents in c1355.

3-4.1.3 Secondary reconvergence

In [MaaR88a], primary and secondary reconvergences are distinguished.

- G is a *primary reconvergence gate* of stem X if there are at least two disjoint paths from X to G .
- F is a *secondary reconvergence gate* of X if it is a primary or secondary reconvergence gate of some

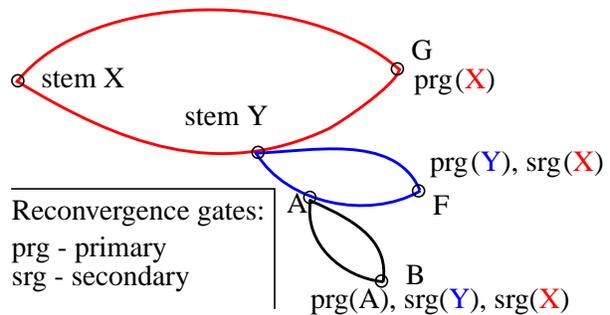


FIGURE 86: Primary (prg) and secondary (srg) reconvergence gates

Y , it is not a primary reconvergence of X , and Y is located on a path from X to any of its primary reconvergence gates. The authors of [MaaR88a] also define narrow and wide reconvergent stem regions. A narrow reconvergent region does not have any reconvergent stems on the path from the stem to a primary reconvergent gate (like the region with stem A in Figure 86). All reconvergent regions which are not narrow are wide.

- A *reconvergent stem region* is redefined in [MaaR88a] to include also all fanout branches of the stem and all branches of the output of the closing gate (Figure 87).

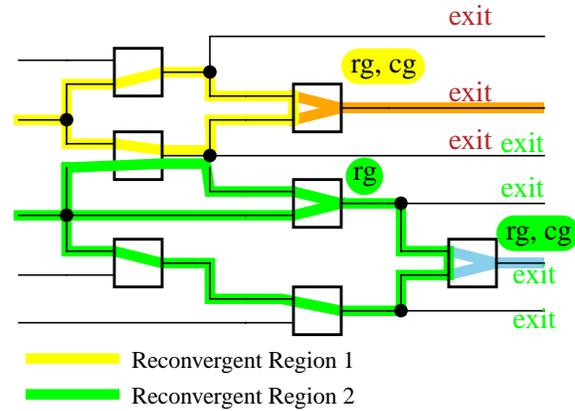


FIGURE 87: Enlarged reconvergence regions

- The definition of an *exit line* is refined, but it keeps its original meaning: a line is an exit line of a stem region if both: 1) it is an output line of a net in the region and 2) it is an input line to a node outside the region.

In the next section we explain a fault analysis algorithm based on reconvergent regions. Note that the fault analysis is not directly relevant to power verification. It is presented here only to provide an example of the use of reconvergent regions to understand their properties in depth. The reader not interested in fault analysis can proceed directly to Section 3-4.3 on page 110.

3-4.2 Fault analysis based on reconvergent regions

From the definition of reconvergent regions it follows from [MaaR88a] that the subnetworks driven by exit lines of the same reconvergent stem are disjoint (Figure 88).

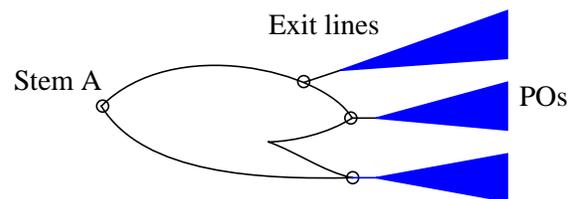


FIGURE 88: Disjoint subnetworks driven by exit lines

A direct application of stem regions and the properties of exit lines to fault simulation is shown in Theorem 1 in [MaaR88a]: For a given test vector on PIs, a stem fault is

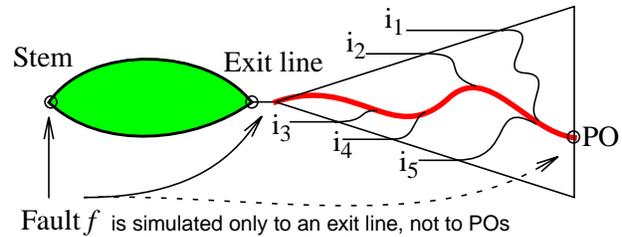


FIGURE 89: Consequence of exit line properties

detected on a primary output if and only if the fault is detected on an exit line and that exit line is critical to the output (there is a sensitized path). Note that this is possible, since other lines (i_1 to i_5 in Figure 89) driving the cone of the exit line are topologically independent of the stem, i.e., they are not influenced by the fault.

The fault analysis algorithm operates in two steps. First, all the stem regions are simulated to determine whether a stem fault is detectable on exit lines and then the criticality of every exit line is verified to determine whether it can propagate a fault value to POs. The algorithm was modified for better performance as reported in more recent papers.

3-4.2.1 The fault model and dynamic reductions

In [MaaR88b], the authors define the fault model for the first time. For a certain input vector V , each line l has a stable (fault-free) value L . Detectability of stuck-at- L fault on line l is denoted $d(l_{\bar{L}})$. If $d(l_{\bar{L}}) = 0$ then the fault cannot be detected.

To check whether the opposite fault \bar{L} is detectable one must compute $d(l_{\bar{L}})$, further denoted as $d(l_*)$, which is done as follows:

1. If l is a PI, then $d(l_*)=1$.
2. For a gate, it is calculated from the detectability of the opposite of the fault-free value on the output $d(o_*)$ and the sensitizing condition for line l , i.e., the detectability $d^o(l_*)$ of fault l_* on l observed on o (critical path tracing) is $d(l_*) = d^o(l_*) \cdot d(o_*)$, which can be easily computed in one backward pass starting from the POs. Note: all detectabilities are easy to compute because the authors

calculate them for each vector, thus the detectabilities are Boolean values, not Boolean functions of PIs.

3. If l is the stem of a reconvergent stem region with exit lines y_i , then the fault on l is simulated to the exit lines, resulting in detectabilities $d(y_i)$, one Boolean value for each exit line. Then, $d(l_*) = \sum_{\text{all } i} d(y_{i*}) \cdot d^{y_i}(x_*)$, i.e., the stem fault is detected if it is detected on at least one exit line ($d^{y_i}(x_*)$) which has a sensitizable path to a PO ($d(y_{i*})$). This is achieved in a number of steps equal to the number of lines in the region.

The fault simulation algorithm proceeds as follows. In the initial phase, all the reconvergent regions are extracted. This is followed by two passes for each input vector. In the forward pass, fault free values are computed and stem faults are propagated to the exit lines of reconvergent regions. In the backward pass (from POs towards PIs) detectabilities of individual lines are computed based on their successors or exit line detectabilities in the cones of stems. Since all the operations are Boolean, the implementation uses word (32-bit) operations to process 32 input vectors at the same time.

The paper also proposes how to dynamically reduce processing steps necessary for fault simulation. The following three methods are proposed:

- If a fault on a stem has been detected as well as all faults in the subnetwork driving it, then neither the stem region is simulated in the forward pass, nor a path from the POs to the exit lines is traced in the backward pass.
- A fault on stem x is also dropped if all stem faults of the regions with exit line x have been detected. The faults in the subnetwork driving x , excluding subnetworks driving stems in the subnetwork, are dropped too (Figure 90).

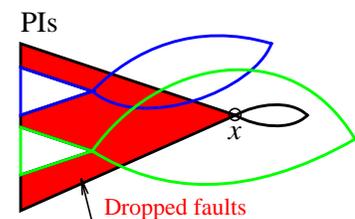


FIGURE 90: Dropping faults, method "2"

- When all faults in the subnetwork driving POs reached from a line x have been detected, x is dropped from the simulation in the forward pass.

3-4.2.2 Results

Reference [MaaR90a] is a very comprehensive work which summarizes the previous publications in this area by the authors and newly introduces the following:

- The fault dropping is better formalized using (a topological) dependency graph.
- They describe some factors the fault dropping efficiency depends on - the size and the topology of the circuit, the presence of random-pattern-resistant faults, the presence of redundant faults.
- The experimental results include c6288, however the CPU time is over 1 order of magnitude higher than for the slightly larger c7552. The authors ascribe it to the much larger average region overlap in c6288.

3-4.3 Reconvergence in timing and power verification

The reconvergent regions as a way of identifying internal spatial correlation within a circuit have the following potential for our case analysis:

- The stems of “appropriate” regions and region sets can be used as points where the case analysis is performed.
- To help identify subregions of the circuit that are suitable for exhaustive analysis.

As for the first problem, the ideal place for case analysis (with respect to our representation of signals by abstract waveforms) are those nodes that influence as large a portion of the circuit as possible independently of other nets. In practice, it means that the case analysis can be performed on the stem of a large reconvergent region with no entry nets (nets which are in the reconvergent region but are driven by a gate outside of the reconvergent stem region). Since this situation is very rare in practice, a set of overlapping reconvergent stem regions with the least number of entry nets can be used as shown in Figure 91.

Ideal region with no entry nets



Set of regions with minimum of entry nets

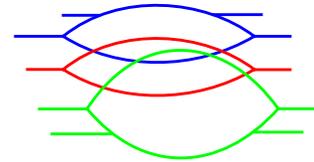


FIGURE 91: Case analysis on overlapping reconvergent stem regions

However, at the present time it is not clear how to find such sets of regions. There is an obvious algorithm of exponential complexity based on measuring the overlap and the number of entry nets to the set of regions for each pair, triplet, quadruplet, etc., of regions. Such an algorithm is not realistic due to its high complexity. Therefore, we tested a simplified version for one region, i.e., the case analysis is performed on nets which are stems of reconvergent regions with the highest ratio

$\frac{ng}{ne}$ of the number ng of gates (size) to the number ne of entry nets. Another possi-

bility is to include also the region overlap on entry nets, for example $\frac{ng \sum_{i=0}^{ne} R_i}{ne}$, where R_i is the region overlap on entry net i . Such a criterion gives preference to regions with more gates, fewer entry nets, and with entry nets that belong to a higher number of reconvergent stem regions.

As for the second problem, the exhaustive enumeration of waveforms for power analysis, we need as large a (not necessarily purely reconvergent) region as possi-

ble, but with a limited number of entry nets to the regions. Overlap is not desired, hence we are looking for a partitioning of the circuit into such regions. Again, the ideal regions would be reconvergent regions or their sets with the minimal number of entry nets such as the regions in Figure 91. Probably a good starting point is to take regions with the smallest number of entry nets and if needed then cut some to fit within the limit on the number of entry nets.

3-4.4 Algorithm for reconvergence analysis

The algorithm is based on the work of Maamari [MaaR90a]. However, we generalized the algorithm for multi-output gates and simplified it. For every fanout stem it performs the following steps:

- The cone of nodes reachable from the stem is tagged by TAG0. TAG0 is an integer greater than zero, and it is the same for all branches of one net. The stem is assigned TAG0=1. The output of the gate reached from a branch with a TAG0 is assigned TAG0 equal to the lowest yet unused value. This identifies the transitive fanout “cone” of the stem (Figure 92).

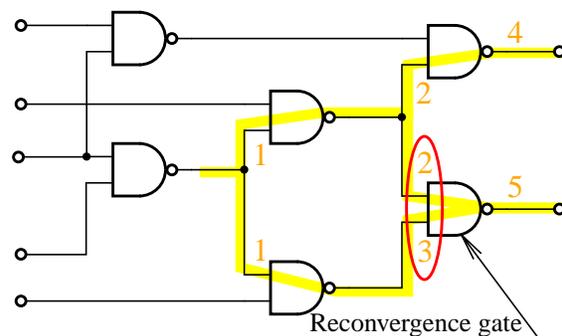


FIGURE 92: Finding reconvergence gates - TAG0

- Reconvergence gates are identified by checking inputs of the reached ($TAG0 \neq 0$) gates. A gate is a reconvergence gate, if at least two of its inputs have different values of TAG0, as shown in Figure 92. The

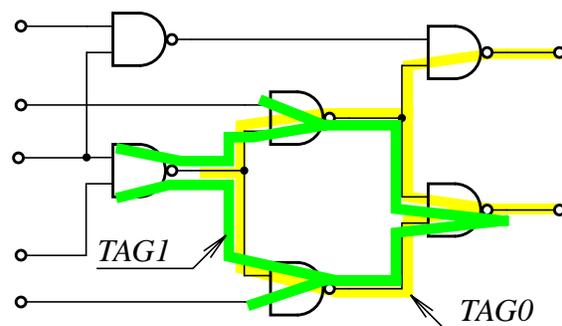


FIGURE 93: Finding reconvergence gates - TAG1

list of the so identified reconvergence gates is stored.

- Then a second tag (TAG1) is introduced. TAG1 is a Boolean that is 1 if the net is in the fanin cone of a reconvergence gate. The tagging by TAG1=1 stops on nets with no TAG0 to reduce the complexity of the tagging procedure (Figure 93).
- A reconvergent region (its nets, exit nets, and gates) is identified by tracing the stem's fanout cone in the forward direction. A net is in the reconvergent stem region, when it has at least one fanout branch with both

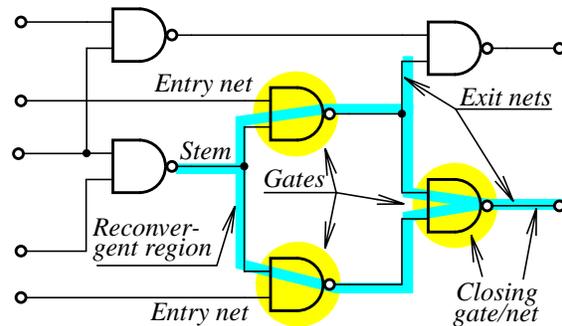


FIGURE 94: Identifying reconvergent regions

one fanout branch with both tags TAG0, TAG1 greater than zero. A gate is in the region when its output is in the region or at least two of its inputs are in the region. An *exit net* is a net of the region which has at least one branch going outside the region. Each net in the region is looked up in the list of existing reconvergent regions. If it is a stem of an already identified region, this secondary reconvergent region is merged with the region where the net was found. Finally, the *entry nets* are identified. An entry net is a net in the region which is either a PI or it is driven by a gate outside the region (Figure 94).

- Since some circuits may have a very large number of regions, we added a limit on the number of gates in the region to reduce the search. The regions larger than the limit are ignored. A secondary reconvergent region of such a large region is stored only if this secondary region is smaller than the limit.
- A *forest of the hierarchy of regions* is created. A region *R1* is a subregion of *R2* if the stem of *R1* is a net of *R2* (secondary reconvergence).

An example of region dependencies is shown in Figure 95. Each node is a reconvergent region, the size of the bullet corresponds to the number of gates.

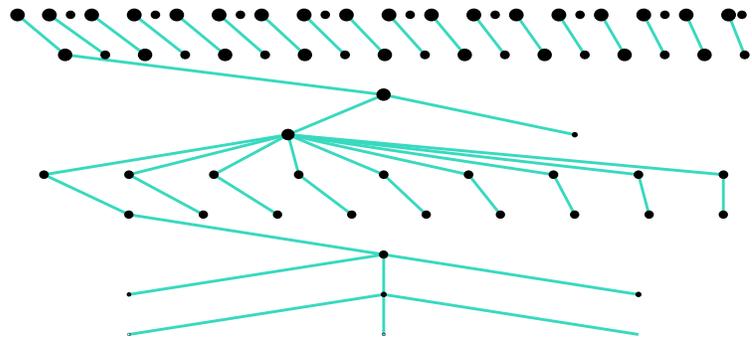


FIGURE 95: Forest of reconvergent stem regions in c432

An edge represents the relation “subregion of”.

In the previous section we explained reconvergent region analysis. It is one of many ways for supporting the choice of nets for the case analysis. In the next section we explain another method: global implications on Boolean values obtained by learning.

3-5. Learning

Learning of logic implications on internal signals of a circuit has been widely used to speed up ATPG [KuPr93]. Consider the simple circuit in Figure 96.

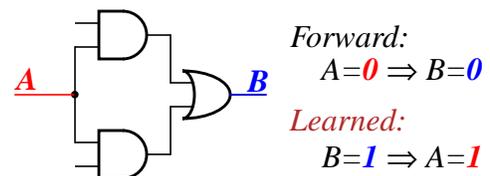


FIGURE 96: Implication and learning

When input A is assigned the value of 0

then B will necessarily be 0 because both the inputs of the OR gate carry value 0 regardless the value on the side inputs. Recall that $X \Rightarrow Y$, where \Rightarrow represents logical implication represents logical implication, is equivalent to $\bar{Y} \Rightarrow \bar{X}$ for any two Boolean values X and Y . Therefore, if $A=0 \Rightarrow B=0$ then $\overline{B=0} \Rightarrow \overline{A=0}$ which is $B=1 \Rightarrow A=1$.

Such implications can relate values on nets anywhere in the circuit, therefore, the implications are called *global implications*. Global implications are discovered by simulation of the circuit with all but one of the circuit nets set to the unknown value. In our example in Figure 96, all nets are set to X , then A is set to 0 and the values are simulated on the circuit. The net B as well as the inputs to the OR gate change from X to a single Boolean value, i.e., we learned an implication of $A=0$. Note that 3 implications have been learned, but only one (from A to B) is shown in the figure for illustration. Then, all nets are set to X again and A is assigned 1. In this case the simulation does not yield any value other than X ; therefore, no implication of $A=1$ has been learned.

Global implications are a way of expressing spatial correlation in the circuit directly by Boolean values. They help to speed up backtracking of values in special cases such as ATPG as it is briefly discussed next in Section 3-5.1. The use of global implications in our verification method is explained in Section 3-5.2.

3-5.1 Global implications in ATPG

The issue is how to generate a test vector for a fault. To detect a fault stuck at $f \in \{0, 1\}$ on net N denoted N_f , the net N must be driven to the value of \bar{f} (*controllability*). At the same time, the faulty value f on N must be observed on a primary output (*observability*). To find an input test vector making the fault detectable, one has to propagate or justify Boolean values in the circuit. This is usually done by making local value assignments to nets and backtracking if a conflict is detected. The knowledge of global implications can substantially speed up this process [KuPr93].

3-5.2 Use of global implications in the analysis of switching activity

Let us analyze again the circuit from Figure 96, and assume that it is a part of a larger circuit. Suppose that in the case of power analysis we obtain the abstract waveforms as shown in Figure 97.

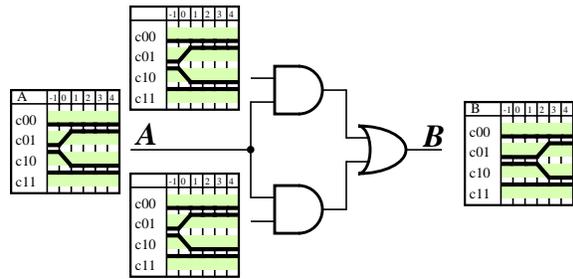


FIGURE 97: Learning - abstract waveforms

Furthermore, let us assume that during case analysis the constraint $B=c11$ is assigned. The partial inverse functions cannot eliminate waveform classes $c00$, $c01$, and $c10$ on either inputs, because any of those combined with $c11$ on the other input can produce $c11$ on the output, as shown in Figure 98.

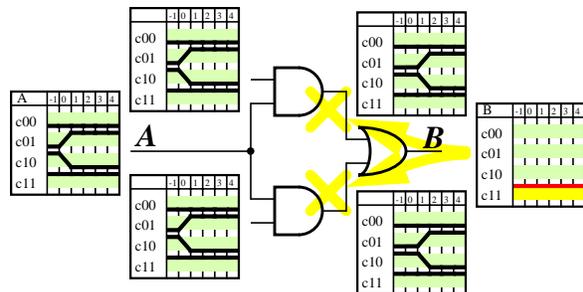


FIGURE 98: Learning - constraint $B=c11$

However, if we consider the derived implication $B=1 \Rightarrow A=1$ then all the classes that have their initial or final value 0 will be eliminated on A, as shown in Figure 99.

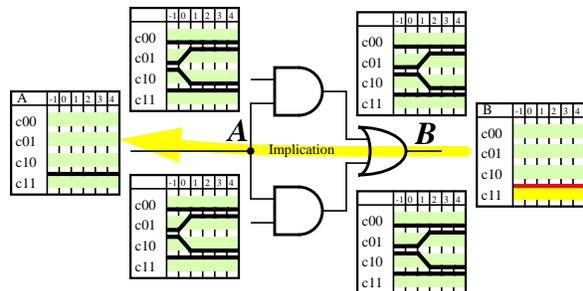


FIGURE 99: Learning - effect of the learned implication

The resulting value does not propagate further, because the side inputs to the AND gates contain each all four classes. If, however, one of them contained only classes $c10$ and $c11$, then the resulting reduc-

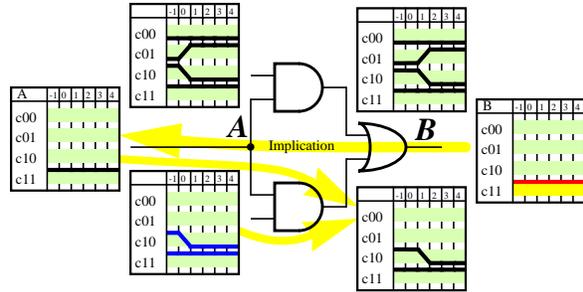


FIGURE 100: Learning - propagation due to a reduced side input

tion on A to $c11$ can propagate further, as shown in Figure 100. This may result in a tighter bound on the transition count because of the eliminated classes on A and all the other nets where the reduced waveform set propagates to.

Global learning can be performed on any circuit net and all the learned implications used. The number of implications can be enormous. However, many implications can be derived one from the other or the implications are by controlling values only and these are already obtained by the partial inverse functions of the gates. Therefore, we perform learning only on the stems of reconvergent regions. This guarantees that the number of learned implications is relatively small, and that we include only

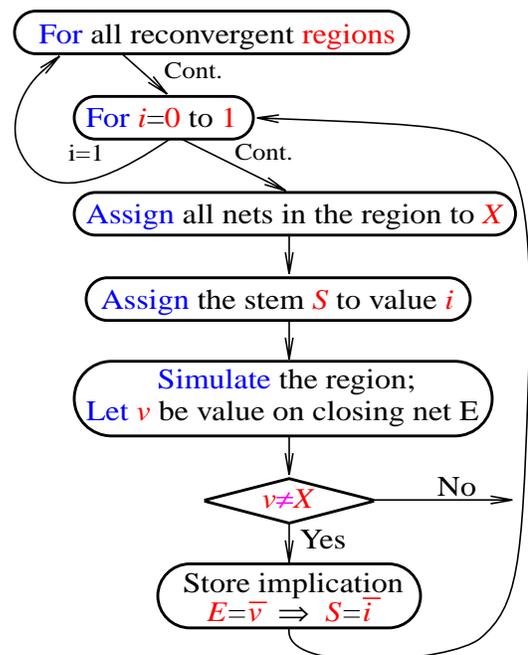


FIGURE 101: Learning along reconvergent regions

implications that the partial inverse function are not able to derive. The algorithm proceeds as shown in Figure 101.

In the previous sections we described reconvergence analysis and global learning. These serve as the basis for two of our heuristics for guiding the case analysis. The heuristics are described in the next section.

3-6. Heuristics for net selection in case analysis

The case analysis as presented in Section 3-2 makes decisions on statically selected circuit nets. The selection criterion is fanout. The nets with larger fanout are used earlier in the case analysis because they are connected to more gates and thus (hopefully) the assigned constraints may propagate further, eliminating more transitions. The fanout is not the only possible criterion for net selection, nor must the choice be statically predetermined. The problem of net selection for case analysis is very similar to the problem of variable ordering in BDDs: with a good net selection the case analysis tightens the lower bound faster because an earlier “good” decision prunes the entire subtree of otherwise needed decisions. Similarly, the size of the BDD tree representing a Boolean function can be minimized by ordering the variables. In this section we analyze several possible selection criteria.

3-6.1 Fanout

Net selection based on the largest fanout is a natural choice. The higher the fanout, the larger portion of the circuit that is influenced by the decisions made on the net. Consider the portion of a circuit shown in Figure 102.

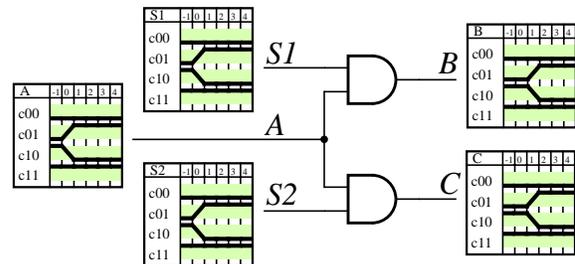


FIGURE 102: Fanout influence

Making a decision on net A is equivalent to making two decisions, on nets B and C, when applying constraint $A=c00$. This is because 0 is the controlling value for the AND gates, thus 0

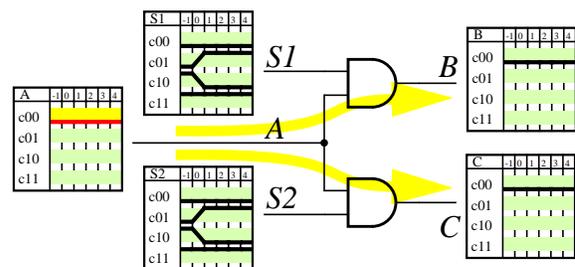


FIGURE 103: Fanout influence - constraint $A=c00$

on A causes both B and C to be 0 too (Figure 103).

Since our gate models include full projections from any terminal to any other, including outputs to inputs, why would not $c00$ assigned to B propagate backward? The reason lies in the

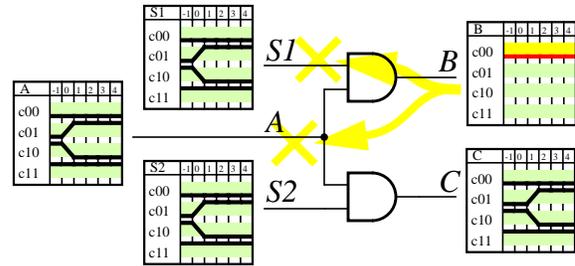


FIGURE 104: Fanout influence - constraint $B=c00$

four classes on the side input $S1$. When B is 0, we cannot decide whether it is caused by $S1$ or by A ; therefore, partial inverse functions of the gate cannot remove any classes from $S1$ and A (Figure 104). This means that to achieve the same reduction of waveforms on B and C , the assignment $A=c00$ is equivalent to two simultaneous assignments $B=c00$ and $C=c00$.

The list of nets ordered by decreasing fanout is the default net selection in our case analysis. However, fanout is not the only criterion which determines the strength of the influence of a net on the other nets in

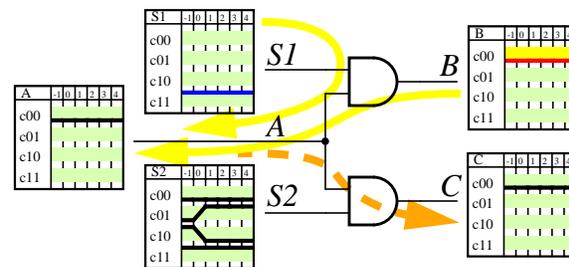


FIGURE 105: Fanout influence - constraint $B=c00$ when $S1=c11$

the circuit. This concept of “influence” was explained in Section 3-5 in terms of learning and necessary implications. The influence is determined by the topology, function of the circuit and the already assigned constraints. For example, the assignment $S1=c11$ in Figure 105 allows the assignment $B=c00$ to propagate backward to A and further from A to C .

3-6.2 Fanout on primary inputs

Since all signal values are derived from the primary inputs it is worth investigating how good it is to choose only the primary inputs (sorted by largest fanout) for the case analysis. For a circuit with a small number of primary inputs it is always better to perform the case analysis on them. Even the order is not important because in the worst case all combinations of the individual classes will be tried. However,

when there are too many primary inputs the assigned constraints cannot propagate too far because of side inputs because we cannot enumerate all possible assignments on the primary inputs. In some circuits there exist primary inputs that propagate assignments deeply into the circuit regardless of the values on the other inputs. However, in general, there are more internal (fanout) nodes which have higher influence on other nets than the primary inputs. This has been confirmed by our experiments on the ISCAS'85 circuits, as will be shown in Section 4-6.

3-6.3 Dynamically created list of nets

The best order of applying decisions can be chosen by an exhaustive search, i.e., by trying all possible orders and running the case analysis until a test vector is obtained. Unfortunately, the number of possible orders and the execu-

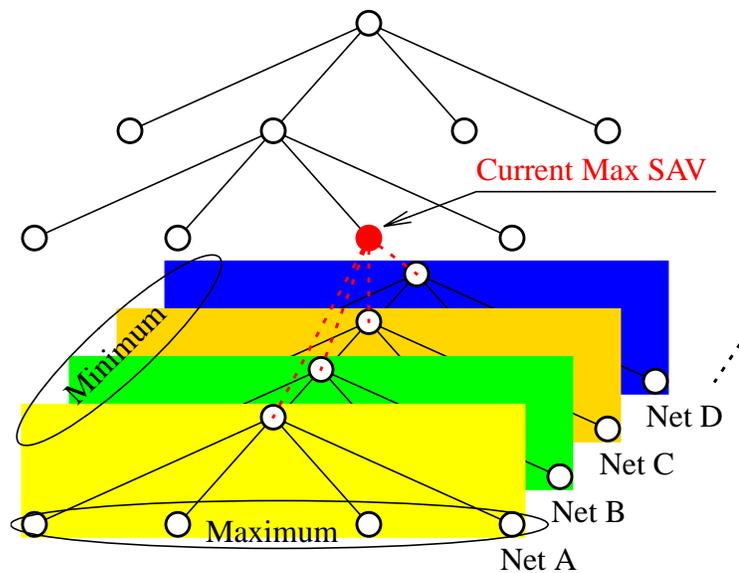


FIGURE 106: Dynamically created list of nets

tion of the case analysis until the end are both exponential in the number of nets. A much simpler version is to choose the nets on-the-fly during the case analysis. The algorithm proceeds as follows (Figure 106): For the node with the current maximum SAV make (up to four) decisions on classes on each net one at a time, and then choose the net with the lowest maximum over all the decisions (up to four; only decisions that result in consistent constraint system). In this way the locally best net is chosen. This does not guarantee making the overall best choice of circuit nets, because applying two constraints individually, one at a time, may never yield any improvement, but the two together may be the best overall choice. Hence, choosing the locally best net does not guarantee that the case analysis

would converge rapidly toward a test vector. Furthermore, the dynamic net selection is very costly. Assume a circuit with N nets. To make a 1-level decision in the analysis, one has to try N 1-level decision analyses. In the next level, it is $4(N-1)$ to make 4 decision analyses, and so on, which is far too costly.

3-6.4 Reconvergent regions

Reconvergent regions are not only used to determine the place for global learning, but also they can be used to identify the order of nets for case analysis. Stems of reconvergent regions have a fanout of at least 2. If a single-class constraint is assigned to each of the entry nets of the reconvergent regions, then spatial correlation is enforced, resulting in a single class on each net of the region (like in a simulation). Furthermore, if global implications are added, the decisions may be done on the closing nets rather than on the stems. In Section 4-5, we shall present experimental results for the following heuristic procedure (called RCVFAN):

- Form a list of all closing nets of all reconvergent regions and all primary inputs,
- Sort the list in decreasing order of fanout.

3-6.5 Other heuristics for net selection in case analysis

The heuristics mentioned in the previous sections are only a small example of the possible approaches to net selection in the case analysis. We also tested selection based on the number of transitions on each net, time of the earliest and the latest transitions on each net, and the size of the reconvergent regions. None of these heuristics produced better results than any of the heuristics described in the previous section; therefore, they are not included in the experimental results.

In Section 3-6 we presented several heuristics for net selection for the case analysis. As will be shown in Section 4-6 none of them is the best one for all circuits but some of them provide better results on more circuits than others. Before presenting the experimental results in Chapter 4, we briefly describe in the next section yet

another way to tighten the upper bound. We then outline the implementation of our switching activity estimation tool in Section 3-8.

3-7. Use of timing constraints

By knowing the exact circuit delay (or an upper bound that is smaller than the topological delay), one can eliminate transitions in the abstract waveforms that occur after that delay, because they cannot happen in reality. The timing information is generally available, because it is essential for correct circuit function and/or desired performance¹. We can thus add the timing information as an additional constraint to our switching activity method. The best choice would be to obtain the exact delay on each net, i.e., time intervals where transitions cannot occur anymore. Then we can add constraints which eliminate all rising and falling transitions from those time intervals. Note that such constraints are no longer class-based, hence a separate proof that adding such constraints preserves validity would be needed.

3-8. Implementation

The switching activity verification method was implemented in C++. The use of an object oriented approach allowed us to write a constraint resolution system which can be shared by our switching activity verification tool and by a timing verification tool. The input is a Verilog netlist; commands are passed as command line arguments, and the output is both textual and graphical (FrameMaker format), giving information about the switching activity in the circuit. For more details about the implementation and the use of our tool see Appendix A.

1. That includes timing information on internal nodes, because a static timing tool needs it to compute timing information on the connected nodes. Though in most case the tool would not export such information to the user and thus a special API is necessary, similarly as in e.g., analysis of gate delay slowdown/speedup due to crosstalk.

CHAPTER 4

Experimental Results

In this section, we present the results of our experiments on the ISCAS'85 benchmark circuits [BrgF85]. We compute an upper bound on the peak switching activity. Two upper bounds are obtained: the first one after the initial fixpoint calculation and the second one after the case analysis. The upper bounds are compared to the lower bounds obtained by random simulation in Verilog XL. The true switching activity lies somewhere between the lower and the upper bounds. In all experiments the gates were assigned a fixed delay of 1 unit.

We first give in Section 4-1 the lower bounds on switching activity as obtained by simulation. The initial upper bounds obtained by constraint resolution appear in Section 4-2, and then the results obtained using the individual heuristics are shown in Sections 4-3 to 4-5. We then compare the results in Section 4-6. The effectiveness of the two main case analysis implementations, PCA and HPCA, are compared in Section 4-7. The benefit of using timing constraints is evaluated in Section 4-9. Finally, the performance of our method is compared with other methods in Section 4-10.

Table II describes the topological properties of the benchmarks: the numbers of primary inputs (*PI*), primary outputs (*PO*), gates, circuit nets, and the size of the largest fanout through buffers and inverters.

Name	PI	PO	Gates	Circuit nets	Fanout
c432	36	7	160	196	9
c880	60	26	383	443	8
c499	41	32	202	243	12
c1355	41	32	546	587	12
c1908	33	25	880	913	25
c2670	233	140	1193	1502	28
c3540	50	22	1669	1719	22
c5315	178	123	2307	2485	31
c6288	32	32	2416	2448	16
c7552	207	108	3512	3720	72

TABLE II: ISCAS'85 circuits [BrgF85]

4-1. Lower bound simulation

A random simulation was performed to obtain lower bounds on the maximal switching activity. It was carried out using a Verilog simulator from Cadence, Inc., over a period of 12 months on 4 machines, running 7 hours a day. Primary inputs of all 11 cir-

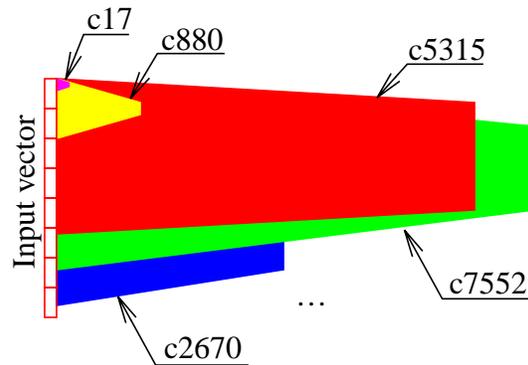


FIGURE 107: Simulation of ISCAS'85 circuits

cuits were connected in parallel to simulate them at once using only one Verilog license on each machine (Figure 107). The input vector was always composed of 8 consecutive 32-bit random numbers to create a 256-bit input vector. The simulation on each machine was assigned a separate seed - the initial random number. It proceeded as follows: At time 0 the primary inputs were assigned the first vector. Once all the internal nets stabilized the second vector was applied. After the longest topological delay of the circuits, the calculated switching activity for each circuit was compared with the current maximum for the circuit and the higher of the two was stored. Then, a new input vector was generated and assigned to the primary inputs, which started the next simulation cycle. The switching activity was calculated for each circuit as a sum of the switching activities on all nets. Each net

was assigned a transition counter that was reset to zero before a new input vector was assigned to the primary inputs and incremented each time a transition occurred on the net.

The simulation results are summarized in Table III. The table also compares the results for a relatively small number (150 thousands) and a large number (13 million) of test-vector pairs.

An example of how the lower bound progresses with the number of input vectors is shown in Figure 108 for c432 and in the subsequent figures for several other circuits.

An initial seed of 23 and a simulation for 13

million vectors were used. When fewer vectors are shown, it means that the lower bound did not change after the last vector shown. Since the lower bounds on circuit delay were obtained as a side product of the simulation, they are shown as well.

Circuit	Switching activity (SA) by simulation [transitions]		Ratio $SA_{13M}/SA_{0.15M}$ [%]
	$SA_{0.15M}$ ($150 \cdot 10^3$ vectors)	SA_{13M} ($13.08 \cdot 10^6$ vectors)	
c432	378	412	109
c499	253	273	108
c880	724	940	130
c1355	1049	1286	123
c1908	2087	2242	107
c2670	2558	2558	100
c3540	4258	4532	106
c5315	5697	5725	101
c6288	54812	59349	108
c7552	8612	8612	100

TABLE III: Lower bounds

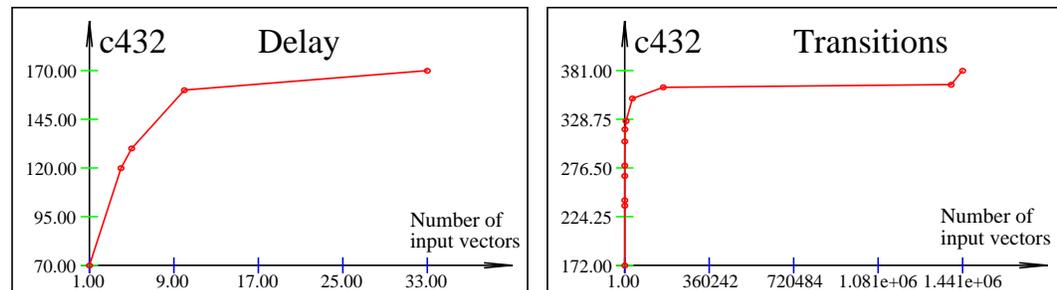


FIGURE 108: Lower bounds on delay and switching activity in c432

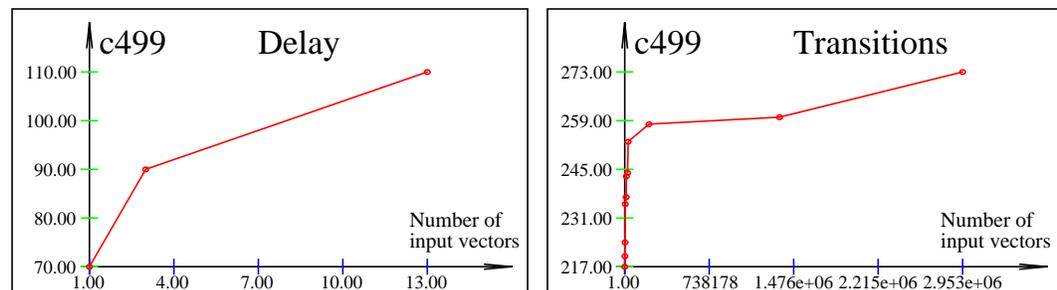


FIGURE 109: Lower bounds on delay and switching activity in c499

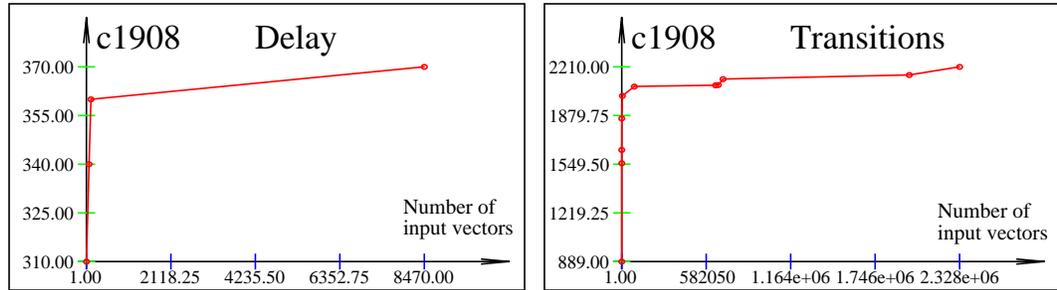


FIGURE 110: Lower bounds on delay and switching activity in c1908

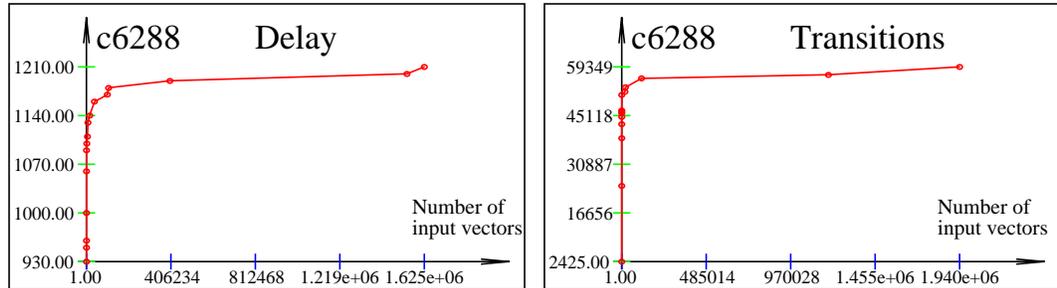


FIGURE 111: Lower bounds on delay and switching activity in c6288

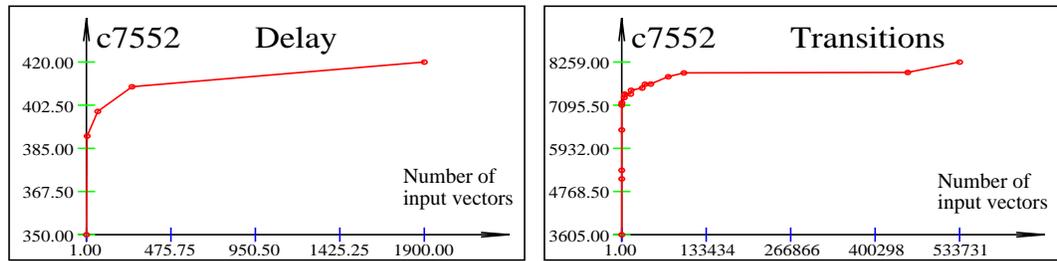


FIGURE 112: Lower bounds on delay and switching activity in c7552

4-2. The initial upper bound on switching activity

Once the circuit netlist is parsed and the constraint system constructed, the primary inputs are assigned 2-vector stimuli and the greatest fixpoint of the constraint system is calculated. The switching activity obtained for this state of the constraint system is the initial upper bound. It is calculated rapidly but is rather conservative.

Circuit	Initial Upper bound	CPU sec	Memory MB
c17	15	0.03	0.1
c432	903	0.53	1.1
c499	785	0.52	1.5
c880	2316	1.38	2.6
c1355	4985	1.98	3.8
c1908	8753	2.43	5.4
c2670	6288	4.17	7.9
c3540	14187	5.38	10.4
c5315	16112	8.92	15.1
c6288	95642	16.2	17.7
c7552	25430	14.4	22.1

TABLE IV: Initial upper bound on switching activity

The initial upper bounds on switching activity for the ISCAS85 circuits are shown in Table IV. The CPU time is for an UltraSparc1/140 workstation. It includes parsing and building the constraint system.

The initial upper bound is compared to the lower bounds obtained by simulation in Table V. It shows that the initial upper bound is about 280% of the lower bound¹. The closest upper bound of 161% was obtained for the c6288 circuit and the most distant one of 390% for c1908. The most reasonable explanation is that c6288 which is a multiplier has fewer long false paths than the controller c1908.

Circuit	Switching activity [transitions]		Ratio UI/L [%]
	L - Lower bound by simulation for $13.08 \cdot 10^6$ vectors	UI - Initial upper bound by constraint resolution	
c432	412	903	219
c499	273	785	288
c880	940	2316	246
c1355	1286	4985	388
c1908	2242	8753	390
c2670	2558	6288	246
c3540	4532	14187	313
c5315	5725	16112	281
c6288	59349	95642	161
c7552	8612	25430	295

TABLE V: Comparison of lower bounds and the initial upper bounds

4-3. Case analysis on nets sorted by decreasing fanout

4-3.1 FANOUT without learning for 1000 decisions

The case analysis on nets sorted by decreasing fanout is the simplest heuristic we used (called *FANOUT*).

1. The percentage is not weighted by the size of the circuit.

The experimental results are shown in Table VI. The progress of the case analysis is shown on two graphs for each circuit in Figures 113-123. The first one denoted “SAV” - switching activity value - shows how the upper bound on switching activity is lowered by each step of the case analysis. “Level” indicates the depth of the decision tree.

Circuit	Upper bound on SAV			Case Analysis		Total	
	Initial	Final	F/I%	CPU [s]	MEM [MB]	CPU	MEM [MB]
c17	15	14	93	<1	0.016	<1	0.133
c432	903	831	92.0	328	0.523	330	1.734
c499	785	474	60.38	155	0.391	157	1.891
c880	2316	2233	96.42	478	0.602	480	3.289
c1355	4985	3917	78.58	368	0.453	371	4.258
c1908	8753	7992	91.31	864	0.578	868	6.023
c2670	6288	5629	89.52	1811	1.164	1816	9.078
c3540	14187	12980	91.49	4984	1.250	4990	11.664
c5315	16112	15472	96.03	733	0.766	742	15.844
c6288	95642	95548	99.90	4809	2.234	4829	19.914
c7552	25430	23189	91.19	6372	2.188	6385	24.297

TABLE VI: Case analysis on all nets sorted by fanout, 1000 decisions

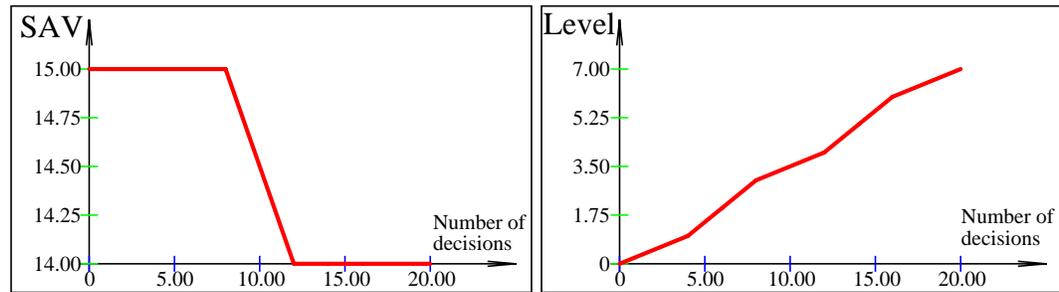


FIGURE 113: Progress of case analysis, c17, fanout, up to 1000 decisions

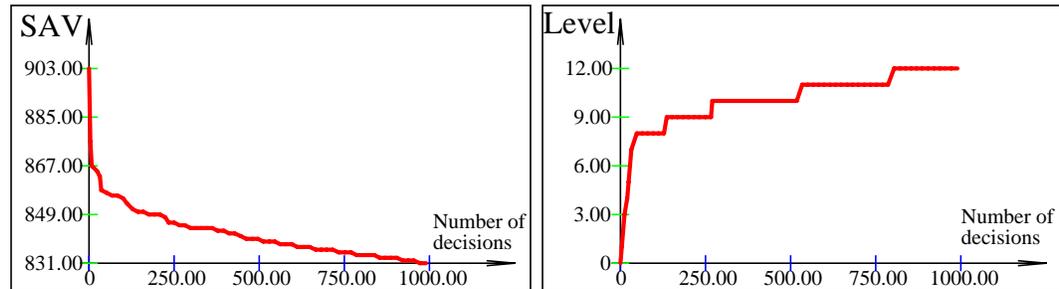


FIGURE 114: Progress of case analysis, c432, fanout, up to 1000 decisions

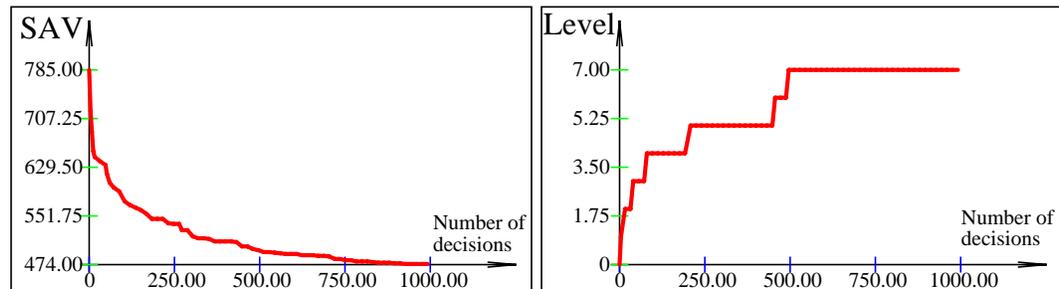


FIGURE 115: Progress of case analysis, c499, fanout, up to 1000 decisions

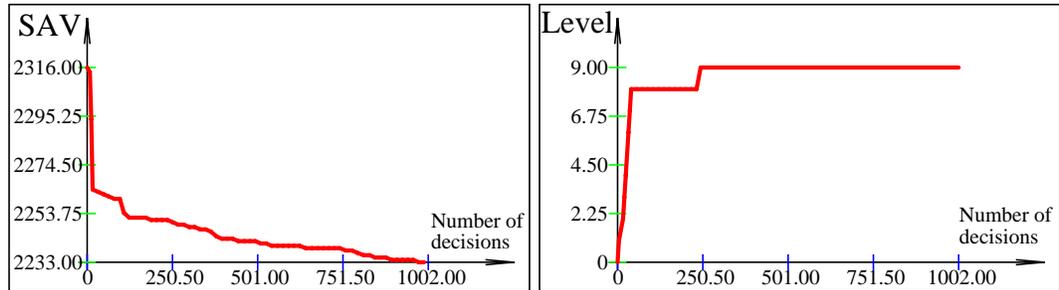


FIGURE 116: Progress of case analysis, c880, fanout, up to 1000 decisions

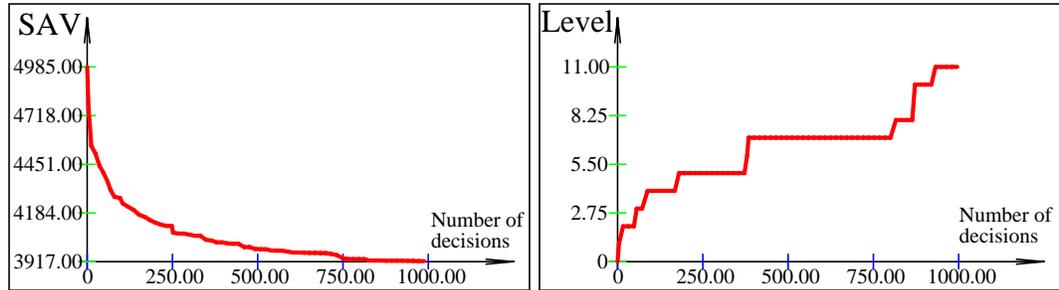


FIGURE 117: Progress of case analysis, c1350, fanout, up to 1000 decisions

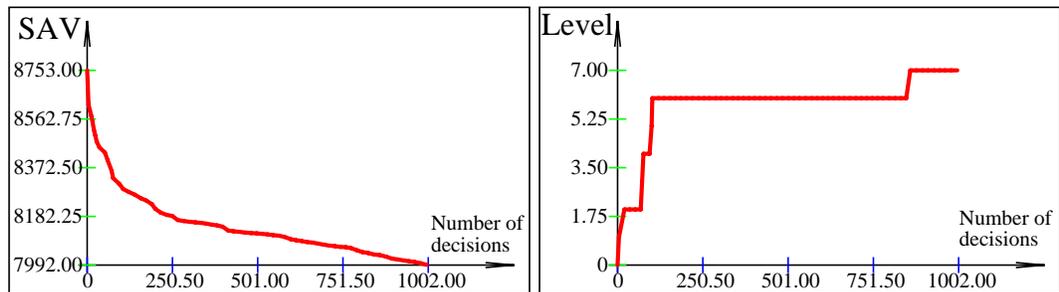


FIGURE 118: Progress of case analysis, c1908, fanout, up to 1000 decisions

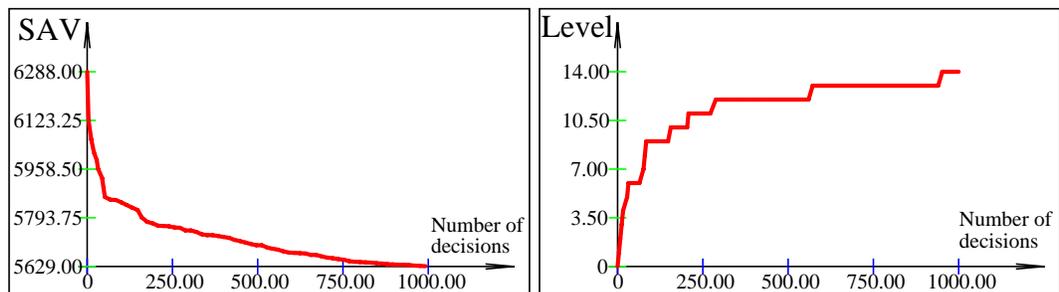


FIGURE 119: Progress of case analysis, c2670, fanout, up to 1000 decisions

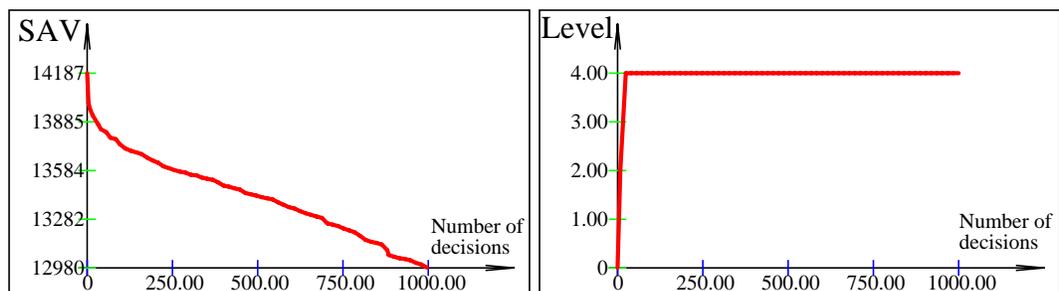


FIGURE 120: Progress of case analysis, c3540, fanout, up to 1000 decisions

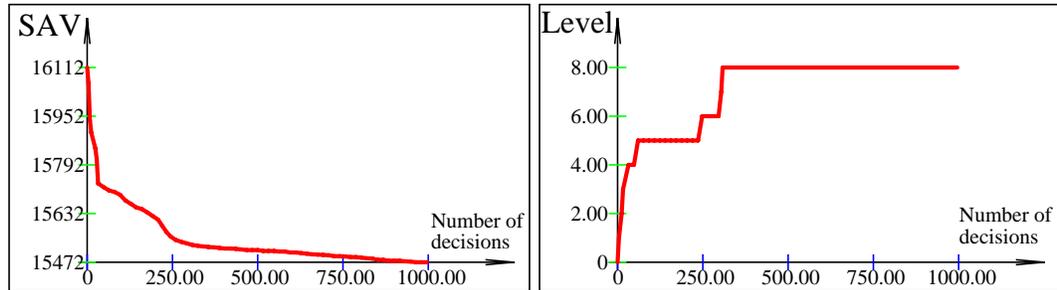


FIGURE 121: Progress of case analysis, c5315, fanout, up to 1000 decisions

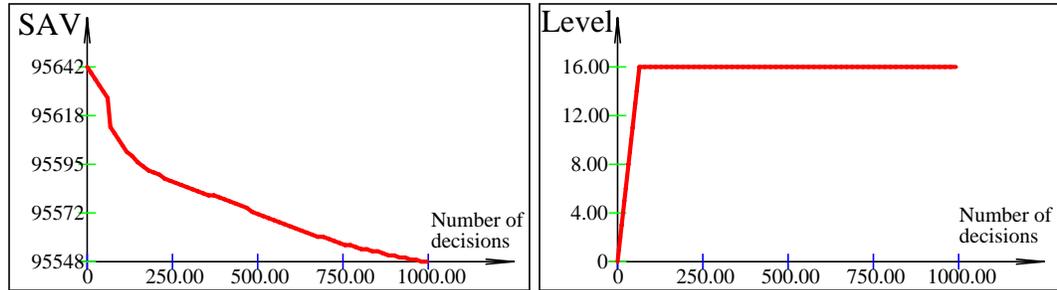


FIGURE 122: Progress of case analysis, c6288, fanout, up to 1000 decisions

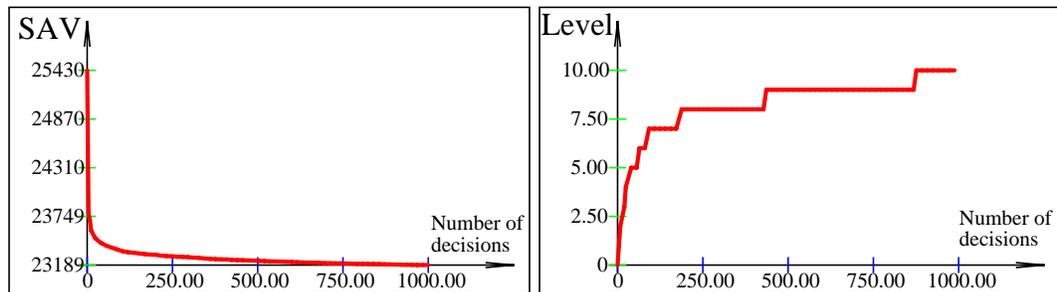


FIGURE 123: Progress of case analysis, c7552, fanout, up to 1000 decisions

The SAV decreases with the number of decisions. The speed of decrease (first derivative) decreases as well. This is because making a decision may result in four new leaves of the decision tree, of which 3 have the SAV as the parent node and the last one has a lower SAV. Then 3 more subtrees must be explored before the upper bound drops. Such exponential behavior is observed in e.g., c499, c1350, c2670, and c7552. The best case behavior is linearly decreasing SAV, when each decision results in four leaves with lower SAVs than the parent node. No circuit showed this best-case behavior, but c3540 is the closest. Note that not all decisions are equally “powerful” because all heuristics sort the nets where the decisions are applied so that the “best” nets are used first. On some circuits one can see that the first few decisions were very useful, dropping the upper bound rapidly, as e.g., in c880, c5315.

Observing the number of levels of the case analysis is even more interesting. The level corresponds to the number of constraints applied at the same time. With more constraints the waveforms sets are smaller, up to the simulation with one simple waveform on each circuit net. Typically, when the levels grow rapidly, the SAV decreases rapidly too as, e.g., for the first 100 decisions in c880 and c6288. When the heuristic hits the exponential growth of the number of leaves with the same SAV the level remains the same for a long time, as in c880 and c7552.

4-3.2 FANOUT without learning for 10000 decisions

In this section we present the results for the same heuristic (FANOUT) as in the previous section, but for 10000 decisions.

Circuit	Upper bound on SAV			Case Analysis		Total	
	Initial	Final	F/I%	CPU[s]	ME M [MB]	CPU	MEM [MB]
c17	15	14	93	<1	0.016	<1	0.133
c432	903	795	88.0	3787	4.844	4023	6.102
c499	785	452	57.6	1045	4.742	1253	6.281
c880	2316	2190	94.56	6741	4.180	6962	6.899
c1355	4985	3856	77.35	3843	4.719	4071	8.555
c1908	8753	7513	85.83	9049	3.953	9279	9.406
c2670	6288	5487	87.26	14656	5.953	14876	13.875
c3540	14187	11777	83.01	50976	4.703	51234	15.109
c5315	16112	15211	94.41	7764	4.289	8002	19.336
c6288	95642	95476	99.83	69017	6.938	69271	24.594
c7552	25430	23033	90.57	67759	5.719	67990	27.797

TABLE VII: Case analysis on all nets sorted by fanout, 10000 decisions

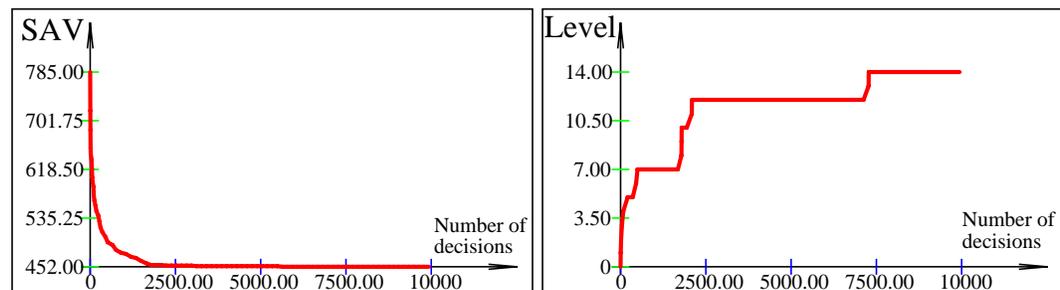


FIGURE 124: Progress of case analysis, c499, fanout, up to 10000 decisions

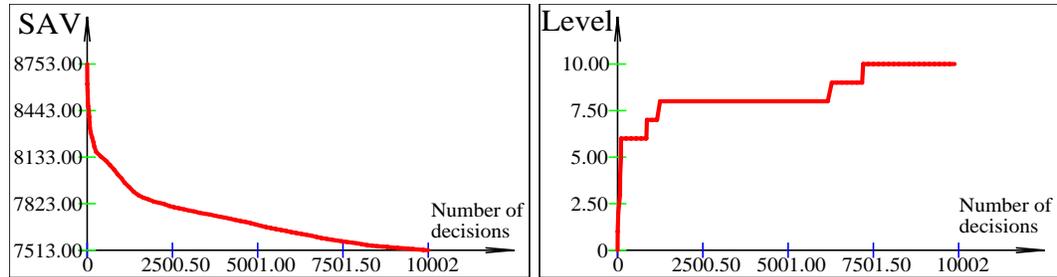


FIGURE 125: Progress of case analysis, c1908, fanout, up to 10000 decisions

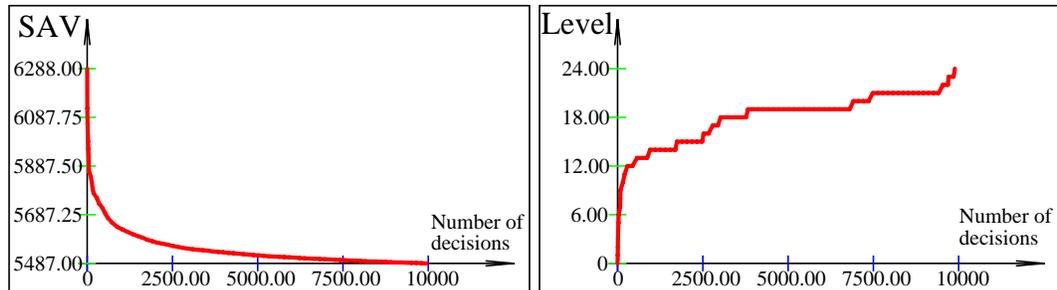


FIGURE 126: Progress of case analysis, c2670, fanout, up to 10000 decisions

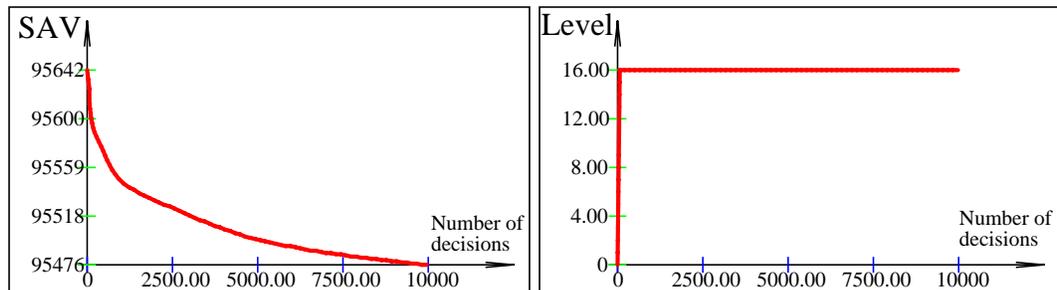


FIGURE 127: Progress of case analysis, c6288, fanout, up to 10000 decisions

For most circuits, one can observe a similar effect as identified in the previous section: After the “best” decisions are done, the upper bound and the level change less and less. Typical examples are circuits c499 and c7552, where making more than 1500 decisions brings a negligible decrease in the upper bound. The same is suggested by the SAV graph of c2670, but the level continues to increase, indicating progress in eliminating transitions, which may later show as a decrease in the upper bound. Similarly for c1908. Most graphs look similar to those in the previous section, therefore we present only few of them in Figures 124 to 127. An experiment with even more decisions is presented later in the thesis.

4-3.3 FANOUT with learning for 1000 decisions

In this section we present the results for the heuristic FANOUT as in Section 4-3.1, but we include learning of global implications and use them to improve propagation of decisions towards primary inputs.

Circuit	Upper bound on SAV				Case Analysis		Total	
	Initial	Final	F/1%	F/nL%	CPU [s]	MEM [MB]	CPU	MEM [MB]
c17	15	14	93	100	<1	0.031	<1	0.156
c432	903	831	92.0	100	326	0.508	331	2.539
c499	785	474	60.4	100	153	0.375	156	2.383
c880	2316	2233	96.42	100	472	0.586	476	3.953
c1355	4985	3916	78.56	99.97	367	0.453	384	5.398
c1908	8753	7992	91.31	100	845	0.570	881	8.945
c2670	6288	5629	89.52	100	1794	1.156	1821	11.250
c3540	14187	12980	91.49	100	4982	1.242	6106	22.344
c5315	16112	15472	96.03	100	730	0.750	778	19.399
c6288	95642	95541	99.89	99.99	4628	2.242	44959	49.071
c7552	25430	23189	91.19	100	6260	2.180	6397	31.719

TABLE VIII: Case analysis on all nets, fanout, learning, 1000 decisions

Field F/nL in Table VIII is the ratio of the final SAV values with (Table VIII) and without (Table VI) learning. The graphs SAV, Level = function(number of decisions) are not shown (except of c1355 as an example) because visual inspection cannot distinguish them from those shown in Section 4-3.1. That indicates that the use of learning does not considerably change the analysis. Use of learning does however bring some improvement as shown in Table VIII. The improvement will also be discussed later in Section 4-6.

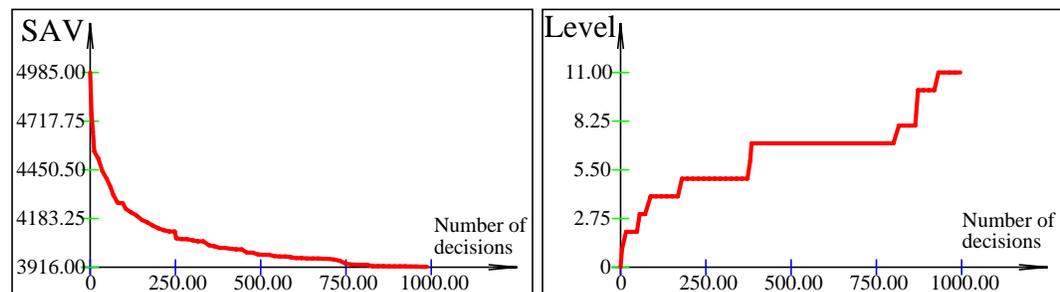


FIGURE 128: Progress of case analysis, c1355, fanout, learning, 1000 decis.

4-3.4 FANOUT with learning for 10000 decisions

In this section we present the results for heuristic FANOUT as in Section 4-3.2, but with learning of global implications.

Circuit	Upper bound on SAV				Case Analysis		Total	
	Initial	Final	F/1%	F/nL%	CPU[s]	MEM [MB]	CPU	MEM [MB]
c17	15	14E	93	100	<1	0.031	<1	0.156
c432	903	795	88.0	100	3477	4.828	3713	6.891
c499	785	452	57.6	100	1007	4.734	1216	6.758
c880	2316	2190	94.56	100	6609	4.148	6835	7.516
c1355	4985	3855	77.33	99.97	3608	4.703	3847	9.641
c1908	8753	7513	85.83	100	8411	3.945	8667	12.281
c2670	6288	5480	87.15	99.87	14197	5.789	14430	15.852
c3540	14187	11777	83.01	100	45738	4.695	47144	25.750
c5315	16112	15211	94.41	100	7383	4.273	7647	22.891
c6288	95642	95464	99.81	99.99	63674	7.086	104738	53.899
c7552	25430	23033	90.57	100	61506	5.703	61849	35.211

TABLE IX: Case analysis on all nets, fanout, learning, 10000 decisions

As in the previous section, visual inspection of these graphs does not show any differences compared to those in Section 4-3.2. The only exception is c2670. The maximum level with learning is actually lower than without learning, and the SAV is lower. This is because the learned implications are additional constraints, and thus a lower upper bound on switching activity can be achieved with fewer steps of the case analysis. The benefit of learning on all ISCAS85 circuits is summarized later in Section 4-6.

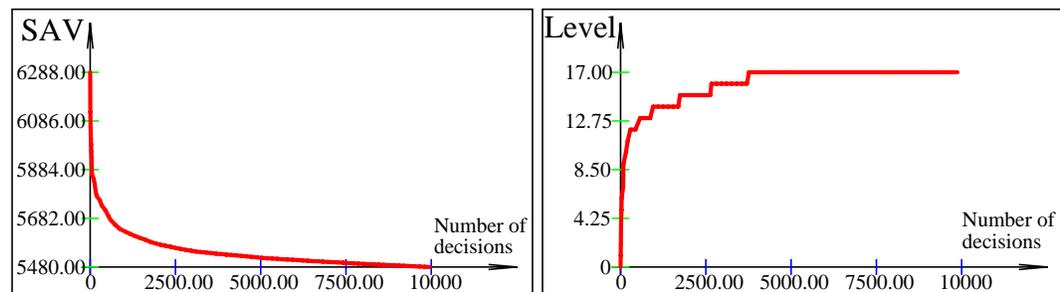


FIGURE 129: Progress of case analysis, c2670, fanout, learning, 10k decis.

4-4. Case analysis on primary inputs sorted by decreasing fanout

4-4.1 PIFAN without learning for 1000 decisions

In this section we present a table of results for heuristic PIFAN (primary inputs sorted by fanout). The case analysis was done for 1000 decisions. Column $F/Fa\%$ compares the final SAV of heuristics PIFAN (Table X) and FANOUT (Table VI). Note that PIFAN was better (resulted in a lower bound) in one case, worse in 6.

Circuit	Upper bound on SAV				Case Analysis		Total	
	Initial	Final	F/I%	F/Fa%	CPU[s]	MEM [MB]	CPU	MEM [MB]
c17	15	14	93	100	<1	0.016	<1	0.133
c432	903	834	92.4	100.36	153	0.438	155	1.648
c499	785	782	99.62	164.98	17	0.320	20	1.820
c880	2316	2243	96.85	100.45	354	0.648	357	3.336
c1355	4985	4899	98.27	125.07	232	0.508	235	4.313
c1908	8753	7138	81.55	89.31	1941	0.719	1945	6.164
c2670	6288	6020	95.74	106.95	1221	1.156	1226	9.070
c3540	14187	12980	91.49	100	4908	1.219	4913	11.633
c5315	16112	15472	96.03	100	729	0.727	738	15.805
c6288	95642	95548	99.90	100	4749	2.195	4762	19.875
c7552	25430	23414	92.07	100.97	8798	2.297	8811	24.406

TABLE X: Case analysis on PIs sorted by fanout, 1000 decisions

4-4.2 PIFAN with learning for 1000 decisions

In this section we present a table of results for heuristic PIFAN as in the previous section, but with learning of global implications. Column $F/nL\%$ compares the final SAV of heuristic PIFAN with (Table XI) and without learning (Table X). In three cases, learning resulted in a lower SAV. Note that learned implica-

Circuit	Upper bound on SAV				Case Analysis		Total	
	Initial	Final	F/I%	F/nL%	CPU[s]	MEM [MB]	CPU	MEM [MB]
c17	15	14	93	100	<1	0.016	<1	0.141
c432	903	834	92.4	100	154	0.430	160	2.461
c499	785	782	99.6	100	18	0.305	21	2.313
c880	2316	2243	96.85	100	353	0.641	358	4.008
c1355	4985	4899	98.27	100	233	0.508	250	5.453
c1908	8753	7136	81.53	99.97	1941	0.719	1975	9.094
c2670	6288	6020	95.74	100	1221	1.148	1247	11.242
c3540	14187	12980	91.49	100	4916	1.211	6041	22.313
c5315	16112	15472	96.03	100	730	0.719	778	19.367
c6288	95642	95541	99.89	99.99	4616	2.203	44558	49.031
c7552	25430	23402	92.03	99.95	8788	2.289	8924	31.828

TABLE XI: Case analysis on PIs, fanout, with learning, 1000 decisions

tions help in spite of decisions being made on primary inputs only. Each decision propagates first from a PI toward POs, and then backward (toward another PI) along the learned implication.

4-5. Case analysis on closing nets and primary inputs sorted by decreasing fanout

4-5.1 RCVFAN without learning for 1000 decisions

In this section we present a table of results for heuristic RCVFAN (closing nets of all reconvergent regions sorted by fanout). The case analysis was done for 1000 decisions. Column $F/Fa\%$ compares the final SAV of heuristics RCVFAN (Table XII) and FANOUT (Table VI). The RCVFAN resulted in better SAV in two circuits, worse in four circuits.

Circuit	Upper bound on SAV				Case Analysis		Total	
	Initial	Final	F/I%	F/Fa%	CPU[s]	MEM [MB]	CPU	MEM [MB]
c17	15	14	93	100	<1	0.016	<1	0.141
c432	903	834	92.4	100.36	153	0.430	159	2.453
c499	785	718	91.5	151.48	77	0.570	80	2.570
c880	2316	2243	96.85	100.45	378	0.641	382	3.992
c1355	4985	3917	78.58	100	362	0.445	379	5.359
c1908	8753	7430	84.89	92.97	1297	0.711	1331	9.055
c2670	6288	5934	94.37	105.42	1621	1.094	1647	11.125
c3540	14187	12980	91.49	100	4905	1.258	6070	22.290
c5315	16112	15472	96.03	100	736	0.742	782	19.305
c6288	95642	95548	99.90	100	4726	2.188	44880	48.914
c7552	25430	23181	91.16	99.97	6305	2.141	6435	31.547

TABLE XII: Case analysis on closing nets, 1000 decisions

4-5.2 RCVFAN with learning for 1000 decisions

In this section we present a table of results for heuristic RCVFAN (closing nets of all reconvergent regions sorted by fanout) as in the previous section but with learning of global implications.

Column $F/nL\%$ compares the final SAV of heuristic RCVFAN with (Table XII) and without learning (Table XIII). In three cases, learning resulted in a lower SAV. The results for different heuristics are summarized in the next section.

Circuit	Upper bound on SAV				Case Analysis		Total	
	Initial	Final	F/I%	F/nL%	CPU[s]	MEM [MB]	CPU	MEM [MB]
c17	15	14	93	100	<1	0.016	<1	0.141
c432	903	834	92.4	100	156	0.430	162	2.461
c499	785	718	91.5	100	78	0.570	81	2.578
c880	2316	2243	96.85	100	378	0.641	382	4.008
c1355	4985	3916	78.56	99.97	371	0.445	388	5.391
c1908	8753	7374	84.25	99.25	1604	0.727	1638	9.102
c2670	6288	5934	94.37	100	1615	1.094	1641	11.188
c3540	14187	12980	91.49	100	4904	1.258	6034	22.360
c5315	16112	15472	96.03	100	729	0.742	777	19.391
c6288	95642	95541	99.89	99.99	4620	2.203	44645	49.031
c7552	25430	23181	91.16	100	6247	2.141	6383	31.680

TABLE XIII: Case analysis on closing nets of reconvergent regions, with learning, 1000 decisions

4-6. Comparison of heuristics for net selection

In the preceding sections we presented experimental results obtained using several heuristics for the selection of nets for the case analysis. We compare the effectiveness of the heuristics in this section.

For *small circuits* the best is definitely **Fanout with learning**. It gives better or comparable results to **Fanout** without learning and comparable CPU times. For c1908, the best heuristic is **PIs with learning**. For *larger circuits* Fanout with learning is comparable to the other two heuristics **with learning** and for the *2 largest circuits* the selection based on **Closing nets with learning** provides the best results. In the case of c3540 and c5315 where all heuristics returned the same upper bound, the **PIs** without learning is the fastest followed by **Fanout** without learning.

Note that in Table XIV, the best (lowest) upper bound(s) for each circuit is/are bold (red in the color version).

In Figure 130, we show how all the heuristics compare for circuit c1908 in both the number of decisions and the CPU time. It is a composition of Figures 118, 125, and five other figures not presented here. The figure shows that the contribution of learning was marginal in c1908 compared to the choice of the heuristic, of which the one based on PIs performed the best. The c1908 circuit was actually the only one among the ISCAS85 circuits in which case analysis on primary inputs outran the two other heuristics. This is ascribed to the topology and function of c1908. There are two primary inputs in c1908 which propagate the constraints deeply into the circuit.

Circuit	Upper bound on switching activity [transitions] and total CPU time [seconds] for 1000 decisions												
	Initial	Fanout				PIs				Closing nets			
		Normal		With learning		Normal		With learning		Normal		With learning	
		UB	CPU	UB	CPU	UB	CPU	UB	CPU	UB	CPU	UB	CPU
c17	15	14	0.28	14	0.30	14	0.25	14	0.27	14	0.28	14	0.29
c432	903	831	330	831	331	834	155	834	160	834	159	834	162
c499	785	474	157	474	156	782	20	782	21	718	80	718	81
c880	2316	2233	480	2233	476	2243	357	2243	358	2243	382	2243	382
c1355	4985	3917	371	3916	384	4899	235	4899	250	3917	379	3916	388
c1908	8753	7992	868	7992	881	7138	1945	7136	1975	7430	1331	7374	1638
c2670	6288	5629	1816	5629	1821	6020	1226	6020	1247	5934	1645	5934	1641
c3540	14187	12980	4990	12980	6106	12980	4913	12980	6041	12980	6070	12980	6034
c5315	16112	15472	742	15472	778	15472	738	15472	778	15472	782	15472	777
c6288	95642	95548	4823	95541	44959	95548	4762	95541	44558	95548	44880	95541	44645
c7552	25430	23189	6385	23189	6397	23414	8811	23402	8924	23181	6435	23181	6383

TABLE XIV: Comparison of heuristics for net selection

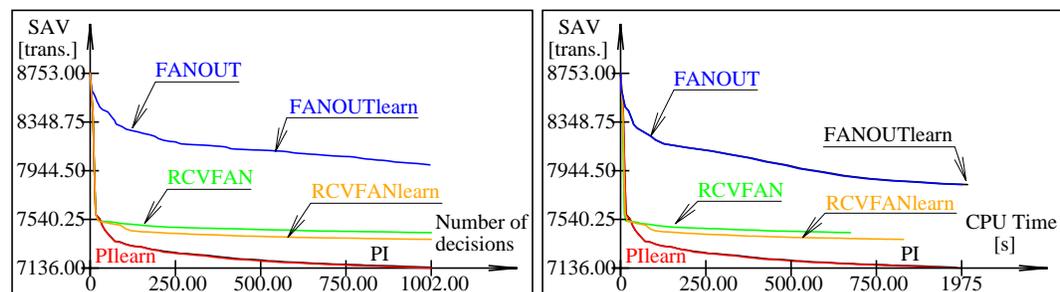


FIGURE 130: Comparison of heuristics in c1908 - 1002 decisions, CPU time

There is no single heuristic which would be the best for any circuit. Based on the experience with the ISCASS'85 circuits we can recommend FANOUT with learning.

4-7. Comparison of PCA and HPCA

The results below show that HPCA is slightly faster than PCA, takes more memory for a small number of decisions, but less memory for a large number of decisions. The CPU time and memory requirements are for the case analysis only, loading and parsing are excluded. The results are summarized in Table XV, with details provided in Tables XVI to XIX. Since both case analyses perform the same algorithm, the upper bounds on switching activity are the same. For the 11 ICSAC'85 circuits, HPCA was 1.004 times faster, but it needed 1.12 times more memory for 1000 decisions. However, for 10000 decisions, HPCA was 1.005 times faster, but needed 3.38 times less memory than PCA. The difference is because of the exponential growth in the amount of memory relative to the number of decisions for PCA compared to linear growth for HPCA.

Heuristics	1000 decisions			10000 decisions		
	UB	CPU[s]	Mem[MB]	UB	CPU[s]	Mem[MB]
PCA	same	32673	10.0	same	213106	49.9
HPCA		32538	11.2		212146	14.8

TABLE XV: Comparison of PCA and HPCA, 11 ISCAS'85 circuits in total

Name	UB [transitions]	CPU [sec]	Mem- ory [MB]
c1355	3917	533	0.445
c17	14	<1	0.012
c1908	7992	1330	0.566
c2670	5629	2789	1.152
c3540	12980	7746	1.234
c432	831	473	0.504
c499	474	220	0.371
c5315	15472	1149	0.754
c6288	95548	7844	2.227
c7552	23189	9891	2.176
c880	2233	698	0.582
Sum	168265	32673	10.023

TABLE XVI: PCA, 1000 decisions

Name	UB[tran- sitions]	CPU [sec]	Mem- ory [MB]
c1355	3917	533	0.559
c17	14	<1	0.391
c1908	7992	1312	0.691
c2670	5629	2748	1.180
c3540	12980	7740	1.387
c432	831	484	0.562
c499	474	224	0.504
c5315	15472	1156	0.840
c6288	95548	7888	2.199
c7552	23189	9737	2.273
c880	2233	714	0.645
Sum:	168265	32538	11.231

TABLE XVII: HPCA, 1000 decisions

Name	UB[tran- sitions]	CPU [sec]	Memory [MB]
c1355	3856	3367	4.711
c17	14	<1	0.008
c1908	7513	8140	3.938
c2670	5487	13204	5.930
c3540	11777	45114	4.688
c432	795	3312	4.820
c499	452	952	4.727
c5315	15211	7176	4.273
c6288	95476	64448	6.898
c7552	23033	61212	5.695
c880	2190	6181	4.164
Sum	165790	213106	49.852

TABLE XVIII: PCA, 10000 decisions

Name	UB[tran- sitions]	CPU [sec]	Memory [MB]
c1355	3856	3327	0.836
c17	14	<1	0.391
c1908	7513	7936	0.992
c2670	5487	12154	1.469
c3540	11777	45006	1.891
c432	795	3263	0.906
c499	452	830	0.797
c5315	15211	7091	1.148
c6288	95476	64764	2.742
c7552	23033	61662	2.625
c880	2190	6115	0.969
Sum	165790	212146	14.766

TABLE XIX: HPCA, 10000 decisions

4-8. Reaching the exact value

The case analysis either stops after the given number of decisions or reaches the exact value of switching activity SAV (Section 3-2.4 on page 96). In

Name	Init	LB	UB	U/I%	U/L%	CPU	MEM
alu2	14045	1067	1067E	8	100	6411	0.39
alu4	30207	2961	2961E	10	100	117219	0.72
cm138a	52	25	25E	48	100	1	0.04
cm162a	206	96	96E	47	100	359	0.14
cm163a	136	82	82E	60	100	32	0.06
cm85a	107	49	49E	46	100	125	0.11
cmb	144	76	76E	53	100	84	0.08
cu	162	67	67E	41	100	54	0.06
f51m	610	189	189E	31	100	240	0.17
parity	15	10	12	80	120	49253	239.28
pm1	83	55	55E	66	100	17	0.08
t481	11392	975	1630	14	167	250560	0.29
x2	159	66	66E	42	100	21	0.08

TABLE XX: SAV in MCNC circuits, PIFAN, no_progress=1000

the results presented so far the exact SAV was obtained only on circuit *c17*. In general, the exact value is obtained easier on small circuits or on circuits with few large reconvergent regions that have only few input lines. Whether the exact value is found also strongly depends on the choice of nets where the case analysis is performed (Section 4-6). Table XX shows that on a set of small MCNC circuits¹ the exact SAV was obtained fairly quickly on all but 2 circuits. A good example for analysis is circuit *parity*. It is a tree of *XOR* gates, therefore the number of decisions is comparable to the number of vectors in exhaustive simulation. The second circuit is *t481*, that has many false paths (note that U/I ratio is 14% !) and the largest number of PIs in the group.

1. FYI: Table XLII on page 191 shows that the exact value of PSAV (peak current) was obtained on all tested MCNC circuits. See Table XXXVII on page 189 for the topological properties of the MCNC circuits.

If the exact SAV is not reached then quality of the method can be evaluated by the ratio of the lower and upper bounds. Table XXI compares lower bounds on SAV obtained by simulation with the best upper bound for any heuristic (Table XIV). The ratio ranged from 161% to 318%. Since the simulation was far from exhaustive, we cannot conclude that the method overestimated

Circuit	Switching activity SAV [transitions]		Ratio UF/L [%]
	L - Lower bound by simulation for $13.08 \cdot 10^6$ vectors	UF - Final upper bound by constraint resolution	
c432	412	831	201
c499	273	474	174
c880	940	2233	238
c1355	1286	3916	305
c1908	2242	7136	318
c2670	2558	5629	220
c3540	4532	12980	286
c5315	5725	15472	270
c6288	59349	95541	161
c7552	8612	23181	269

TABLE XXI: Comparison of lower bounds and the final upper bounds

by 1.6 to 3 times, but rather that the true upper bound is same as the computed upper bound or up to 1.6 to 3 times smaller. Since the simulation was performed for several million vectors, it is likely that the lower bound is reasonably close to the exact SAV. Note that a high UF/L ratio (i.e., more false transitions) is obtained in control circuits with many false paths (e.g., c1908). Low UF/L ratio is obtained either on small circuits (like MCNC) or datapath circuits that do not have many false paths (e.g., multiplier c6288).

4-9. Use of timing constraints

The idea of using timing constraints to tighten the upper bound on the number of transitions was introduced in Section 3-7 on page 122. We used the information obtained as a lower bound by simulation, assuming that it is the exact value. We verified by using another method [CerZ94] that, e.g., for c1908 with unit delay assignment the delay is the same as obtained by simulation (370 units). In fact, for this demonstration it is not very important whether the used timing information is an upper bound or an estimate because we are trying to find out how many transitions can be eliminated using such additional timing constraints. Apparently, if the

used timing constraint is not an upper bound then the result of the switching activity analysis is not an upper bound either. We removed all transitions from the primary outputs after the circuit delay. Table XXII shows the topological and lower bound circuit delays in the ISCAS'85 circuits. Whenever the exact delay is lower than the topological delay, it offers the possibility of using this delay as an additional constraint.

Circuit	Circuit delay		Potential timing constraint?
	Topological	Lower bound	
c432	170	170	No
c880	240	240	No
c499	110	110	No
c1355	240	240	No
c1908	400	370	Yes
c2670	320	240	Yes
c3540	470	460	Yes
c5315	490	470	Yes
c6288	1240	1220	Yes
c7552	430	420	Yes

TABLE XXII: Topological delay and lower bound circuit delay

We tested the approach on two circuits, c1908 and c2670. In both cases the application of timing constraints showed a small improvement in the resulting upper bound after 1000 decisions, as summarized in Table

Circuit	Upper bound on switching activity		
	Initial	After CA	After CA with timing constraints
c1908	8753	7992	7985
c2670	6288	5629	5575

TABLE XXIII: Case analysis with timing constraints

XXIII. The improvement was 0.09% for c1908 and 0.96% for c2670. It is a tiny improvement but considering it incurs no cost it should be used, especially when both the timing and the power verification methods are part of one framework, and thus the time of the last transition on each net is available. Adding timing con-

straints on each net would be even better than in our simple experiment where we placed timing constraints only on the primary output(s) with the longest delays.

4-10. Comparison with other methods

Many commercial tools estimate power by simulation. Probabilistic methods overcome the input-pattern dependency of simulation. Such methods are simple and fast but cannot give upper bounds, only estimates of the average power. An exact method for computing switching activity was proposed in [LBSV94]. In spite of the efficient representation of timed Boolean functions by BDDs, the method is limited to small circuits. Among the pattern-independent methods that have been reported in the literature, the work of Kriplani [KrNH92, KNYH93, KrNH95] is the closest one to our approach (see Section 2-3.5.1 on page 77). The best approach for comparing our work with Kriplani's would be to use a circuit of the same netlist and delay mapping, apply both methods, and compare the ratios of the upper and lower bounds on power (U/L). However, this is not possible because of several factors:

- We would need access to the technology libraries used in [KrNH95] to convert switching activity figures to power,
- We determine the upper bound on the total number of switching events and, therefore, on the peak power over one clock cycle. [KrNH95] determines the upper bound on the peak current using a proprietary current mapping and, therefore, the upper envelope of the total switching current in the power bus,
- The U/L ratios for the basic algorithm which is used as a reference differ in [KrNH92] and [KNYH93].

It follows that the U/L ratios in the two cases are not directly comparable. The comparison in [ZCSR96] should be interpreted with these major differences between the two methods in mind. Please refer to Section 6-4.1 on page 185 for a more detailed, though still not very accurate comparison.

CHAPTER 5

Parallel Case Analysis

In this chapter, we describe a parallel implementation of the case analysis. This is achieved by distributing the exploration of decision subtrees over many workstations on a local network. The algorithm for parallel case analysis is presented in Section 5-1, followed by an example in Section 5-2 and a description of our C++ implementation over TCP in Section 5-3. The experimental results obtained using this implementation are summarized in Section 5-4. The general characteristics required for a successful application of this case analysis algorithm (and directly its C++ implementation) to other problems are discussed in Section 5-6.

5-1. Algorithm for parallel case analysis

The independence of subspaces as discussed at the end of Section 3-2.7 on page 99 enables us to parallelize the case analysis. We investigated two parallel algorithms and then implemented one of them. A static search-space-division algorithm is described in the next section, followed by a description of the implemented dynamic division algorithm.

5-1.1 Static search-space-division algorithm

The independence of subspaces of the search space brings up the possibility of evaluating them in parallel with a *statically-divided search space*, i.e., each processor explores a certain subtree of the decision tree. The maximum computed SAV over all processors is the resulting upper bound on switching activity. A more practical version of the algorithm could proceed as follows. Consider $N+1$ processors, one called *master* and the others *slaves*. The master processor runs the case analysis until

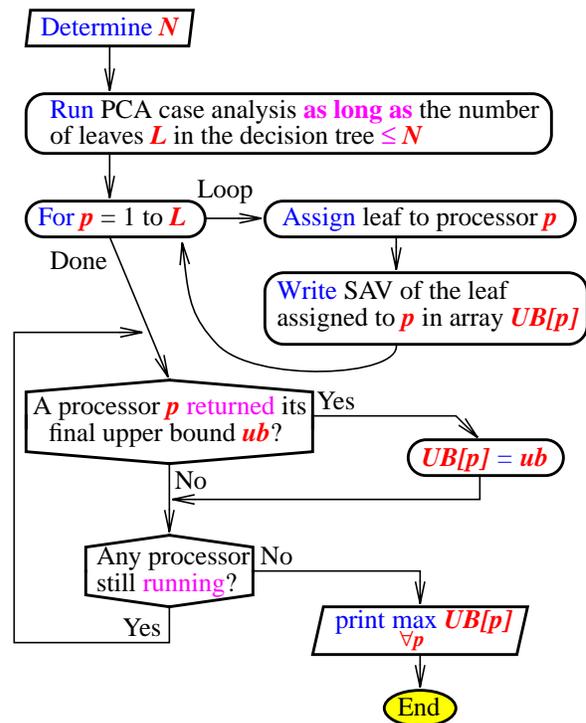


FIGURE 131: Parallel case analysis with static division of the search space

the number of leaves of the decision tree is as high as possible but not greater than N . Then it assigns each subtree to a slave. Slave processors apply any one of the case analysis algorithms mentioned in Section 3-2 on page 93. At the end (on timeout or exhausting the search subspace) all processors communicate the upper bound obtained in their subspace to the master processor. The master then computes the maximum over all returned SAVs. This algorithm is summarized in Figure 131.

The algorithm is fairly simple and does not need virtually any communication during the execution unless the user wishes to observe the progress of the global upper bound, rather than see it only at the end. Since the decision tree is distributed to local slaves, it lowers the memory requirements on each processor. However, when sub-trees assigned to fast processors terminate quickly as opposed to the other processors, the performance of the system may deteriorate, since more and more processors become idle.

Worse, there might be few subspaces or even only one subspace with a very high SAVs and all the others with much lower ones. The resulting upper bound would then depend only on that difficult sub-tree and the case analysis would proceed as on a single processor. It has been observed that the search space is generally not evenly distributed in the ISCAS'85 benchmarks [BrgF85]. An example of the decision tree for c1908 is shown in Figure 132. One can see that many branches stop after the third or fourth level, and the tree is actually much less symmetrical than it may seem from the figure.

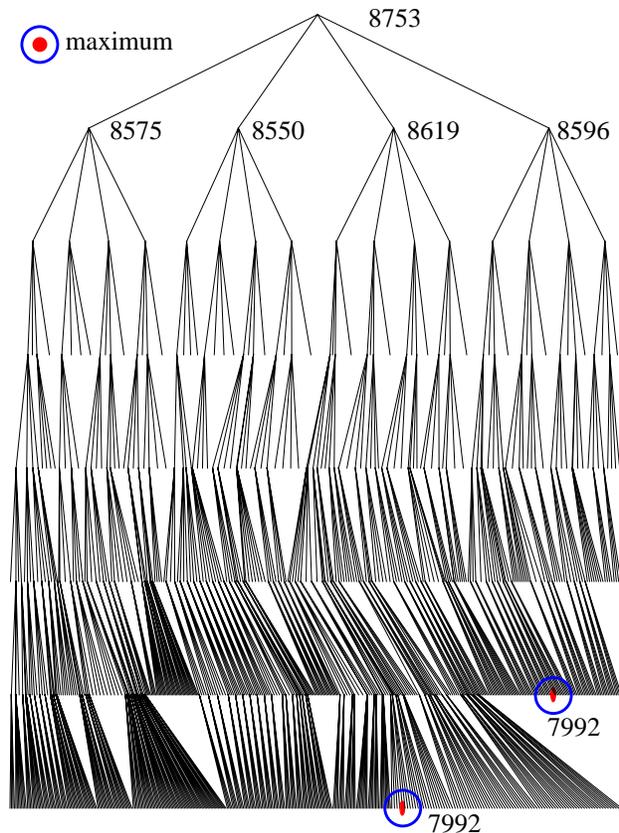


FIGURE 132: Decision tree - c1908, 1000 decisions

5-1.2 Dynamic search-space-division algorithm

Since one fixpoint calculation in the constraint system takes a considerable amount of CPU time (seconds to tens of seconds), we can refine the granularity of parallelism to smaller subspaces at the cost of increased communication. Furthermore, we can *dynamically reassign the subspaces* to processors, thus eliminating the biggest problem of the static algorithm - spending CPU time on subspaces that need no exploration. We could proceed as follows: First, choose a leaf with the highest SAV in the decision tree and assign a processor to carry out the analysis on that decision node. While the processor is working, take the second highest leaf and assign it to the second processor, and so on as long as there are available processors. Whenever

a processor finishes the analysis, replace the node in the decision tree by the subtree the processor returns and mark the processor as free (Figure 133).

An advantage of the algorithm is that CPU time is spent only on interesting subspaces of the search space. Yet, the strategy could degenerate if the decision tree has always the highest SAV only in one branch. However, this would be the ideal behavior for the case analysis! Recall that a level in the tree corresponds to one circuit net. After each decision, the algorithm descends one level in the decision tree, leading quickly (in linear time in the number of nets) to a test vector, and no parallelism is needed in that case.

There are two potential disadvantages - the cost of communication and the size of the decision tree. The master which maintains the entire tree communicates the node to be split (called a job) to a slave and awaits its reply. The slave's reply is the sub-tree created by the expansion of the node received from the master. Therefore, the length of the messages and the delay through the communication channel are critical.

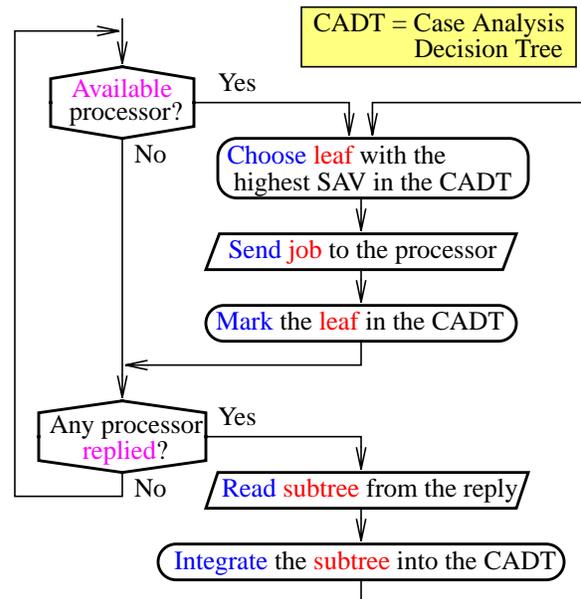


FIGURE 133: Dynamic parallel case analysis

The master must also send jobs to and receive replies from all the slaves with a minimum delay, otherwise some slaves would remain idle. It follows that such an implementation is useful and would have a nearly linear speedup if the following conditions are met:

1. The master can schedule jobs and process replies rapidly.
2. The time to make one decision on a slave is considerably longer than the com-

munication cost.

The implementation must be carefully designed with the first condition in mind. The master processor must be fast. The second condition is easy to satisfy here. Our experience indicates, that on the average, the processing of a

Media	Maximum raw throughput [kbits / s]	Reasonable practical throughput on a medium-loaded network [kBytes / s]
1Gb Ethernet	1000000	50000
ATM 620 Mbps	620000	40000
100 base T Ethernet	100000	5000
10 base T Ethernet	10000	500
ISDN	128	7
Serial cable	112	6
Modem 33.3	33.3	1

TABLE XXIV: Network media throughput

decision on a slave takes from 5 to 50 seconds for the larger ISCAS85 benchmarks. To allow, say, 10% of this time for communication, it gives us at least 500ms for both sending a job and returning the reply. Table XXIV shows how many bytes can be transferred in 1s over a medium-loaded network. If the total message length is *1kByte* we can use a network with modem connections, when it is *500kBytes* then the old 10baseT network is sufficient.

With a larger number of processors and/or a more complex network, the reliability of the system decreases. Therefore, a parallel system is much more prone to a failure than a single CPU. What happens in the case that a processor does not reply in the algorithms in Figures 131 and 133? The static algorithm (Figure 131) will loop forever. The dynamic algorithm (Figure 133) will work correctly.

We assume the following possible failures:

1. A slave processor does not accept communication.
2. A slave processor does not reply.
3. The master processor that schedules jobs fails.

The first and the second failures can cause a deadlock on the master. These two failures can be resolved by asynchronous communication with timeout. If the timeout occurs during any communication with a slave, the master will further ignore

that slave and reassign its job. Such a solution brings up another problem: a slave could reply after timeout. Therefore, each message must carry a tag to allow the master to discard stray messages.

The master processor could be protected against the third type of failure by maintaining multiple copies of the decision tree on several master processors. A much simpler solution is to have only one master and periodically save the current decision tree. Note that the saved decision tree must also include unfinished jobs which are currently assigned to slaves. A fail-safe version of the parallel case analysis algorithm is shown in Figure 134.

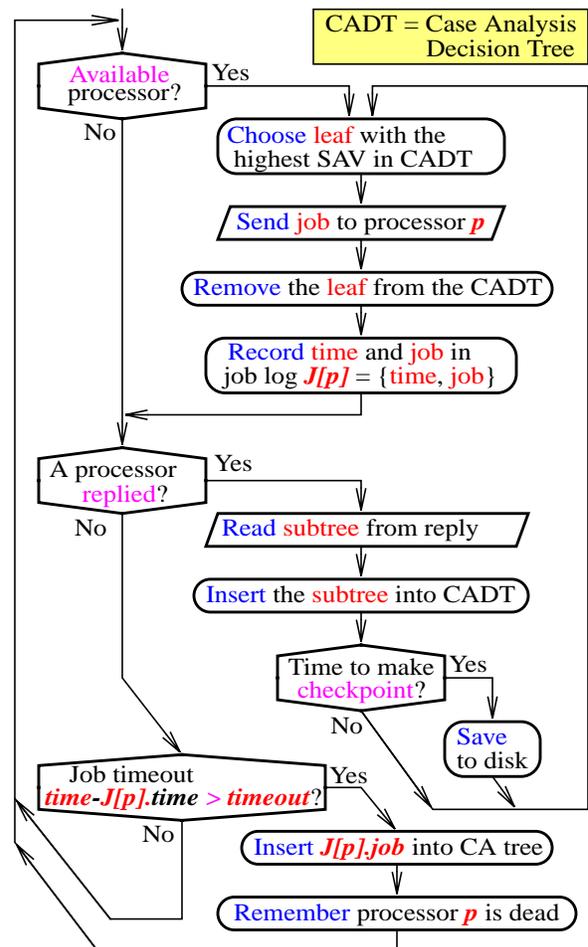


FIGURE 134: Fail-safe dynamic parallel case analysis

5-2. Example of Parallel Case Analysis

In this section we illustrate the dynamic case analysis algorithm from Section 5-1.2 on the *c17* circuit (Figure 76). We use a configuration consisting of two slaves *S1*, *S2* and one master (Figure 135).

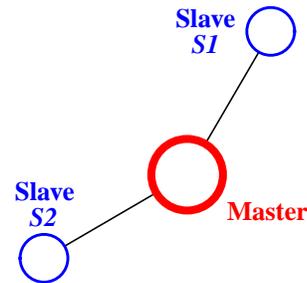


FIGURE 135: Network architecture - 2 slaves

The master starts first, parses the circuit, constructs the constraint system and computes the initial estimate *iSAV* which is the root of the decision tree. The master then waits for a slave

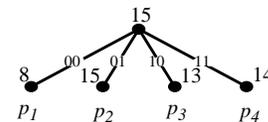


FIGURE 136: Parallel analysis - *c17*, first 4 paths

to connect. When a slave starts, it parses the circuit description and constructs its own copy of the constraint system. It also constructs a local list (the same on all slaves) of nets where the case analysis will be performed. It tries then to connect to the master. Let the first slave be *S1*. The master accepts the connection and sends the first job to *S1*. The job is the root of the decision tree, i.e., an empty path (with no constraints), denoted $p_0 = \{\}$. When the slave *S2* connects, the master has no job to give. While *S2* remains idle, *S1* makes four 1-level decisions starting from the received path p_0 . Making the first decision yields 4 paths, p_1, p_2, p_3, p_4 (Figure 136) which are returned to the master.

The master extends the root of the decision tree by the received paths. Now it has 4 paths p_1, \dots, p_4 , i.e., 4 available jobs and two idle slaves *S1*, *S2*. It selects the path with the highest SAV $p_2 = \{c01\}$ and sends it to a slave, say *S1*. Then it assigns the next highest path, $p_4 = \{c11\}$, to

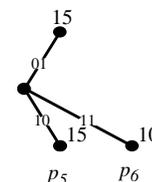


FIGURE 137: Parallel analysis - *c17*, second decision analysis

S2. If S1 finishes first it returns two paths to the master: $p_5=\{c01, c10\}$ and $p_6=\{c01, c11\}$ as shown in Figure 137. The two other paths, $\{c01, c00\}$ and $\{c01, c01\}$, represent inconsistent constraint assignments.

The master receives the paths p_5 and p_6 and merges them with the decision tree. The resulting tree is shown in Figure 138. The master thus schedules the path with the highest SAV, $p_5=\{c01, c10\}$, to S1.

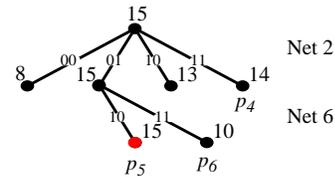
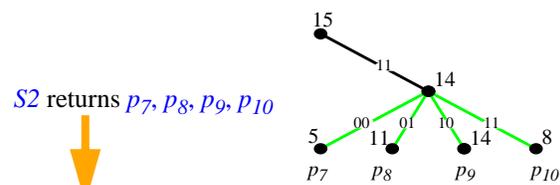


FIGURE 138: Parallel analysis - c17, after the second decision analysis

When S2 finishes extending the path $p_4=\{c11\}$, it returns paths p_7, p_8, p_9, p_{10} to the master that inserts them into the decision tree, as shown in Figure 139.



S2 returns p_7, p_8, p_9, p_{10}
Master inserts p_7, p_8, p_9, p_{10} into the decision tree

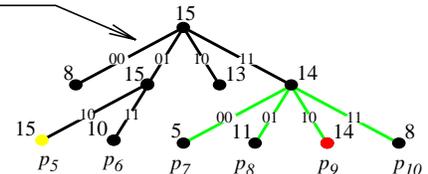
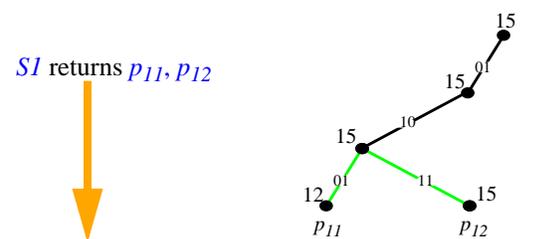


FIGURE 139: Parallel analysis - c17, the third decision analysis

The master assigns the path p_9 with SAV 14 to S2. Note that p_5 has the highest SAV of 15, but it has been assigned to S1. When S1 finishes extending p_5 , it returns $p_{11}=\{c01, c10, c01\}$ and $p_{12}=\{c01, c10, c11\}$. The master inserts the received p_{11} and p_{12} in the decision tree as shown in Figure 140.



S1 returns p_{11}, p_{12}
Master inserts p_{11}, p_{12} into the decision tree

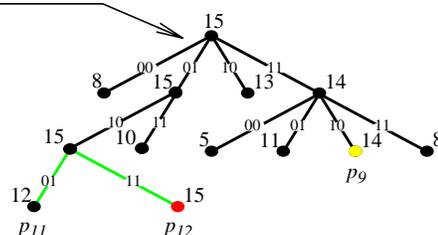


FIGURE 140: Parallel analysis - c17, the fourth decision analysis

The next scheduled path would be p_{12} to $S1$ and so on. Whenever a slave finishes the assigned job, the master inserts the resulting paths in the decision tree and assigns the highest-SAV path to the slave to keep all slaves busy. The execution trace for this example is shown in Figure 141.

There are many possible execution traces depending on the effective speed (considering the load by other users) of the slaves and the number of slaves. The parallel version performs a higher total number of decisions than the serial one (compare Figure 140 with the top part of Figure 73). However, we show in Section 5-5.1 that the increase in the number of decisions affects the overall performance only on small circuits.

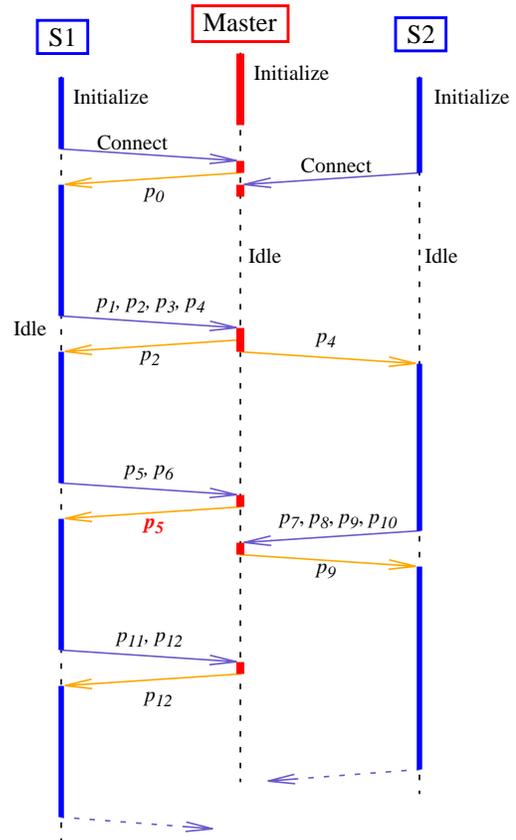


FIGURE 141: Parallel analysis - c17, execution traces

5-3. Implementation

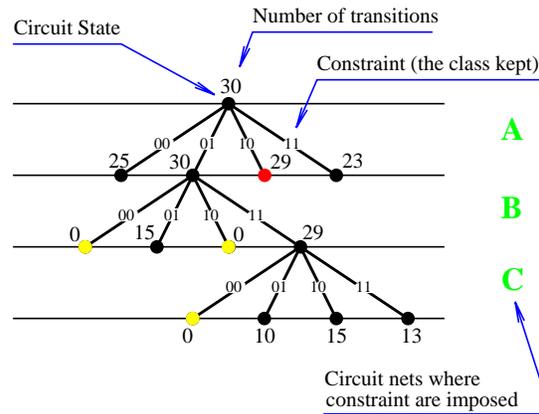
The parallel case analysis algorithm was implemented in C++ with network communication over TCP virtual channels. In this section we briefly describe the implementation.

A small prototype for testing job scheduling and network communication was initially written in Perl. Since the *ICproject* uses external C libraries for Verilog parsing, a different test task was used, namely, a DES decryption. The Perl prototype performs decryption by an exhaustive search over the space of keys. The space was divided statically, but the boundaries were calculated dynamically on the master to imitate our case analysis problem. The whole prototype was written in 1 week and has about 800 lines of code including an implementation of the fail-safe features. As expected, it demonstrated close-to-linear speed-up on a network of slaves heterogeneous in both performance and operating systems. Machines ranging from SparcStation IPC to UltraSPARC 1 and Pentium Pro were used. The operating systems were SunOS 4.1, 5.4, 5.5, Linux, HPUX, OSF/1, IRIX. Also, a combination of networks was used: 10BaseT Ethernet, 28.8kbps phone line and an Internet line going across North America.

We then proceeded with an implementation for the gate-level verifier. The most powerful case analysis algorithm HPCA (Section 3-2 on page 93) was used. In the next section we look more closely at the representation of the decision tree in HPCA.

5-3.1 The decision tree data structure

A heap was chosen for its $O(\log(n))$ time complexity to insert 1 element in the heap of size n , and an immediate access to the item with the highest key. The tree is represented by its leaves. One item in the heap is an encoding of the path from the root to the leaf (Figure 142). Note that a heap is not a canonical representation unlike the decision tree, but this is not essential for our algorithm. Each *path* is a list of constraints, each constraint is a waveform class.



Heap:

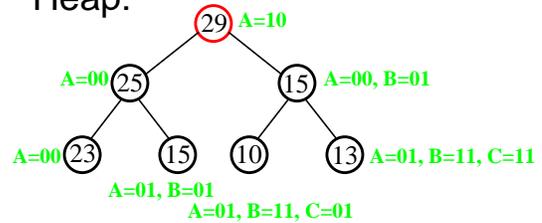


FIGURE 142: CA tree as a heap

Therefore, there can be up to 4 different constraints on each circuit net. If the order of the circuit nets is fixed, a very efficient path representation becomes possible, namely, a fixed-length packed array of two-bit elements, each representing a class constraint. The maximal path length is set to 128 constraints which was sufficient for even the longest analyses. With 2 bits per constraint, it requires 8 words on a 32-bit machine.

A heap entry also holds information on the SAV associated with the path and the number of constraints (Figure 143). All

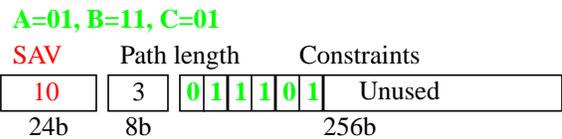


FIGURE 143: Path implementation

the data are translated into a network format (byte-oriented) before communication, to avoid problems with the little-versus big-endian architectural differences.

The heap is represented by the C++ class `ANALh_heap`, with typical access methods for the insertion and the removal of elements. The heap itself is not the usual simple dynamic structure or a single array. We opted for a combination of the two to minimize memory require-

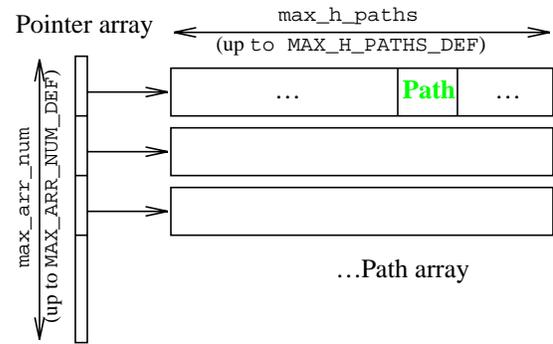


FIGURE 144: Heap architecture

ments and to reduce overhead due to repetitive allocation and release of dynamic memory. The heap is a 2-dimensional matrix of paths. One dimension (`max_arr_num`) is an array of pointers. Each points to an array of paths. The length of the path array (`max_h_paths`) is the smallest unit we allocate each time the heap grows. The heap can be constructed in two ways. Either the size is known ahead of time and then the exact heap dimensions are stored in the variables `max_arr_num` and `max_h_paths` (minimizing the number of arrays `max_arr_num`). Or the size is not known in advance, and then the absolute limits defined by the constants `MAX_ARR_NUM_DEF` and `MAX_H_PATHS_DEF` are used (Figure 144).

The individual path arrays are allocated dynamically as needed. The maximum heap capacity is `MAX_ARR_NUM_DEF` times `MAX_H_PATHS_DEF`, both set to 10000. This provides for allowing for the maximum capacity of 10^8 paths, allocating an array of 10000 paths at a time. Assuming that the size of 1 heap entry is about 40B on a 32-bit computer, the smallest allocated fixed-size heap occupies 1 pointer (say 4B) and one path array of size 1, i.e., about 44B. The smallest variable-size heap has the size of $40 * 10000 + 4 * 10000 = 429kB$ and grows in $40 * 10000 = 390kB$ increments. The largest heap can have the size of $44 * 10^8 + 4 * 10000 = 4.4GB$ or the size of the available virtual memory, whichever is smaller.

The average length of a message is one job description consisting of one path (40B), a heap with up to 4 paths (about 160B) is needed for the result if the deci-

sion analysis is 1-level deep or with up to 16 paths for a two-level decision (about 640B).

5-3.2 Implementation layers

The parallel case analysis is implemented in three layers: case analysis, scheduler, and network. A heap is used in both the master and the slaves. The master maintains the entire decision tree in one heap. It removes, translates, encodes its extensions into a message which it sends to a slave. The slave explores a sub-tree starting from the received path, encodes the resulting sub-tree (heap) and sends it back. The implementation is subdivided

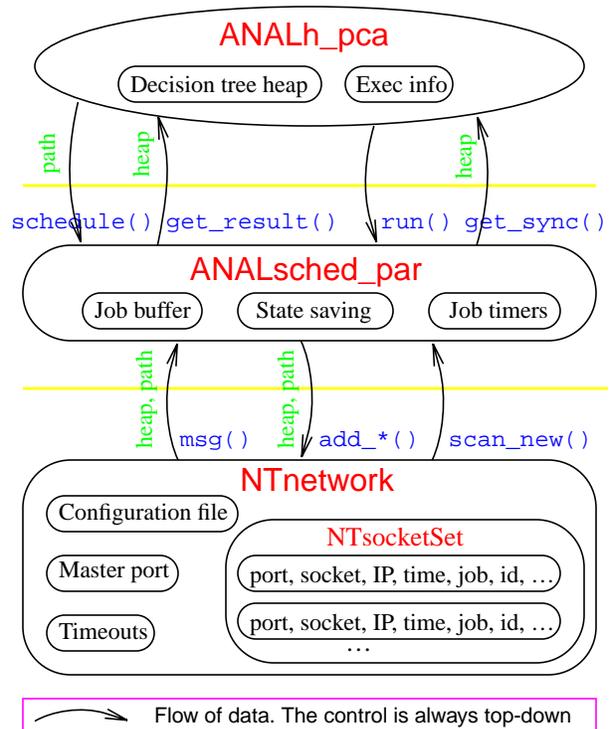


FIGURE 145: C++ architecture of parallel case analysis

into several C++ classes, to allow reasonably easy replacement of functional blocks. For example, the network interface is the class `NTnetwork` implemented over TCP/IP, but any communication layer can be plugged in as long as it preserves the interface (sending and receiving messages, and connection of new slaves at any time). The C++ architecture is shown in Figure 145.

All operations above the network layer `NTnetwork` are synchronous and triggered by the upper level. We explain how the interactions between the layers implement the algorithm for parallel analysis in the next section.

5-3.3 Master

The algorithm for the parallel case analysis (Figure 134) is implemented as follows: The master keeps this order of calls to the scheduler: `get_result()`; `schedule()`; `run()`. By calling `get_result()` the master recovers finished jobs from the scheduler and inserts the obtained sub-trees into the decision tree. The master removes a path with the highest SAV from the decision tree and inserts it into the scheduler's job buffer (which has the capacity of 10 jobs) using `schedule()`. By calling `run()` the case analysis gives a chance to the scheduler to distribute jobs and to receive results. The scheduler encodes paths as messages, and calls the network layer to send them and to receive replies. The scheduler decodes the received messages and stores the heaps in its buffer. The scheduler calls periodically `scan_new()`, which looks if a new host wants to connect as a slave. A slave is started by the user or a script as a process with the hostname of the master and the TCP port in the arguments. Whenever a slave connects, it is included in the list of active slaves and assigned a job in the next cycle. A slave can be disconnected by the user by killing the slave's process. This is detected by a timeout on the master during its periodical comparison of the time values in the table of assigned jobs and the current time. This mechanism allows for both dynamic resizing of the network and for resolving failures in slaves. Furthermore, a short timeout can be used to naturally select the fastest¹ machines from the set of available slaves.

5-3.4 Decision tree checkpoints

The master periodically saves its state for eventual recovery, including the decision tree, in a checkpoint file on a hard disk, should the machine where the master runs crash.

Periodically, after a certain number of decisions, the master calls `get_sync()` to recover the scheduled-but-not-finished paths and dumps the heap of the decision tree in a disk file after converting it into the network format. To save disk space,

1. "fastest" in the sense how many operations per second the user can get on that particular computer, considering all other load.

the dump file is a UNIX pipe through the compression program `gzip`. During start-up, the user can indicate whether to load the decision tree from a checkpoint file or whether to start from scratch.

5-3.5 Slave

A slave computer executes exactly the same code as the master, except that when HPCA is called, it verifies whether the user indicated on the command line that it is to be a slave. In that case the program awaits receiving a path, calls `extend_path()` to perform an extension by 1-level¹ decisions on the leaf determined by the path, and sends the resulting heap back to the master.

5-3.6 Network layer

The network layer is implemented over TCP virtual channels. The architecture is thus a virtual star with the master in the center. Each slave has a proprietary reliable buffered channel to the master. UPD, IP, or any other protocol could be used instead, however. We benefited from the TCP's virtual channels by saving on programming effort in identifying and acknowledging messages to create a reliable point-to-point connection. However, one very important aspect of the implementation is the asynchronous communication at the network layer. All the TCP ports are set as non-blocking which allows us to handle many ports from within one process. It also allows for dynamic reconfiguration - a slave can connect and disconnect at any time. Another benefit of non-blocking ports is timeout checking. If the master does not reply within a certain time interval, the slave terminates. If a slave does not reply within a time interval, the master reassigns the job and excludes the slave from its list of active slaves.

5-3.7 Object to message translation

Each data structure which is to be transmitted over the network has a method `o2msg()`. It dumps the content of the object into a binary buffer, while converting

1. 1-level, 2-level, ..., or more as specified by the user

all word-oriented data from the host format to the network byte format. A similar process is repeated whenever a message is received - each class has a constructor which reads a binary buffer. The created objects are kept in temporary buffers until a higher layer (e.g., the scheduler) takes them.

5-3.8 MIF export interface

The overview of the implementation would not be complete without mentioning some auxiliary service tasks such as the CPU execution time measurement and a Maker Interchangeable Format 4.0 export interface. The latter was used to generate many of the images used in this thesis. The core class called `EXPORTmif` provides basic functionality such as writing text, lines, rectangles, and other geometric objects. Many classes such as heap or abstract waveform have a method `print()` which displays the object in a textual form on standard output, and `mif()` which writes the graphical representation in the MIF log file. The PCA and HPCA case analyses as well as some other methods also produce graphical output (graphs and decision trees) in the MIF log file.

5-3.9 File descriptors

The number of slaves is theoretically limited by the number of existing ports, i.e., about 63000 on UNIX. However, each channel requires a socket and thus a file descriptor. Since our local network was not prepared for such a use, the default number of descriptors per process was set to 64 and the hard limit compiled into the UNIX kernels ranged from 256 to 1024 depending on the machine. It means, that on the computer where the master process runs the user has to set the limit on the number of descriptors to the expected maximum number of slaves. The hard limit can be changed only by re-compiling the kernel. Another way to allow more slaves is to close the connection after each transfer or use other means of communication than TCP.

5-4. Experimental results

We tested the parallel implementation on the ISCAS'85 benchmarks [BrgF85], summarized in Table II on page 124. Since the parallel exploration is almost independent¹ of the used circuit, we present the results for the c1908 only.

In this section we analyze the results obtained from the following experiments:

1. Switching analysis with 1000 decisions² on each configuration of 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10 computers of similar performance; c1908 only.
2. Switching analysis for constant time of 30 minutes on each configuration of 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10 computers of similar performance; c1908 only.
3. Switching analysis with the total of $3 \cdot 10^6$ decisions on 87 computers of various performance.

The raw CPU speed of each computer was converted to an equivalent number of SpareStations 10/45, based on the CPU run time for the HPCA in the single-processor mode on the c1908 and other small test circuit. This calibration is for HPCA only. Loading, parsing, and constraint system construction is excluded, because HPCA accounts for most of the execution time on larger circuits. However, all the measured CPU times of the experiments include all these functions. All the computers were on a local network connected by 10BaseT Ethernet. Before discussing the results we first define the basic measures of performance for parallel systems.

-
1. As mentioned in Section 5-1, the parallel algorithm does not use more than one CPU only on trivial circuits, but then the exact solution is obtained in time linear in the number of circuit nets; in all other cases, all CPUs are used
 2. The number of decisions is a user-specified limit. When the user asks for ND decisions, then up to $ND+3$ decisions must be performed to complete the decision analysis on one net

5-5. Performance measures for parallel processing

In this section we give a framework for evaluating the performance of parallel processing based on [Hwan93].

Instantaneous execution rate $R_{inst}(t)$ is the number of 1-level decisions a processor performs per second. The amount of *work* W in decisions that a processor can accomplish within real time T is $W = \int_{t=0}^T R_{inst}(t) dt$. The *average execution rate* or *performance* is $R = \frac{W}{T}$. More often we use notation $R(N)$ for a parallel system composed of N processors (or machines), each with performance R_i , where i is the processor identification. If $R_i = R_j$, for all $i \neq j$ then the parallel system is called *homogeneous*.

To compare *heterogeneous* parallel systems ($R_i \neq R_j$) we introduce the notion of *equivalent number of processors*. If a machine A has performance R_A and machine B performance R_B , we say that system B has *equivalent number of processors*

(*machines*) $p_{BA} = \frac{R_B}{R_A}$ with respect to the reference machine A .

For a heterogeneous system with N machines we define an *equivalent homogenous system* as the system with N reference machines, the same network architecture and running the same program as the heterogeneous system. The idea is to measure the performance on available heterogeneous systems, scale it to equivalent homogeneous systems and compare these scaled performances in terms of speedup and efficiency (will be defined later).

We base our scaling on the *maximum achievable performance* $RM(N)$. Consider a heterogeneous system composed of processors with performances R_1, R_2, \dots ,

R_N , with the corresponding equivalent number of processors of type A being p_{1A} , p_{2A} , ..., p_{NA} , respectively. The *maximum achievable performance* of the multi-

computer is $RM(N) = \sum_{i=1}^N R_i = \sum_{i=1}^N R_A p_{iA} = R_A \sum_{i=1}^N p_{iA} = R_A P$; it is a theoretical

limit on the performance of a given heterogeneous parallel system. The

equivalent homogeneous system has performance $ERM(N) = \sum_{i=1}^N R_A = NR_A$, or

we say that the heterogeneous system has *equivalent maximum achievable performance* $ERM(N)$. If a heterogeneous parallel system accomplishes work W in time

T , its *performance* is $R(N) = \frac{W}{T}$. If the workload is evenly¹ distributed, the system

with the higher (lower) maximum achievable performance will have the higher (lower) real performance. We approximate such scaling by the following

linear function: The equivalent homogeneous system has performance

$ER(N) = \frac{ERM(N)}{RM(N)} R(N) = \frac{N}{P} R(N) = \frac{NW}{PT}$, where $P = \sum_{i=1}^N p_{iA}$. Or we say that

the real heterogeneous system has the *equivalent performance* $ER(N) = \frac{N}{P} R(N)$.

The *speedup* of a heterogeneous system is $S(N) = \frac{R(N)}{R(1)}$. It indicates how many

times the system of N machines is faster from the user point of view compared to one machine with performance $R(1)$. It does not allow to compare objectively 2

different heterogeneous systems, however, because $R(N)$ does not scale linearly with the number of machines (considering a theoretical system with ideal parallel-

ism). To assess the quality of a parallel algorithm and its implementation we define

the *equivalent speedup* $S(N) = \frac{ER(N)}{R_A} = S(N) = \frac{NR(N)}{P R_A}$.

1. It assumes that a faster machine in a parallel system can accomplish more jobs (greater W over the total execution time) than a slower one. This is satisfied in our algorithm.

The *parallel system efficiency* $E = \frac{ES(N)}{N}$ describes how well the processors are used. The value of 1 means that the system achieved its maximum achievable performance $ERM(N)$, i.e., all machines are fully loaded. In practice $0 \leq E < 1$.

5-5.1 Analysis on a small number of computers

We carried out the analysis described in Section 4-3.1 on page 127 on circuit c1908 in the parallel mode on configurations of 1, 2, ..., 10 workstations. The characteristics of the computers are shown in Table XXV. To obtain a realistic performance calibration, the same analysis was run on each computer individually. This was done on off-loaded machines with no other user processes running. All machines have enough memory so that no disk swapping occurred in any of the experiments. Table XXV shows the hostname, type, and performance measured by the standard benchmark suite SPECintbase92 [SPEC97]. In the fourth column is the total (i.e., including loading and parsing) CPU time for 1002 decisions¹ on c1908, followed by the total required memory and the total real time. In the seventh column, we show the performance in decisions per second, $R_i = \frac{W}{T_i}$, where $W=1002$ decisions, and T_i is the CPU time for machine i . We are using CPU time rather than real time for calibration, because we cannot guarantee that the system activity present during the calibration will be present during the actual experiments. Note the differences between the CPU and the real times in the

Host-name	Type	SPECint base92	CPUsec T_i [s]	Mem MB	Real Time	R_i	p_{SS10}
olga	SS10/41	53.2	1982	6.1	1981	0.51	1.00
biggar	SS10/ 41	53.2	1937	6.1	1969	0.52	1.02
jolliet	SS10/ 41	53.2	1996	6.1	1997	0.50	0.99
rond	SS10/ 41	53.2	2119	6.1	2135	0.47	0.94
carre	SS20/ 61	98.2	1302	6.1	1327	0.77	1.52
croche	SS5/85	65.3	1708	6.1	1734	0.59	1.16
tesecau	SS5/85	65.3	1728	6.1	1745	0.58	1.15
sawin	U1/140	215.0	842	6.1	855	1.19	2.35
anicet	SS4/110	68.8 ^a	1432	6.1	1459	0.70	1.38
garda	SS4/110	68.8 ^a	1434	6.1	1490	0.70	1.38

TABLE XXV: Calibration table c1908, 1002 decisions

a. Extrapolated from $SPECintbase95_{SS4/110}=1.37$ by multiplying by $SPECintbase92_{SS10/40}=50.2$

computer individually. This was done on off-loaded machines with no other user processes running. All machines have enough memory so that no disk swapping occurred in any of the experiments. Table XXV shows the hostname, type, and performance measured by the standard benchmark suite SPECintbase92 [SPEC97]. In the fourth column is the total (i.e., including loading and parsing) CPU time for 1002 decisions¹ on c1908, followed by the total required memory and the total real time. In the seventh column, we show the performance in decisions per second, $R_i = \frac{W}{T_i}$, where $W=1002$ decisions, and T_i is the CPU time for machine i . We are using CPU time rather than real time for calibration, because we cannot guarantee that the system activity present during the calibration will be present during the actual experiments. Note the differences between the CPU and the real times in the

1. Why 1002, not 1000? The number of decisions is controlled by the user, but each decision involves 4 classes (in case of 1-level decisions). If the user specifies N decisions, then the actual number of decisions is greater or equal to N and smaller than N+4.

table. The number of equivalent SparcStations10/41 $p_{SS10} = \frac{R_i}{R_{SS10}}$ is shown in the last column. The machine with the hostname *olga* was chosen as reference. The order of machines in the table is the order in which they were used for the experiments: 1 slave the first line, 2 slaves the first two lines, ..., 10 slaves all ten lines.

It is interesting to compare the performance for our task (p_{SS10}) with a standard benchmark suite (*SPECintbase92*). Table XXVI shows that an UltraSparc 1/140 is about 4-times faster than a SparcStation10 using

Hostname	Type	Performance factor $X = X_i / X_{olga}$	
		if X is <i>SPECint base92</i>	if X is p_{SS10}
olga	SS10/41	1.0	1.0
carre	SS20/ 61	1.8	1.5
croche	SS5/85	1.2	1.2
sawin	U1/140	4.0	2.4
anicet	SS4/110	1.3	1.4

TABLE XXVI: Comparison of SPECint92 with obtained performance

using SPECint92, but it was only 2.3-times faster for our case analysis. The SparcStation4/110 is the opposite.

Machine *rond* was also used as the master. This is possible without sacrificing accuracy, because the master does not require much CPU time for such a small number of slaves (see Figure 141). E.g., with 9 slaves, the master consumed 7 CPU seconds and the slave over 200 seconds, both on *rond*.

The results for the parallel case analysis are summarized in Table XXVII. N is the number of slaves, UB the calculated upper bound on switching activity, T is the real time to accomplish the work of $W=1002$ decisions. The number of equivalent SparcStations10 is in the third column.

The equivalent performance

$$ER(N) = \frac{NW}{PT}$$

is followed by the

equivalent speedup

$$ES(N) = \frac{ER(N)}{R_{SS10}}$$

and the effi-

$$ciency E = \frac{ES(N)}{N}$$

as derived in

Section 5-5; $R_{SS10}=0.5057$ from the

calibration in Table XXV. The cal-

culated upper bound on switching

activity in the second column is little worse for more machines than for one, because when many slaves start to explore the decision tree, there is not enough high-SAV paths to assign, hence a constant time (linear in the number of slaves) is wasted. Note that the real time includes about 5 to 10s of master and slave start-up overhead, i.e., the speedup would be better on longer runs.

N	UB [tr]	P	Real time T [s]	Eq. Performance ER(N)	Eq. Speedup ES(N)	Efficiency E
1	7992	1.00	1981	0.51	1.0	1.00
2	7992	2.02	1051	0.94	1.9	0.93
3	7997	3.02	694	1.4	2.8	0.95
4	7997	3.95	537	1.9	3.7	0.93
5	7997	5.47	418	2.2	4.3	0.87
6	7997	6.63	359	2.5	5.0	0.83
7	7997	7.78	284	3.2	6.3	0.90
8	7997	10.1	244	3.2	6.4	0.80
9	7997	11.5	214	3.7	7.2	0.80
10	7997	12.9	184	4.2	8.3	0.83

TABLE XXVII: Analysis results - circuit c1908, 1002 decisions, 1 to 10 computers in parallel

The upper bound on switching activity is plotted against the number of decisions in Figure 146. It shows that in the long run we obtain the same upper bound, but locally most of the time the parallel version lags behind the serial one. This is because more subspaces get

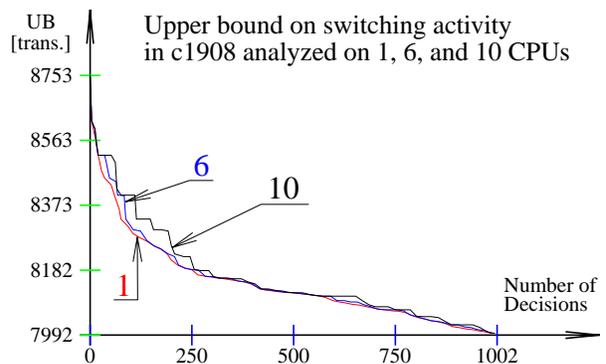


FIGURE 146: Upper bound as a functions of the number of decisions - c1908, 1002 decisions, 1, 6 and 10 CPUs

evaluated in parallel and the reported upper bound is the maximum over all subspaces including those currently assigned for processing. Therefore, with more slaves the curve is flatter for a long time and then suddenly drops when the slave processing the previous maximum returns its results to the master.

More interesting is to look at the same graph in real time (Figure 147), where we can see about $2x$ speedup for 2 slaves, and $8x$ speedup for 10 slaves (the exact data are in Table XXVII).

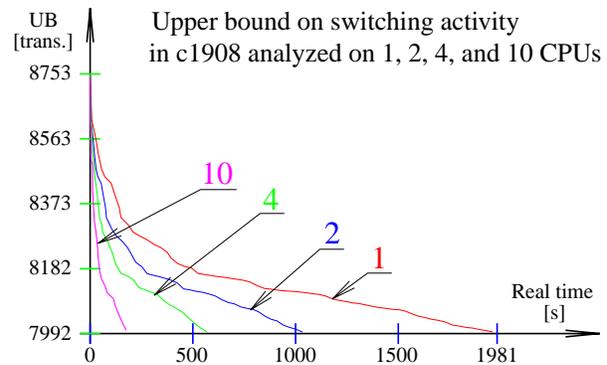


FIGURE 147: Upper bound in real time - circuit c1908, 1002 decisions, 1, 2, 4, 10 CPUs

In Figure 148, we show the equivalent system performance $ER(N)$ as a function of the number of computers working in parallel. It demonstrates that the parallel algorithm has close to linear speedup. The efficiency is about 95% for a small number (2, 3) of slaves and around 85% for a higher (8, 9, 10) number of slaves. It deteriorates with the number of slaves due to the communication overhead, the delay on the master, and the start-up delay. The behavior of the algorithm on a larger network is discussed in Section 5-5.3. In the next section we analyze a constant-time experiment on 1 to 10 machines.

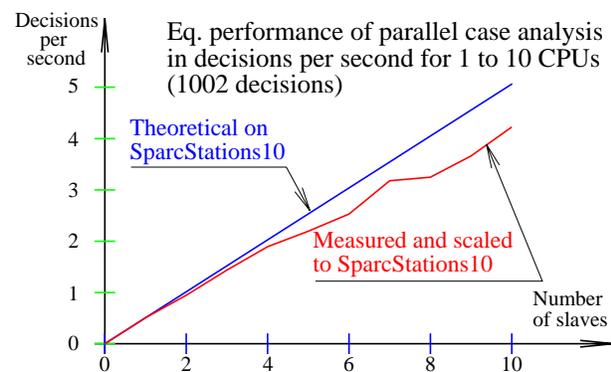


FIGURE 148: Performance of parallel case analysis - decisions per second, c1908, 1002 decisions, 1 to 10 CPUs

Note that the x-axis in Figure 148 is the number of slaves. It may not seem correct because the master is also a part of the parallel system. Recall the properties of the parallel case analysis described earlier in Section 5-1.2: the amount of the work done by the master is very small. In the experiment with 10 slaves the master process was taking far below 1 % of the CPU time on an unloaded machine. The error caused by measuring the CPU and real time is more than the error caused by neglecting the master. In other words, when we measured speedup on N machines

each being assigned one slave process, we obtained the same speedup regardless of whether the master process runs on the extra $(N+1)$ -th machine or on one of the slave machines.

The CPU time consumed by the master cannot be neglected when N is very large. However, in such case the speedup deteriorates too much due to network latency, context switch response on the master and the CPU time spent on the master (see Section 5-6). Then the speedup with N slaves is actually speedup with $N+1$ (highly loaded) machines, but since N is large then the speedups with N and $N+1$ slaves are approximately the same.

5-5.2 Analysis on a small number of computers for a constant time

We give here results of constant time performance analysis rather than constant work as in the previous section. We show the improvement in the upper bound on switching activity in the c1908 that can be obtained after 30 minutes of computation. The results are presented in Table XXVIII.

N	W [decisions]	UB [tr]	P	Eq. Performance $ER(N)$	Eq. Speedup $ES(N)$	Efficiency E
1	912	8014	1.00	0.51	1.0	1.0
2	1554	7875	2.02	0.86	1.7	0.85
3	2654	7795	3.02	1.5	2.9	0.97
4	3770	7742	3.95	2.1	4.2	1.0 ^a
5	4906	7688	5.47	2.5	4.9	0.99
6	5898	7639	6.63	3.0	5.9	0.98
7	7396	7575	7.78	3.7	7.3	1.0 ^a
8	8522	7539	10.1	3.8	7.4	0.92
9	9668	7519	11.5	4.2	8.3	0.92
10	11558	7493	12.9	5.0	9.8	0.98

TABLE XXVIII: Analysis results - circuit c1908, 30 minutes, 1 to 10 computers in parallel

- a. The calculated value of efficiency was a few percent above 1.00 due to numerical errors and inaccuracy in calibration

The upper bound is plotted against the number of slaves in Figure 149. It shows that 10 machines tighten the upper bound in 30 minutes by about twice the number of transitions compared to a single machine.

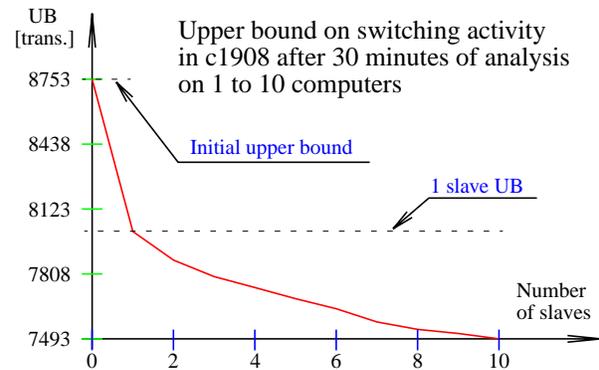


FIGURE 149: Upper bound as a function of the number of slaves - c1908, 30 minutes, 1 to 10 CPUs

The equivalent performance of the configurations of 1 to 10 machines is shown in Figure 150.

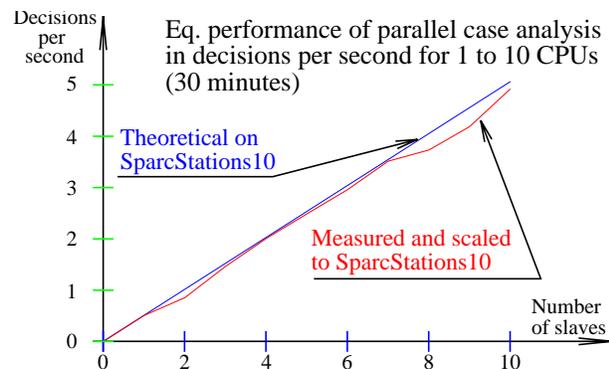


FIGURE 150: Performance of parallel case analysis as a function of the number of slaves - c1908, 30 minutes, 1 to 10 CPUs

5-5.3 Analysis on a large number of computers

We also tested the parallel algorithm on all machines in our department that could run the executable file, i.e., SunOS 4.X and 5.X. We found 87 such machines, ranging from up-scale servers down to some of the oldest SUN machines. To minimize the effect of network traffic and CPU load, this experiment was performed between 2am and 6am for 10 nights, 300000 decisions per night. The performance on the *c17* circuit was used for calibration (p_{ss10}). As in all calibrations, CPU time was used instead of real time for measuring T_i .

Checkpointing of the decision tree was done after 300000 decisions and the analysis terminated. The following night the analysis was restarted from the previous checkpoint. In total 3 million decisions were performed. The state saving and recovery took also a certain portion of the time, making the reported results conservative.

N	Switching activity		U/I %	P	Real time T [s]	Eq. Performance $ER(N)$	Eq. Speed up $ES(N)$	Efficiency E
	Initial [tr]	UB [tr]						
87	8753	6105	69.7	78.5	139461	24	47	0.54

TABLE XXIX: Analysis results - circuit c1908, $3 \cdot 10^6$ decisions, 87 computers in parallel

The results are summarized in Table XXIX. The values of $ER(N)$, $ES(N)$, E are calculated as in Section 5-5.1 but for $W=3 \cdot 10^6$, $N=87$, $R_{ssIO}=0.506$. The progress of the upper bound against the number of decisions is plotted in Figure 151. The analysis took

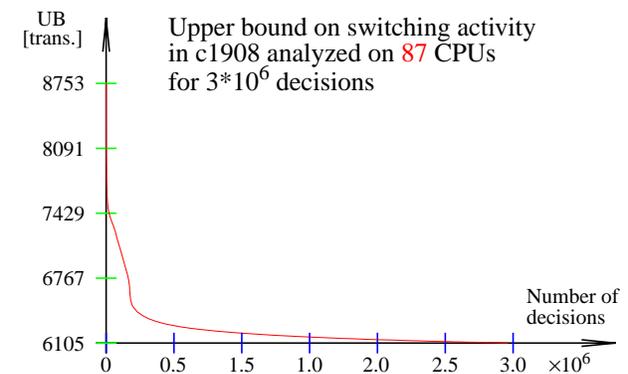


FIGURE 151: Upper bound as a function of the number of decisions - c1908, $3 \cdot 10^6$ decisions, 87 CPUs

139461 real-time seconds including checkpoint saving, parsing, loading, and the master and slave start-up times. **We achieved the parallel efficiency of 54.13% on 87 machines.** Note that we did not account for load by other user processes which makes the calculated speedup and efficiency conservative. The size of the checkpoint file containing the decision tree after $3 \cdot 10^6$ decisions was about 18 MB.

5-5.4 Depth of local exploration

The decision tree can be expanded by 1-level decision analysis at a time as shown so far, but also by 2 levels or more. We conducted an experiment with increments of depth of local exploration, running on 6 machines. We compared local expansion by 1 and

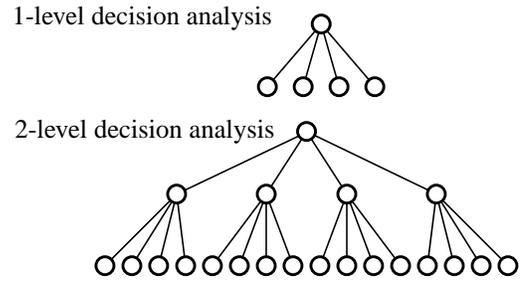


FIGURE 152: Depth of local decisions

2 levels. The 2-level analysis means up to 20 decisions, that is, a leaf is replaced by a sub-tree of up to 16 leaves as shown in Figure 152.

What effect can such modification have on the performance of the parallel algorithm? In the first place, it means sending longer messages over the network. But they are sent less frequently, because it takes longer to compute a

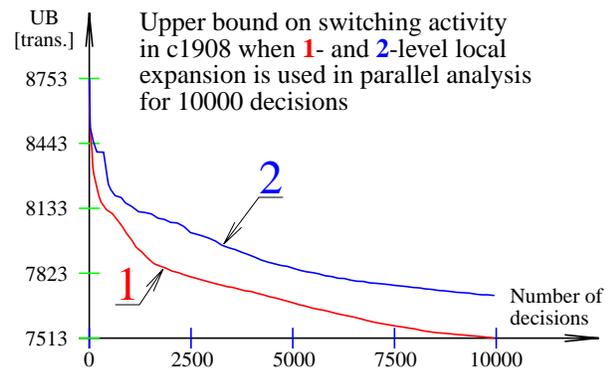


FIGURE 153: Comparison of 1- and 2-level local decision analysis - number of decisions, c1908, 10000 decisions, 6CPUs

larger subtree locally. Statistically, the network communication diminishes, because instead of sending l path per decision, we now send $16p/20d$, or $4^N / \left(\frac{4^{N+1} - 1}{4 - 1} - 1 \right)$ paths which for a very large number of levels N is 0.25 paths per decision only. However, many leaves with low SAVs get explored too, leading to wasted CPU time. From Figure 153 it seems that the 1-level decisions are more efficient.

Yet, each slave computes one 2-level decision analysis faster than (up to) five 1-level decision analyses. This is because the constraint system implementation is able to save the constraint system variables after each constraint assignment which speeds up the traversal of the decision subtree. Therefore, it is interesting to compare the 1- and 2-level local decision analyses in real time (Figure 154).

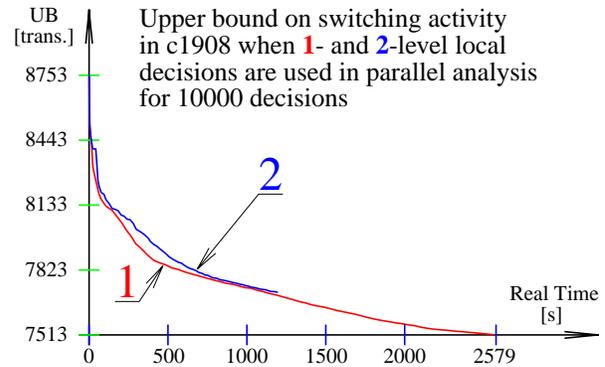


FIGURE 154: Comparison of 1- and 2-level local decision analysis - real time, c1908, 10000 decisions, 6 CPUs

The disadvantage of the 2-level local decision analysis of making useless decisions may actually disappear after making more decisions. This is because all subspaces with SAV higher than the resulting upper bound on the SAV must be explored; therefore, the probability that a

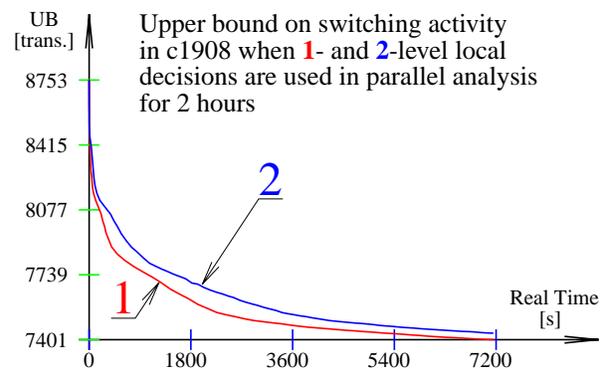


FIGURE 155: Comparison of 1- and 2-level local decision analysis - real time, c1908, 2 hours, 6 CPUs

subspace was uselessly explored diminishes with more decisions. It is impossible to predict which subspaces should be explored and which should not without actually knowing at least approximately the resulting upper bound on SAV. Purely hypothetically: if such an upper bound were known, each individual exploration would have stopped when the local maximum over the leaves drops below the upper bound. Therefore, we conducted the same experiment as the previous one, but the constant time of 2 hours was used rather than constant work load. The resulting upper bound plotted against real time (shown in Figure 155) indicates that after 2 hours the 1-level decision analysis still provides better results. This is because the resulting upper bound on SAV (which determines whether an explored

subspace was useful or wasted) decreases too slowly - logarithmically (\log_4) - with the number of decisions.

5-6. Theoretical model

5-6.1 Properties necessary for the application of our parallel algorithm

The parallel case analysis algorithm and the implementation can solve a typical large-space search problem - it uses the “divide and conquer” strategy. However, it performs the “divide” part dynamically. It means that the search space (or set of jobs) is not assigned statically but the subspaces are assigned dynamically to available slaves, eliminating the search in subspaces which are not needed and balancing load on slaves. The only restriction which the algorithm imposes on the type of the problems is that the individual subspaces must be independent. The algorithm can be modified to handle certain dependence as long as it allows at least some parallelism, but the performance would diminish. Network speed, and granularity of the parallelism influence the performance, and this is studied in the next section.

5-6.2 Speedup

Consider an execution trace as shown in Figure 156. Let t_{m1} be the time to prepare a job on the master, t_{m2} the time to process the results of one job on the master, t_{c1} the time to communicate a job to a slave, t_{c2} to communicate the result to the master, and t_s the average (over all slaves and jobs) time to do a job on a slave. To complete one job takes on the average time t_s (units of time) on a single machine. On a parallel machine it is

$$t_J = t_{m1} + t_{m2} + \frac{t_{c1} + t_s + t_{c2}}{N} + t_{confl},$$

assuming that all N machines have about the same performance and there are communication conflicts costing t_{confl} . Denoting $t_m = t_{m1} + t_{m2}$, $t_c = t_{c1} + t_{c2}$, we

get $t_J = t_m + \frac{t_c + t_s}{N} + t_{confl}$. Assuming a conflict of 2 messages occurring with probability P_2 and costing t_m , the lost time $t_{confl} = t_m P_2$. In a general case for all

conflicts of 2 to N messages the lost time is $t_{confl} = \sum_{i=2}^N P_2^{(i-1)} (i-1) t_m =$

$\sum_{i=1}^{N-1} i P_2^i t_m$, assuming a conflict of k messages occurs with the probability

$P_k = P_2^{k-1}$ and costs $(k-1)t_m$ units of time.

To compute the probability P_2 of a conflict of two messages we assume that there will be a penalty of up to t_m units of time when a message arrives while the master is busy (for t_m) with another message or within a t_m -long window before that.

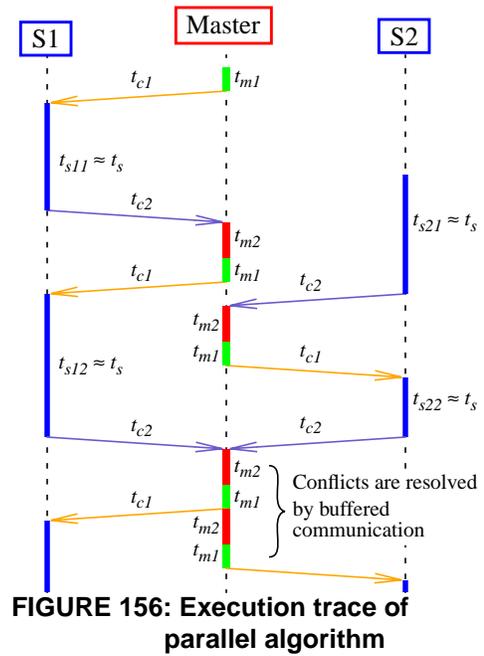


FIGURE 156: Execution trace of parallel algorithm

Therefore, during J jobs the conflict period is $2Jt_m$ of the total time Jt_j . The probability of a conflict of two messages arriving at the same time is then

$$P_2 = \frac{2Jt_m}{Jt_j} = \frac{2t_m}{t_j}. \text{ The lost time is thus } t_{confl} = t_m \sum_{i=1}^{N-1} iP_2^i = t_m P, \text{ where } P =$$

$$\sum_{i=1}^{N-1} iP_2^i = \sum_{i=1}^{N-1} i \left(\frac{2t_m}{t_j} \right)^i. \text{ The time for one job on a slave is then}$$

$t_j = t_m + \frac{t_c + t_s}{N} + t_m P$. Solving the last two equations for t_j means solving a polynomial equation of degree $N+2$ (see Appendix B) which the software [Mupa97] used for symbolic calculation could not handle. Therefore, we assume only conflicts of 2 messages.

When only conflicts of 2 messages can occur then $P = P_2 = \frac{2t_m}{t_j}$. Then we

express t_j as a function of N and finally the speedup¹ becomes $S(N) = \frac{t_s}{t_j}$,

$$\text{(Eq. 2) } S(N) = \frac{2Nt_s}{t_c + t_s + Nt_m + \sqrt{2t_c t_s + 2Nt_c t_m + 2Nt_m t_s + t_c^2 + t_s^2 + 9N^2 t_m^2}}$$

The speedup is $S(N)=N$ if there is no communication nor master overhead, $t_c=0$, $t_m=0$.

If the time t_c for communication is comparable with the time t_s for processing 1 job

$$\text{on a slave, } t_c=t_s, \text{ the speedup is } S(N) = \frac{2Nt_s}{2t_s + Nt_m + \sqrt{4Nt_m t_s + 4t_s^2 + 9N^2 t_m^2}}.$$

1. Since this is for a homogeneous parallel system, or an equivalent homogeneous parallel system, $S(N) = ES(N)$

When the number N of slaves is just large enough for the master to be completely busy with handling messages and scheduling jobs ($2Nt_m \approx t_s$) and with $t_c=0$, the

speedup approaches $\frac{4}{3 + \sqrt{17}}N \approx 0.562 N$.

With a very large number of slaves the speedup then approaches the value

$$\lim_{N \rightarrow \infty} S(N) = \frac{t_s}{2t_m}.$$

We also calculated the speedup $S(N)|_{3msg}$ when considering conflicts of up to three messages. It gives a long expression the limit of which for large number N of

slaves is $\lim_{N \rightarrow \infty} S(N)|_{3msg} \approx 0.3613 \frac{t_s}{t_m}$.

5-6.3 Conditions for the best performance

The results from the previous section can be summarized in the following conditions necessary for the algorithm to be the most efficient (with the speedup close to the number of slaves N): $t_s \gg t_m$ and $t_s \gg t_c$. That is, the time to process a job on a slave is much longer than the time spent on one job on the master and the time for communication. That is essentially a basic property of any parallel system, so is no surprise.

5-6.4 Theoretical analysis of our practical results on 87 machines

In this section we give an example on how to apply the formula for speedup (EQ 2) to estimate the performance before actually implementing the algorithm. Let us try to estimate the speedup for our experiments as performed on the largest configuration of 87 machines. We had the following parameters:

- $N=78.54$, assuming a homogenous network of SparcStations10 (Table XXIX).
- $t_s=5.93s$ derived from the performance of 0.506 decisions per second on the

reference machine assuming 3 decisions per job on average (Table XXVII).

- $t_c=2\text{ms}$ consisting of 0.2ms necessary for the transfer of 200B over a 10BaseT network multiplied by 10 to account for latency on drivers.
- $t_m=23\text{ms}$ based on measured 7 CPU seconds for 1002 decisions (Section 5-5.1) of which we assume 6s were used for initialization and 1s for the job scheduling and message processing. This means 3ms for 334 3-decision jobs. We add estimated 20ms for context switching¹.

The theoretical speedup (EQ 2) is $ES(78.54) = 54.76$, which corresponds to the parallel efficiency of $E = \frac{ES(N)}{N} = 69.7\%$. We account the difference (about 14%) in our measured equivalent speedup $ES(78.54)=47.1$

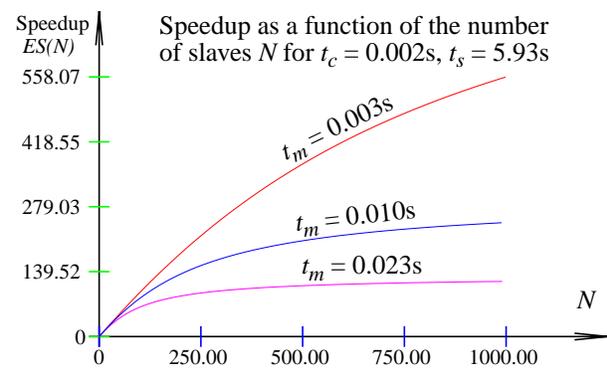


FIGURE 157: Calculated speedup

and the theoretical one to the other load on the machines and the possibly imprecise estimate of t_m , i.e., the time the master needs to process the messages and to send jobs. Note that t_m influences speedup very strongly; therefore, the exact measurement of the average time lost due to context switching is important. Without context switching ($t_m=3\text{ms}$ only), the speedup $ES(N)$ would be 75.29 instead of 54.76. The calculated speedup is plotted against the number of slaves in Figure 157. It shows that the speedup is almost linear for less than 50 slaves no

1. More precisely, it is the real time delay from the moment a data is received until the master process is scheduled to run. Our estimate is $\frac{1}{2}$ of the time quantum for which the second highest priority process is allowed to run on a multitasking operating system. We used the command `dispadmin -c TS -g` which reported 20ms for the highest priority, 40 ms for the next 18 priorities, ..., and 200ms for the lowest priority level.

matter what the value of t_m is (from 3ms to 23ms). It also shows that the speedup would be about 560 with an off-loaded master and 1000 slaves.

5-7. Parallel implementation - conclusions

We developed a **parallel algorithm that exploits the independence** of tasks performed during the case analysis. The algorithm is capable of rapidly synchronizing the jobs whenever the independence disappears. The C++ implementation over TCP on 10BaseT Ethernet achieved the **parallel efficiency** of **83%** on **10** computers and an unloaded network (Table XXVII on page 166). On a heterogeneous network of **87** computers we achieved the **parallel efficiency** of **54%** (Table XXIX on page 170). In terms of the upper bound on switching activity the parallel case analysis lowered the **upper bound for c1908 by 14%** of the initial upper bound on **10** machines compared to **8% on 1 machine** during 30 minutes of real time.

We also provided a theoretical model of the used parallel heterogeneous system. The experimental speedup was within 14% of the speedup calculated using the theoretical model. The accuracy can be further improved by measuring network delays rather than estimating them.

Our parallel control algorithm **is not limited** to our switching activity method. For any type of a problem that fits the general characteristics described in Section 5-6.1 on page 173, it offers a simple way of parallel evaluation with practically proven **high parallel efficiency** on a local network of inexpensive computers.

CHAPTER 6

Voltage Drops in Power Busses

In this chapter, we describe an enhancement to our switching activity verification method to obtain switching activity data necessary for estimation of voltage drops in power busses. The algorithm to obtain a switching activity profile is explained in Section 6-1, followed by an example in Section 6-2, and a description of the C++ implementation in Section 6-3. Experimental results obtained on the MCNC and ISCAS85 benchmarks are summarized in Section 6-4.

6-1. Switching Activity Profile Algorithm

The problem of malfunction due to voltage drops in power busses is explained in Section 2-3.7.1. Voltage drops can be analyzed by constructing the resistance network of the power bus and calculating the current drawn by connected gates or blocks. The resistance network can be extracted from layout or approximated from net length data which is computed in the same way that synthesis tools estimate net parasitics [BrOt98]. The current is computed from switching activity (Section 2-1.9). However, so far we computed only the worst power consumption in a clock period. This is not sufficient for power bus voltage drop analysis. The highest voltage drop will occur when the highest current is drawn (called *peak current*)

[KrNH95]. Therefore, peak switching activity at any time is needed. *Current profile* is the current drawn from the power lines expressed as a function of time. *Switching activity profile* is switching activity expressed as a function of time. *Peak current* is the maximum current in the current profile. *Peak switching activity* (denoted further *PSAV*) is the maximum switching activity in the switching activity profile.

The most generic model is to compute switching activity profile over the time interval we are interested in (many clock periods, at worst the entire uptime of the system) and take the maximum. Similarly as in the case of peak power consumption described in the previous chapters, we will focus on the worst case clock period (no temporal correlation). Temporal correlation or other operating conditions can be added in a form of additional constraints on the pseudo-primary inputs of our model.

The algorithm for computing voltage drops can be implemented as follows. First the circuit is partitioned in such way that each partition has a single contact point to the power bus (1 power and 1 ground point) and the resistance networks leading to the contact points are derived. The size of the partitions is used to control the accuracy and can range from individual cells to functional blocks. Then the switching activity for each partition is computed, and from the switching activity profile the current profile is computed. The maximum voltage drop is then determined from the peak current in the current profile.

Since the constraint system approach to power verification is very open and flexible, only a minor modification of the algorithm was needed and all the advanced functions (e.g., case analysis, parallel analysis) were preserved. Only the function counting transitions during the clock period was changed to capture the transition activity profile - the number of transitions in one interval of discrete time. The maximum over all intervals of discrete time is then the *peak switching activity*.

We focus on computation of the switching activity. For more accurate analysis RCL parasitics rather than a simple resistance network should be extracted and the dynamic effect of current in power busses should be considered. Furthermore, current profile should be created directly by mapping individual transitions to current rather than building the switching activity profile first.

6-2. Example of Switching Activity Profile Calculation

The calculation of a switching activity profile is illustrated on circuit *c17* (Figure 158) from the ISCAS85 benchmark circuits [BrgF85, Brya89].

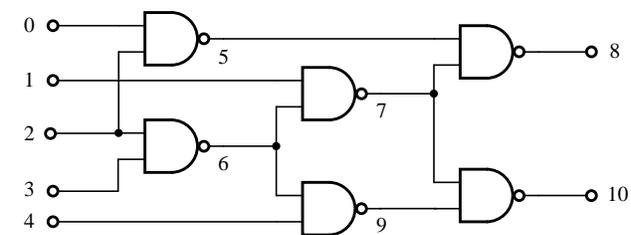


FIGURE 158: Circuit *c17*; all gates have unit

Primary inputs of the circuit are assigned the two-vector waveform sets and these are then propagated through the network. The resulting abstract waveforms on all nodes of the circuit are shown in Figure 159.

The number of transitions during each interval of discrete time excluding primary inputs is: zero transitions during the *0-th* time interval (or also called “at time 0” of the discrete time), four transitions at 10 and 20, and 2

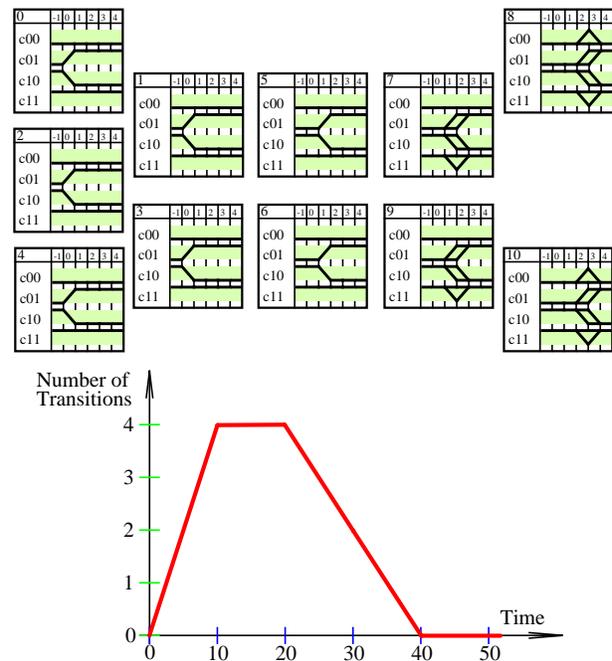


FIGURE 159: Waveforms and the current profile of *c17* after initial forward

transition at time 30. The maximum number of transitions during any time interval

In the next section we describe the implementation of the switching activity profile computation.

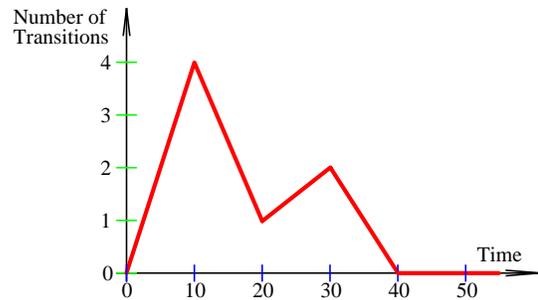


FIGURE 162: Switching activity profile c17

6-3. Implementation of Switching Activity Profile Analysis

The implementation within the existing power verification framework is quite straightforward. One array of integers per partition¹ is added - one integer per one interval of discrete time. For each interval of discrete time the number of transitions is computed as follows: For each gate output in the partition the maximum number of rising and falling transitions within the time interval over all waveform classes is found. The sum of such maxima over all gate outputs is the value of the switching activity profile at the given interval of discrete time. The array of integers is then updated in each step of the case analysis in order that the up-to-date switching activity profile be available after each decision.

Unfortunately, no simple caching of the profile is possible because a change on a single net can change the profile. Caching would require storing the previous abstract waveform and comparing it with the current one to determine which time intervals have changed. This is more costly than computing the number of transitions in each time interval, and it needs much more memory.

To test the algorithm on more realistic circuits we decided to map the ISCAS85 circuits onto a widely-accepted commercial ASIC library. This step actually took much more effort than to implement the calculation of the switching activity pro-

1. In our experiments a single partition (the entire circuit) was used.

file. The lsi_10k library from LSI Logic, Inc., was selected. The ISCAS'85 circuits were mapped from Boolean Verilog primitives onto the lsi_10k using a Perl script. The gates that were not available in lsi_10k were replaced by their structural equivalents. The design was then analyzed using the static timing analyzer PrimeTime from Synopsys, Inc. Typical operating conditions and the default wireload model were used. Net delays were included in the cell delays, and the cell delays were saved in SDF format. The pin-to-pin rise/fall min-max delays were compressed into a single min-max interval per gate output using a Perl script. The resulting intervals were expressed in 200ps intervals and rounded to the nearest integer (down for min, up for max). Note that the current implementation of our constraint system switching activity method supports only fixed gate delays therefore whenever the min-max gate delay interval is longer than one interval of discrete time then the average of min and max is internally considered as the gate delay. The results obtained on the lsi_10k-mapped ISCAS'85 circuits are summarized in the next section.

6-4. Experimental results

The calculation of the peak switching activity and the switching activity profiles was tested on two sets of circuits - ISCAS'85 circuits mapped onto the lsi_10k library and the MCNC benchmark circuits. The MCNC circuits [LoSy89] with a small number of primary inputs were used to test how fast the case analysis can reach the exact solution. In all cases the HPCA case analysis was used, and the heuristic FANOUT (in the case of MCNC circuits also PIFAN) with no additional advanced options (learning, reconvergence analysis, ...) was deployed. We selected the number of subsequent decisions during which the case analysis made no progress as the stopping criterion, i.e., it reached a region in which performing more decisions would not influence the final result. The results are presented for 10, 100, 1000, and 10^6 no-progress decisions.

6-4.1 Peak switching activity in ISCAS'85 circuits

Table XXX

shows the topological information about the ISCAS'85 circuits mapped onto the lsi_10k library.

Name	Topology					
	PI	PO	Gates	Nets	Fan	Topological Delay
c17	5	2	6	11	2	8
c432	36	7	167	203	9	107
c499	41	32	210	251	12	67
c880	60	26	383	443	8	95
c1355	41	32	554	595	12	99
c1908	33	25	899	932	25	164
c2670	233	64	1202	1435	28	162
c3540	50	22	1671	1721	22	206
c5315	178	123	2330	2508	31	179
c6288	32	32	2416	2448	16	627
c7552	207	107	3568	3775	72	164

TABLE XXX: Topological data for ISCAS85_Isi10k benchmarks

Table XXXI

shows both the lower bound and the initial peak power, and the peak current. Note that the following notation is used in the tables in this chapter: peak power is

Name	Initial			Lower Bound		
	Delay	Power (SAV)	Peak Current (PSAV)	Delay	Power (SAV)	Peak Current (PSAV)
c17	5	10	4	5	9	4
c432	79	2545	83	79	858	61
c499	58	1602	82	58	403	56
c880	63	4868	179	63	999	67
c1355	69	10426	258	69	1148	128
c1908	114	19236	340	104	3068	83
c2670	122	13225	324	117	3450	117
c3540	151	38823	650	141	9044	160
c5315	121	35738	835	120	7675	248
c6288	217	68062	2054	203	26265	913
c7552	110	60789	1437	109	10539	314

TABLE XXXI: Initial, and simulation switching activity for ISCAS85_Isi10k benchmarks

expressed as the maximum number of transitions during one clock period over all clock periods (SAV), peak current is expressed as the maximum number of transitions during one interval of discrete time over all intervals and all clock periods (PSAV). The lower bounds were obtained by simulation for a number of vectors ranging from 50 thousand (on larger circuits) to 2 million (on smaller circuits).

Table XXXII summarizes the peak switching activity value (PSAV) obtained from the switching activity profiles for the limit of 10 decisions of case analysis with no progress. The CPU time is in seconds, memory size in MB, loading and parsing is excluded. Suffix “E” means that the value of PSAV is exact.

Name	Init	LB	UB	U/I%	U/L%	CPU	MEM
c17	4	4	4E	100	100	0	0.04
c432	83	61	83	100	136	17	0.16
c499	82	56	82	100	146	3	0.04
c880	179	67	173	97	258	17	0.23
c1355	258	128	258	100	202	9	0.05
c1908	340	83	324	95	390	176	0.07
c2670	324	117	322	99	275	99	0.76
c3540	650	160	593	91	371	2859	0.95
c5315	835	248	795	95	321	37	0.04
c6288	2054	913	2054	100	225	35	0.04
c7552	1437	314	1403	98	447	285	2.21

TABLE XXXII: PSAV (Peak Current) in ISCAS85_Isi10k circuits with no_progress=10, heuristic FANOUT

Table XXXIII summarizes the peak switching activity value (PSAV) obtained from the switching activity profiles for the limit of 100 steps of case analysis with no progress, and Table XXXIV show that same data for 1000 steps.

Name	Init	LB	UB	U/I%	U/L%	CPU	MEM
c17	4	4	4E	100	100	0	0.04
c432	83	61	83	100	136	95	0.20
c499	82	56	82	100	1146	34	0.06
c880	179	67	173	97	258	146	0.23
c1355	258	128	258	100	202	83	0.06
c1908	340	83	321	94	387	517	0.11
c2670	324	117	314	97	268	1662	0.78
c3540	650	160	519	80	324	12959	1.18
c5315	835	248	783	94	316	654	0.05
c6288	2054	913	2054	100	225	944	2.46
c7552	1437	314	1403	98	447	1737	2.21

TABLE XXXIII: PSAV (Peak Current) in ISCAS85_Isi10k circuits with no_progress=100, heuristic FANOUT

Name	Init	LB	UB	U/I%	U/L%	CPU	MEM
c17	4	4	4E	100	100	0	0.04
c432	83	61	83	100	136	884	0.24
c499	82	56	82	100	146	358	0.09
c880	179	67	173	97	258	1372	0.27
c1355	258	128	258	100	202	828	0.09
c1908	340	83	317	93	382	8540	0.25
c2670	324	117	308	95	263	15274	0.82
c3540	650	160	511	79	319	52295	1.40
c5315	835	248	778	93	314	4040	0.13
c6288	2054	913	2054	100	225	11158	3.07
c7552	1437	314	1403	98	447	15987	2.25

TABLE XXXIV: PSAV (Peak Current) in ISCAS85_Isi10k circuits with no_progress=1000, heuristic FANOUT

To compare the heuristics FANOUT and PIFAN, Table XXXV summarizes the peak switching activity value (PSAV) obtained from the switching activity profiles for the limit of 1000 steps of case analysis with no progress using heuristic PIFAN.

Name	Init	LB	UB	U/I%	U/L%	CPU	MEM
c17	4	4	4E	100	100	0	0.04
c432	83	61	82	99	134	1790	0.30
c499	82	56	82	100	146	1019	0.18
c880	179	67	173	97	258	1150	0.25
c1355	258	128	256	99	200	3172	0.22
c1908	340	83	293	86	353	34970	0.62
c2670	324	117	322	99	275	4268	0.86
c3540	650	160	520	80	325	54708	1.17
c5315	835	248	778	93	314	4058	0.14
c6288	2054	913	2054	100	225	11321	3.07
c7552	1437	314	1350	94	430	334214	2.64

TABLE XXXV: PSAV (Peak Current) in ISCAS85_Isi10k circuits with no_progress=1000, heuristic PIFAN

From Tables XXXIV and XXXV, it is clear that there is only a marginal difference between the heuristics FANOUT and PIFAN. A considerable difference is only in c1908, where PIFAN obtained a slightly lower upper bound but needed about 4x more CPU time. The c2670 has shown the opposite effect: FANOUT gives a tighter upper bound but needed about 3x more CPU time. This is due to spatial correlation of signals - c1908 has several large fanouts on primary inputs; therefore, PIFAN needs fewer decisions than FANOUT.

A comparison with other methods is not straightforward. To the author’s knowledge, there are no published methods that would compute the switching activity profile and peak switching activity. The closest is [KrNH95], see Section 2-3.5.1 on page 77. But even this method computed peak current rather than switching activity, and uses an unpublished delay mapping based on fanout while we used a standard industrial library. Therefore, the value of such a comparison is very limited. Table XXXVI compares the improvement by case analysis and the upper to lower bound ratios in both methods. Keep in mind that the data in [KrNH95] represent the peak current, while ours is the peak switching activity. The first data column is from [KrNH95] and shows the I/L ratio, i.e., the ratio of their initial upper bound, obtained with the *iMAX10* algorithm, to their lower bound L , produced with up to 100,000 patterns using their *iLogSim* simulator. The column labeled U/L shows the ratio of their final upper bound, obtained with their *PIE* algorithm to the lower bound L . The following four columns compare the two methods. The values of PSAV used for comparison are those from Table XXXIV.

Circuit	[KrNH95]		Improvement by case analysis		U/L	
	I/L	U/L	[KrNH95]%	Our %	[KrNH95]%	Our %
c432	1.26	1.19	5.56	0	119	136
c499	1.95	1.82	6.67	0	182	146
c880	1.72	1.63	5.23	3.4	163	258
c1355	1.33	1.33	0	0	133	202
c1908	1.22	1.14	6.56	6.8	114	382
c2670	1.38	1.36	1.45	4.9	136	263
c3540	2.53	1.70	32.8	21.4	170	319
c5315	1.71	1.57	8.19	6.8	157	314
c6288	1.39	1.39	0	0	139	225
c7552	1.54	1.44	6.50	2.4	144	447

TABLE XXXVI: Comparison of our results with [KrNH95]; PSAV using FANOUT, no_progress=1000

If the values were really comparable then [KrNH95] achieved better U/L bound ratio in all cases but c499. On top of all the differences mentioned earlier, the lower bounds were obtained differently, while the exact value is not known. Therefore, in the next section we compare our method with the exhaustive simulation on a set of circuits with a small number of primary inputs.

6-4.2 Peak switching activity in the MCNC circuits

Several MCNC benchmark circuits with a small number of inputs (≤ 16) were run to compare our upper bound on peak switching activity with the exact value, to test how the case analysis can

Name	Topology					
	PI	PO	Gates	Nets	Fanout	Topological Delay
alu2	10	6	354	364	23	9470
alu4	14	8	722	736	22	9510
cm138a	6	8	19	25	8	1040
cm162a	14	5	45	59	6	1690
cm163a	16	5	43	59	5	1640
cm85a	11	3	31	42	3	1470
cmb	16	4	42	58	5	1350
cu	14	11	48	62	8	1430
f51m	8	8	129	137	15	1840
parity	16	1	15	31	1	1220
pm1	16	13	39	55	8	1030
t481	16	1	1945	1961	239	8230
x2	10	7	44	54	10	1080

TABLE XXXVII: Topological data for MCNC benchmarks

reach the exact solution, and to find out when it cannot. The topological properties of the selected MCNC benchmark circuits are shown in Table XXXVII.

The delay, the switching activity, and the peak current after the initial forward propagation, and the lower bounds after a simulation for 1024 test vectors are shown in Table XXXVIII.

Name	Initial			Lower Bound		
	Delay	Power (SAV)	Peak Current (PSAV)	Delay	Power (SAV)	Peak Current (PSAV)
alu2	6215	14045	160	5338	959	26
alu4	6242	30207	371	5728	2240	50
cm138a	770	52	6	770	25	6
cm162a	1233	206	12	1233	71	8
cm163a	1209	136	10	1209	77	10
cm85a	1084	107	12	1084	45	8
cmb	990	144	16	990	57	16
cu	1054	162	8	938	56	7
f51m	1307	610	30	1307	189	24
parity	909	15	8	909	10	7
pm1	763	83	8	763	46	7
t481	6069	11392	769	6037	975	225
x2	804	159	7	804	65	7

TABLE XXXVIII: Initial and simulation switching activity for MCNC benchmarks

In the following several tables we summarize the obtained upper bounds on PSAV in the MCNC circuits. We used the heuristic FANOUT limited by the number of decisions with no progress in the upper bound: 10 (Table XXXIX), 100 (Table XL), 1000 (Table XLI), and 10^6 (Table XLII).

Name	Init	LB	UB	U/I%	U/L%	CPU	MEM
alu2	160	26	62	39	238	827	0.34
alu4	371	50	210	57	420	881	0.71
cm138a	6	6	6E	100	100	1	0.04
cm162a	12	8	11	92	138	2	0.07
cm163a	10	10	10	100	100	1	0.06
cm85a	12	8	11	92	138	2	0.08
cmb	16	16	16	100	100	2	0.06
cu	8	7	7	88	100	2	0.04
f51m	30	24	24E	80	100	20	0.14
parity	8	7	8	100	114	0	0.04
pm1	8	7	8	100	114	2	0.06
t481	769	225	322	42	143	2232	0.13
x2	7	7	7	100	100	2	0.06

TABLE XXXIX: PSAV (Peak Current) in MCNC circuits with no_progress=10

Name	Init	LB	UB	U/I%	U/L%	CPU	MEM
alu2	160	26	37	23	142	3942	0.37
alu4	371	50	123	33	246	25276	0.82
cm138a	6	6	6E	100	100	1	0.04
cm162a	12	8	10	83	125	11	0.08
cm163a	10	10	10E	100	100	2	0.06
cm85a	12	8	10E	83	125	4	0.08
cmb	16	16	16E	100	100	2	0.06
cu	8	7	7E	88	100	2	0.04
f51m	30	24	24E	80	100	20	0.14
parity	8	7	8E	100	114	0	0.04
pm1	8	7	8E	100	114	2	0.06
t481	769	225	235	31	104	29870	0.27
x2	7	7	7E	100	100	2	0.06

TABLE XL: PSAV (Peak Current) in MCNC circuits with no_progress=100

Name	Init	LB	UB	U/I%	U/L%	CPU	MEM
alu2	160	26	28E	18	108	10203	0.33
alu4	371	50	86	23	172	121213	0.73
cm138a	6	6	6E	100	100	1	0.04
cm162a	12	8	9E	75	112	25	0.09
cm163a	10	10	10E	100	100	2	0.06
cm85a	12	8	10E	83	125	4	0.08
cmb	16	16	16E	100	100	2	0.06
cu	8	7	7E	88	100	2	0.04
f51m	30	24	24E	80	100	20	0.14
parity	8	7	8E	100	114	0	0.04
pm1	8	7	8E	100	114	2	0.06
t481	769	225	225E	29	100	40667	0.17
x2	7	7	7E	100	100	2	0.06

TABLE XLI: PSAV (Peak Current) in mcnc circuits with no_progress=1000

Name	Init	LB	UB	U/I%	U/L%	CPU	MEM
alu2	160	26	28E	18	108	10268	0.33
alu4	371	50	57E	15	114	765846	0.73
cm138a	6	6	6E	100	100	1	0.04
cm162a	12	8	9E	75	112	25	0.09
cm163a	10	10	10E	100	100	2	0.06
cm85a	12	8	10E	83	125	4	0.08
cmb	16	16	16E	100	100	2	0.06
cu	8	7	7E	88	100	2	0.04
f51m	30	24	24E	80	100	20	0.14
parity	8	7	8E	100	114	0	0.04
pm1	8	7	8E	100	114	2	0.06
t481	769	225	225E	29	100	40450	0.17
x2	7	7	7E	100	100	2	0.06

TABLE XLII: PSAV (Peak Current) in mcnc circuits with no_progress=10⁶

Table XLIII shows PSAV in the MCNC circuits using heuristic PIFAN (case analysis on primary inputs). Since the circuits have a very small number of primary inputs, the heuristic PIFAN guarantees to find the exact PSAV in a more predictable number of decisions.

Name	Init	LB	UB	U/I%	U/L%	CPU	MEM
alu2	160	26	28E	16	108	6741	0.43
alu4	371	50	57E	13	114	354176	0.71
cm138a	6	6	6E	100	100	0.58	0.04
cm162a	12	8	8E	67	100	160	0.12
cm163a	10	10	10E	100	100	2	0.05
cm85a	12	8	10E	83	125	4	0.08
cmb	16	16	16E	100	100	2	0.06
cu	8	7	7E	88	100	2	0.04
f51m	30	24	24E	80	100	18	0.15
parity	8	7	8E	100	114	0	0.04
pm1	8	7	8E	100	114	2	0.06
t481	769	225	225E	29	100	40717	0.17
x2	7	7	7E	100	100	2	0.06

TABLE XLIII: PSAV (Peak Current) in MCNC circuits with no_progress=10⁶, heuristic PIFAN

In all cases the method with heuristic FANOUT reached the exact solution. In all but 3 circuits it was under 25 CPU seconds (Intel Pentium Pro 150MHz; loading, parsing excluded). The longest CPU time was needed for circuit alu4 - almost 9 CPU days. This is ascribed to the data-path nature of the circuit. Switching activity on the busses of signals carrying numerical (read arbitrary) values is not eliminated by spatial correlation enforced by the constraint system. However, note that the ratio of the exact to the initial value is very low - 15%, indicating that many transitions in the initial estimate were not real. The method lowers the initial upper bound of 371 to 210 in about 880 CPU seconds. Since this is a very small circuit, the exact solution would be obtained sooner by exhaustive simulation in the case of alu4. But for many other circuits our method runs in fractions of second. And even in the case of the 3 circuits with long runtimes it provides a significant advantage: it provides an upper bound at any time, i.e., not completing the process due to time restrictions or system failure does not invalidate all the effort.

The heuristic PIFAN (Table XLIII) resulted in the same upper bound for each circuit as with FANOUT (i.e., the exact value of PSAV in both cases). However, in the case of ALU4, it was achieved in about half of the CPU time, in the case of ALU2 it was 30% less, while all the other circuits needed about the same time.

This is because the inverse gate projections of the constraint system are less efficient than the positive (forward) ones. Since the analysis down to the exact values requires a case analysis on all nets with more than one class, making a decision on an internal net does not propagate to the primary inputs in some cases and therefore more decisions may be needed with the heuristic FANOUT. Thus the heuristic PIFAN provides a better predictability of CPU time than FANOUT when searching for the exact values.

CHAPTER 7

Conclusions

We presented in this thesis a new pattern-independent method based on constraint resolution for computing an **upper bound on switching activity** in the combinational part of a **sequential synchronous circuit**. The method is inspired by constraint satisfaction techniques based on relational representation of operators on discrete domains, and on interval narrowing. The circuit and the sets of possible electrical waveforms on its nets are represented by a **constraint system**. The variables of the constraint system are the sets of electrical waveforms which can occur on nets under given operating conditions. The constraints represent the circuit function and timing (gates or blocks), and the operating conditions. The verification problem is translated into a constraint satisfaction problem as follows: The circuit netlist, the library (gate function and delay) are translated into a constraint system. The operating environment (signals on primary inputs) are translated into additional constraints. Verification is done by tightening the constraint system. The sets of all possible waveforms are computed as the greatest fixpoint of the constraint system in **time $O(n)$** where n is the number of nets in the circuit. This gives (a pessimistic) **upper bound on the switching activity** during any single clock period as well as a pessimistic upper bound on the peak switching activity on each net at any time.

7-1. Constraint system

The method is **efficient** because the sets of all possible waveforms are not enumerated but rather compacted into a structure called an *abstract waveform*. An abstract waveform has all **waveforms classified into 4 waveform classes** according to the initial and final stable values of each waveform. The merger of waveforms within one class is the reason for pessimism, i.e., why the method returns an upper bound rather than the exact solution.

7-2. Case analysis

The notion of waveform classes carrying **independent sets of waveforms allows for efficient case analysis** for determining a tighter upper bound on the circuit switching activity. The case analysis is scalable, i.e., with more resources (CPU time and memory) the upper bound can be tightened down to the exact solution. The case analysis uses efficient pruning but nonetheless **the worst-case complexity** in obtaining the exact solution is comparable to simulation, $O(4^{n_{pi}})$, where n_{pi} is the number of primary inputs. Keep in mind, that unlike simulation, which approaches the exact solution from the bottom up (gives a lower bound) this method approaches it from the top down (gives an **upper bound**). Therefore, it is a **static switching activity method** with a very nice option of going down to the exact solution when desirable (generally for a small critical block).

7-2.1 Initial upper bound on switching activity

We conducted several experiments on the ISCAS'85 benchmark circuits. The **initial upper bound** on the switching activity was about **2.8 times of the lower bound** obtained by simulation (Table V on page 127). While computing the initial upper bound took only **few seconds** (Table IV on page 126), the simulations which ran for almost **one year** (Section 4-1 on page 124) were nowhere near exhausting the space of input vectors.

7-2.2 Case analysis algorithms

Several heuristics and search algorithms were developed. The *case analysis search* algorithms efficiently explore the decision tree during the search for a tight upper bound on switching activity. The case analysis heuristics guide the case analysis search. The case analysis algorithms are independent of the problem of the switching activity verification and thus can be reused, including the C++ implementation for any other search problem with similar characteristics. The experiments show that **the most efficient** case analysis algorithm is the *HPCA* (Section 4-7 on page 139).

7-2.3 Heuristics for net selection

The case analysis finds the **exact solution** in time $O(n_{pi})$ in the **best case** and $O(4^{n_{pi}})$ in the **worst case**. The time needed to find the exact solution is always between these two extremes and depends on the order of the decisions. We developed and tested several heuristics for the selection of nets: nets with the largest fanout (Fanout), primary inputs with the largest fanout (PIs), and closing nets of reconvergent regions. Also, the influence of global learning on Boolean values was tested. The heuristics are compared in Table XIV on page 138. There is no single winner among the heuristics. For small circuits, the best is **Fanout with learning**. For c1908 the best heuristic is **PIs with learning**. For the 2 *largest circuits* (c6288, c7552) the selection of nets on **Closing nets with learning** provided the best results. This is because none of the heuristics is better than all others for any type of circuit, but their performance depends on the topology and the function of the circuits. E.g., circuit c1908 is a controller with several inputs that propagate decisions deeply into the circuit; circuit c6288 is a multiplier with many reconvergent regions.

7-2.4 Parallel case analysis

While studying and practically testing the case analysis algorithms, we realized that many tasks are independent during certain time periods during the case analy-

sis. We developed a **parallel algorithm which exploits this independence**, capable of rapidly synchronizing the jobs whenever the independence disappears. The C++ implementation over TCP on 10BaseT Ethernet achieved the **parallel efficiency** of **83%** on **10** computers and an unloaded network (Table XXVII on page 166). On a heterogeneous network of **87** computers we achieved the **parallel efficiency** of **54%** (Table XXIX on page 170). The parallel case analysis lowered the **upper bound on switching activity for c1908** by **14%** of the initial upper bound on **10** machines, compared to **8% on 1 machine** during 30 minutes of real time in each case.

Our parallel control algorithm **is not limited** to our switching activity method. For any problem that fits the general characteristics described in Section 5-6.1 on page 173, it offers a simple way of parallel evaluation with practically proven **high parallel efficiency** on a local network of inexpensive computers. It can be used, e.g., with the method described in [KrNH95]. The implementation also offers a fail-safe mechanism protecting against failure of any computers including the master, dynamic resizing of the number of active slaves, execution during off-peak hours of the day, and continuing with no loss on the next day.

7-3. Comparison with other methods

The presented method is a static verification method. Therefore, all the characteristics of static methods are true compared to the pattern-dependent methods as described in the literature survey. We compared our method with another static methods in Section 4-10 on page 144, and Section 6-4.1 on page 185. Unfortunately, a direct comparison of the results with an existing method was not possible. The closest method [KrNH95] still has many differences in both the method and the testing strategy.

We perform case analysis on internal nodes, which is further enhanced by backward propagation of waveform constraints using partial inverse functions. In [KrNH95], the case analysis on internal nodes would be quite difficult, because the waveform representation used **does not allow decomposition of waveforms into independent subsets** (like our classes), and the circuit is not described by a set of constraints which allows improved detection of inconsistent assignments. Case analysis on internal nodes has proven to be useful on some circuits, but there are circuits where case analysis on PIs is better (Table XIV). Case analysis on PIs is better if the stems of large reconvergent regions with many false transitions are located closer to PIs.

The work reported in [KrNH95] **provides** a conversion of switching activity into **electric current**, which could be adapted to our solution, though its value is questionable unless a real industrial library is used. Our concept of discrete time is both an advantage and a disadvantage. Our method is able to get **a rough estimate very quickly** by reducing the number of intervals, but it does not remove transitions eliminated by gate's inertial delay when the interval length is smaller than the gate delay.

Our initial estimate is obtained in several tens of seconds and further improved in steps (1 decision), each taking a similar amount of time. This makes the method **scalable**.

We compared the upper bounds on peak switching activity obtained with our method with the lower bounds obtained by exhaustive simulation in Section 6-4.2 on page 189. In all cases our method computed the exact value (i.e., a value that is both an upper and a lower bound) quickly. An exception was, e.g., the circuit *alu4* due to the data-path nature of the circuit.

7-4. Contributions of this thesis

The major contributions of this thesis are summarized below. For a detailed list of contributions and the author's statement of originality please refer to Section 1-3.3 on page 24.

1. We investigated the use of a constraint resolution method for computing an upper bound on switching activity in the combinational part of a digital synchronous circuit.
2. We developed the necessary theory about constructing the constraint system and formulating the switching activity estimation problem.
3. We developed, implemented, and tested the algorithms for fixpoint calculation, case analysis, net selection for case analysis, and parallel case analysis.
4. We investigated how global learning, reconvergent analysis, and several other enhancements can speed up and/or improve this switching activity estimation method.

7-5. Future research

This thesis has never had as its primary goal to develop a fully functional commercial static power verification method. The primary goal was to investigate the use of a constraint resolution method for static switching activity verification. That **has been accomplished**.

There are **many other features** which are needed for a successful commercial EDA tool. For the last several months of writing this thesis the author has been working in a group developing a static timing verification tool with the fastest growing share of the market of the EDA sign-off tools (PrimeTime from Synopsys, Inc.). Even from this short experience, one can see that there are many subtle (from the research point of view) features which are a must for customers. These include,

e.g., delay calculation support, multi-cycle paths, full latch support, various clocking schemes, derived clocks, user-disabled timing arcs, strong library support, on-chip delay variation, incremental approach to handle multi million gate designs, etc. The possible directions of future research outlined in the next sections cover only a few of them.

7-5.1 Circuit models

Our implementation of the switching activity verification method is limited to fixed gate delay and does not assume gate delay correlation nor temporal correlation of input patterns. This is one possible direction of future research. Another one would be to account for inertial delay by eliminating more false transitions.

7-5.2 Order of decisions during case analysis

Among the heuristics for net selection none has proven to be the best for all the tested circuits. Finding a better heuristic is another possible direction of future research.

7-5.3 Conversion of switching activity into electric current

The present implementation computes an upper bound on switching activity. A fairly simple mapping layer can be added for reading a design library and the conversion of the computed switching activity into electric current.

7-5.4 Uses of parallel case analysis

The parallel case analysis can be used for other purposes, not limited to verification of digital circuits or even the electrical engineering domain.

Index

A

activity waveform	68
activity waveforms	67
analysis	
power	
high-level	68
And	85
and operation	85
and operation over abstract waveforms	89
asynchronous	19
ATPG	48, 50
automated test pattern generation	48
average power	8, 20, 30

B

battery life	8
BDD	62
best-first	49
binary decision diagram	62
BIST	7
boolean difference	63
bounded delay	31
bug	xxxv

C

C++	9
c1908	35, 51, 53, 54, 55
capacitance coefficient	69
capacitance feed-through	76
capacitive load	39
CDFG	73
cell	
characterization	13
cells	
library cells	11
Chapman-Kolmogorov	65

circuit	
asynchronous	19
combinational	28, 32
synchronous	19
circuit simulation	95
circuits	
digital	
synchronous	5
clock period	29
CMOS	5, 27
combinational circuit	28, 32
delay	28
combinational part of a synchronous sequential circuit	19
Complement	85
complement	85
component correlation	36, 54
control flow graph	73
controlability	115
correctness	
functional	8
correlation	
spacial	64
spatial	35
spatio-temporal	65
temporal	36, 64
current	
class-A	38
crowbarring	38
dynamic	39
leakage	39
peak	78, 179
short-circuit	38
current waveform	30

D

δ _leaves	52
data flow graph	73
D-calculus	
timed	51

delay	
bounded	31
circuit	
exact	33
maximal	42
clock to Q	28
falling	32
fixed	31
interval	31
pin-to-pin	32
representation	31
rising	32
topological	32
two-vector	33
delay model	
correlated	37
floating-mode	33
non-correlated	37
sequence of vectors	33
transition	33
delay operation over abstract waveform	90
delay-on-inputs	32
depth-first	49
D-error value	51
design for testability	7
design rule check	5
design rules	5
design-space exploration	4
devices	
portable	8
DFT	7
DIFFERENCE - BDD operation	63
difference between the viability and floating-mode criteria	45
digital synchronous circuits	27
don't-care optimization	71
DRC	5
dual bit type	69
dynamic logic	13

E

ECD, Expected Current Distribution	80
EDA	xxxv
ENF	50
equilibrium probability	63
equivalence checking	16

equivalent normal form	50
error	
by neglecting sequential behaviour	64
exhaustive simulation	40
advantages	41
disadvantages	41
exist line	105
exit line	107
exit net	113
Expected Current Distributions	80

F

factoring of boolean expressions	72
false path	35
false path problem	34
feature-line width	4
FF	19
fixed delay	31
flip-flops	19
Floating-mode delay model	33
forest of hierarchy of regions	113
formal verification	15, 16
foundry	4
functional correctness	8

G

gate	32
2-input CMOS NAND	76
CMOS	38
CMOS inverter	76
reconvergence	104
closing	104
primary	106
secondary	106
gate delay model	32
gated clocks	73
gate-level	
annotated	5
glitch	38
zero-with glitches	58
glitching sensitivity	62
global implications	115
glue logic	11
graph covering algorithms	72

H

hazard	
dynamic	38
static	38
hold time	19, 28

I

implication	
one design implies another	7
implications	
global	115
independency	
spacial	61
temporal	61
instantaneous power	30
interconnection delay	12
interval delay	31
inverse and operation over abstract waveform	90
ISCAS-85	53

L

layout	4
least square regression	74
library modules	
gates of various sizes	71
line	
exit	105, 107
logic	
dynamic	13
static	13
lower bound sensitization	46

M

Markov FSM	65
maximum voltage drop	78
method	
power	
based on constraint resolution	78

methods	
path-oriented	41
power	
handling interval gate	65
pattern independent	77
probabilistic	61
statistical	60
power verification	60
minimum clock period	57
model	
gate delay	19
model checking	16
model fitting technique	70
modelling	28
monotone speedup property	44
multi-function unit	70

N

near-closed set	79
net	
entry	113
exit	113
NMOS transistor model	77
non-polynomial complete	22
non-polynomial hard	22
NPC	22
NPH	22

O

observability	115
operations over space of abstract waveforms	87
optimization	
power	
arithmetic unit	73
Boolean network	71
circuit-level	70
DSP processor	74
encoding	72
high-level	73
logic-level	71
system and software level	74
or	85
or operation	85

P

partial input enumeration	78
path	34
longest statically sensitizable	42
sensitizable	41
path balancing	72
peak current	179
peak power	8, 20, 31
peak switching activity	180
performance and cost specification	3
PI,	28
PIE	78
place&route	4
power	
average	8, 20
dynamic	22
gate model	39
Instantaneous	30
peak	8, 20, 31
savings by switching off	73
toggle	38, 61
total of a CMOS gate	39
power average	30
power bus	79
power consumption estimation	
at gate level	12
power supply voltage	69
PPO	28
primary inputs	28
probability waveform	63
PSAV - peak switching activity value	180
pseudo-primary outputs	28

R

real waveform	84, 85
Realization	2
reconvergence	104
reconvergent stem region	
region	
reconvergent	104
region	
reconvergent	107
overlap	105

regions	
reconvergent	
hierarchy	113
retiming	73
RTL	4

S

sensitizable under floating mode criterion	44
sensitizable under lower bound criterion	46
sensitization	
comparison	47
dynamic	43
floating-mode	44
lower bound	46
static	42
viability	44
vigorous	47
sensitization criteria	41
Sequence of vectors	33
setup time	19, 28
Shannon's expansion theorem	62
side inputs	34
signal probability	61
simulation	15
Monte Carlo	60
random	60
slack	33, 71
space of real waveforms	84
spatial independence	63
Specification	2
specification	
behavioral	3
gate-level	4
validation	3
SPECint	8
Spice	75
state explosion	16
state transition graph for power estimation	76
static CMOS logic	13
stem	
reconvergent	104
STGPE	76

switching activity	20, 30
conversion to power	38
intrinsic	74
on buses	73
peak	180
profile	180
simultaneous	62
symbolic probability	62
synchronous	19
synchronous sequential circuit	
properties	28
synthesis	
behavioral	4, 74
logic	4
synthesis for low power	74

T

TBF, timed boolean functions	53
technology mapping	12
temporal independence	63
terminal	23
Test	2
testability	7
theorem proving	17
time	
setup	19
time to market	3
timed D-calculus	50
timed test generation	50, 51
timing constraint	17
timing verification	18
toggle power	38
tome	
hold	19
topological delay	32
total power	69
transistor sizing	71
transition delay	33
transition density	62, 63, 66
transition probability	61
transition space	84
Two-vector delay	33

U

uncertainty waveform	77
union of transition sets	87
union of waveform classes	87
upper bound	96

V

validation	3
value	
controlling	34
non-controlling	34
verificatio	
power	
low-level approaches	75
verification	6
at behavioral level	9
at gate level	11
at RTL	10
formal	15, 16
functional	6
methods	
static	16
power	
high-level approaches	68
power consumption	
at behavioral level	20
at RTL	20
state of the art	21
timing	8, 18
by simulation	18
formal	18
verification by simulation	15
VHD	9
viable	44
VLSI	2
voltage drops	20, 79

W

W, space of real waveforms	84
waveform class union	87

References

- [AMR93] **Ashar, P., Malik, S., Rothweiler, S.**, *Functional timing analysis using ATPG*, EDAC93, pp. 506-511, 1993.
- [AouC97] **Aourid, S. M., Cerny, E.**, *CLP-Based Gate Level Timing Verification with Delay Correlation*, IEEE/ACM International Workshop on Logic Synthesis, Granlibakken, 1997.
- [AyKaxx] **Ayari, B., Kaminska, B.**, *Hierarchical algebraic ATPG based on BDD representation and on new observability concept*
- [BMMP97] **Benini, L., De Micheli, G., Macii, E., Poncino, M., Scarsi, R.**, *Fast Power Estimation for Deterministic Input Streams*, ICCAD97, pp. 494-501, November 1997.
- [BHMS94] **Bahar, R. I., Hachtel, G. D., Macii, E., Somenzi, F.**, *A symbolic method to reduce power consumption of circuits containing false paths*, ICCAD94, pp. 368-371, 1994.
- [BNYT92] **Burch, R., Najm, F., Yang, P., Trick, T.**, *McPOWER: A Monte Carlo approach to power estimation*, ICCAD92, pp. 90-97, 1992.
- [BNYT93] **Burch, R., Najm, F., Yang, P., Trick, T.**, *A Monte Carlo approach for power estimation*, IEEE Transactions on VLSI Systems, Vol. 1, No. 1, March 1993, pp. 63-71, 1993.
- [Bouc93] **Boucouris, S.**, *IC In Core structures documentation*, Nortel, 1993.
- [BrIy88] **Brand, D., Iyengar, V.**, *Timing analysis using functional analysis*, IEEE Trans. Comp., Oct. 1988.

- [BrgF85] **Brglez, F., Fujiwara, H.**, *A neural netlist of 10 combinational benchmark circuits and a target translator in fortran*, International Symposium on Circuits and Systems, pp. 663-698, June 1985.
- [Broph97] **Brophy, D. R.**, *HDL Independent Advanced Delay and Power Calculation Standard*, 2nd Workshop on Libraries, Component Modeling, and Quality Assurance, Spain, 1997, pp. 287-296.
- [BrOt98] **Brayton R., Otten R.**, *Planning for performance*, DAC98, pp. 122-127, June 1998.
- [BrRS87] **Brayton R., Rudell R., Sangiovanni-Vincentelli A., Wan A.**, *A multiple-level logic optimization system*, IEEE Transactions on CAD of Integrated Circuits, Vol. 6, pp. 1062-1081, November 1987.
- [Brya89] **Bryan, D.**, *The ISCAS '85 benchmark circuits and netlist format*, MCNC, http://www.cbl.ncsu.edu/pub/Benchmark_dirs/ISCAS85/DOCUMENTATION/iscas85.ps, October 1989.
- [Bryb87] **Bryant, R. E.**, *A Survey of Switch-level Algorithms*, IEEE Design & Test, vol. 4, no. 4, pp. 26-40, August 1997.
- [Bur95] **Burkett, D.**, *Intel's Response to the Pentium Bug Crisis*, electronic publication, Berkeley, 1995, <http://haas.berkeley.edu/~burkett/intell1.html>
- [CCDL94] **Cheng, S. W., Chen, H.-C., Du, D. H. C., Lim, A.**, *The role of long and short paths in circuit performance optimization*, IEEE Transactions on CAD of Integrated Circuits and Systems, Vol. 13, No. 7, pp. 857-864, July 1994.

- [CerK95] **Cerny, E., Khordoc, K.**, *Interface Specification with Conjunctive Timing Constraints: Realizability and Compatibility*, AMAST Workshop on Real-Time Systems, Bordeaux, pp. 11, June 1995
- [CerZ94] **Cerny, E. , Zejda, J.**, *Gate-level timing verification using waveform narrowing*, EuroDAC, Grenoble, pp. 374-379, September 1994.
- [CGSS97] **Carloni, L.P., McGeer, P.C., Saldanha, A., Sangiovanni-Vincentelli, A.L.**, *Trace Driven Logic Synthesis - Application to Power Minimization*, ICCAD97, pp. 581-589, November 1997.
- [Chan94] **Chan, N.-H.**, *Rapid current analysis for CMOS digital circuits*, MEng thesis, McGill University, July 1994.
- [ChA93] **Chang, H, Abraham, J. A.**, *VIPER: An efficient vigorously sensitizable path extractor*, 30th ACM/IEEE Design Automation Conference (DAC93), pp. 112-117, 1993.
- [ChaS94] **Chang, R. C.-H., Sheu, B. J.**, *An analog MOS model for circuit simulation and benchmark test results*, 1994, pp. 311-314.
- [ChDu91] **Chen, H.-C., Du, D. H. C.**, *Path sensitization in critical path problem*, ICCAD, pp. 208-211, 1991.
- [ChoS95] **Choudry, U., Sangiovanni-Vincentelli, A.**, *Automatic generation of analytical models for interconnect capacitances*, IEEE Transactions on Computer-Aided Design, vol. 14, pp. 470-480, 1995.
- [ChouR96] **Chou, T.-L., Roy, K.**, *Accurate Power Estimation of CMOS Sequential Circuits*, IEEE Transactions on VLSI Systems, Vol. 4, No. 3, September 1996.

- [Chre95] **Chren, W. A.**, *Low delay-power product CMOS design using one-hot residue coding*, ISLPD95, pp. 145-150, 1995.
- [ChRC97] **Chen, Z., Roy, K., Chou, T.-L.**, *Power Sensitivity - A New Method to Estimate Power Dissipation Considering Uncertain Specifications of Primary Inputs*, ICCAD97, pp. 40-44, November 1997.
- [ChRP94] **Chou, T.-L., Roy, K., Prasad, S.**, *Estimation of circuit activity considering signal correlations and simultaneous switching*, ICCAD94, pp. 300-303, 1994.
- [CipR96] **Ciplickas, D. J., Rohrer, R. A.**, *Expected Current Distributions for CMOS Circuits*, ICCAD96, pp. 589-592, 1996.
- [Ciri87] **Cirit, M. A.**, *Estimating dynamic power consumption of CMOS circuits*, ICCAD87, pp. 534-537, 1987.
- [CoRa88] **Cox, H., Rajski, J.**, *A method of fault analysis for test generation and fault diagnosis*, IEEE Transactions on CAD, Vol. 7, No. 7, pp. 813-833, July 1988.
- [CoRa94] **Cox, H., Rajski, J.**, *On necessary and nonconflicting assignments in algorithmic test pattern generation*, IEEE Transactions on CAD of Integrated Circuits and Systems, Vol. 13, No. 4, pp. 515-530, April 1994.
- [CPMR95] **Chandrakasan, A., Potkonjak, M., Mehra, R., Rabaey, J., Brodersen, R.**, *Optimizing power using transformations*, IEEE Transactions on CAD, Vol. 14, No. 1, pp. 12-31, January 1995.
- [DaSu97] **Dai, W., Sun, W.**, *3-D Parasitic Extraction for Deep Submicron IC Design*, Integrated System Design Archive, <http://www.isdmag.com/Editorial/1997/ASIC9707.html>, July, 1997.

- [DeKM93] **Devadas, S., Keutzer, K., Malik, S.**, *Computation of floating mode delay in combinational circuits: Theory and algorithms*, Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 12, No. 12, December 1993.
- [DevM95] **Devadas, S., Malik, S.**, *A survey of optimization techniques targeting low power VLSI circuits*, 32nd ACM/IEEE Design Automation Conference (DAC95), pp. 242-247, 1995.
- [DKMW94a] **Devadas, S., Keutzer, K., Malik, S., Wang, A.**, *Certified timing verification and the transition delay of a logic circuit: practice and implementation*, IEEE Transactions on VLSI Systems, Vol.2, No. 3, pp. 333-342, September 1994.
- [DKMW94b] **Devadas, S., Keutzer, K., Malik, S., Wang, A.**, *Event suppression: improving the efficiency of timing simulation for synchronous digital circuits*, IEEE Transactions on Computer-Aided Design, Volume 13, Number 6, pp. 814-822, 1994.
- [DMQP94] **F. Dartu, N. Menezes, J. Quian, L. T. Pillage.** *A gate-delay model for high-speed CMOS circuits*, 31st ACM/IEEE Design Automation Conference (DAC94), pp. 576-580, 1994.
- [DuNR94] **Dutta, S., Nag, S., Roy, K.**, *ASAP: A transistor sizing tool for speed, area, and power optimization of static CMOS circuits*, International Symposium on Circuits and Systems, pp.61-64, 1994.
- [GDKW92] **Ghosh, A., Devadas, S., Kreutzer, K., White, J.**, *Estimation of average switching activity in combinational sequential circuits*, 29th ACM/IEEE Design Automation Conference (DAC92), pp. 253-259, 1992.

- [GoMa83] **Goncalves, N., De Man, H. J.**, *NORA: A Race-free dynamic CMOS Technique for Pipelined Logic Structures*, IEEE J. Solid-State Circuits, vol. SC-22, pp. 899-901, June 1983.
- [GoMe93] **Gordon, M., Melham, T.**, *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*, Cambridge University Press, 1993.
- [GoOC94] **Goodby, L., Orailoglu, A., Chau, P.**, *Microarchitectural synthesis of performance-constrained, low-power VLSI designs*, ICCD94, pp. 323-326, 1994.
- [GSSB91] **McGeer, P., Saldanha, A., Stephan, P. R., Brayton, R. K., Sangiovanni-Vincentelli, A. L.**, *Timing analysis and delay-fault test generation using path-recursive functions*, ICCAD91, pp. 180-183, 1991.
- [Haye86] **Hayes, John, P.**: *Digital Simulation with Multiple Logic Values*, IEEE Transactions on CAD of ICS, pp. 274-283, April 1986
- [HsRP97] **Hsiao, M.S., Rudnick, E.M., Patel, J.H.**, *Effects of Delay Models on Peak Power Estimation of VLSI Sequential Circuits*, ICCAD97, pp. 45-51, November 1997.
- [Hutt97] **Hutt, J.**, *Hot Topic: Microprocessors: Where will be the major and CAD challenges for the next architectural wave?*, presentation of Joe Hutt, IBM, European Design & Test Conference (EDAC), Paris, March 1996.
- [Hwan93] **Hwang, K.**, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, New York, McGraw-Hill, Inc., 1993, Chapter 3: "Principles of Scalable Performance."

- [ImaP94] **Iman, S., Pedram, M.**, *Multi-level network optimization for low power*, ICCAD94, pp. 372-377, 1994.
- [JyMa94] **Jyu, H., Malik, S.**, *Statistical delay modeling in logic design and synthesis*, 31st ACM/IEEE Design Automation Conference (DAC94), pp. 126-130, 1994.
- [Kame93] **Kamel, T.**, *Design and implementation of a static timing analyzer using CLP(BNR)*, internal report of Computing Research Lab, Bell-Northern Research, 1993.
- [Kapo94] **Kapoor, B.**, *Improving the accuracy of circuit activity measurement*, 31st ACM/IEEE Design Automation Conference (DAC94), pp. 734-739, 1994.
- [KKLV94] **Kondratyev, A., Kishinevsky, M., Lin, B., Vanbekbergen, P.**, *Basic gate implementation of speed-independent circuits*, 31st ACM/IEEE Design Automation Conference (DAC94), pp. 56-62, 1994.
- [KKRV95] **Kumar, N., Katkoori, S., Rader, L., Vemuri, R.**, *Profile-driven behavioral synthesis for low-power VLSI systems*, IEEE Design & Test of Computers, pp. 70-84, Fall 1995.
- [KNYH93] **Kriplani, H., Najm, F., Yang, P., Hajj, I.**, *Resolving signal correlations for estimating maximum currents in CMOS combinational circuits*, 30th ACM/IEEE Design Automation Conference, pp. 384-388, 1993.
- [KoNa97] **Kozhaya, J.N., Najm, F.N.**, *Accurate Power Estimation for Large Sequential Circuits*, ICCAD97, pp. 488-493, November 1997.

- [KrNH92] **Kriplani, H., Najm, F., Hajj, I.**, *Maximum current estimation in CMOS circuits*, 29th ACM/IEEE Design Automation Conference (DAC92), pp. 2-7, 1992.
- [KrNH94] **Kriplani, H., Najm, F., Hajj, I.**, *Improved delay and current models for estimating maximum currents in CMOS VLSI circuits*, ISCAS94, pp. 1.435-1.438, 1994.
- [KrNH95] **Kriplani, H., Najm, F., Hajj, I.**, *Pattern independent maximum current estimation in power and ground buses of CMOS VLSI circuits: Algorithms, signal correlations, and their resolution*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 14, No. 8, pp. 998-1012, August 1995.
- [Krod91] **Krodel, T. H.**, *PowerPlay - fast dynamic power estimation based on logic simulation*, ICCD91, pp. 96-100, 1991.
- [KuPr93] **Kunz, W., Pradhan, D. K.**, *Accelerated dynamic learning for test pattern generation on combinational circuits*, IEEE Transactions on CAD of ICS, vol. 12, no. 5, pp. 694-694, May 1993.
- [LaB94] **Lam, W. K. C., Brayton, R. K.**, *Timed boolean functions - a unified formalism for exact timing analysis*, Kluwer Academic Publishers, ISBN 0-7923-9454-2, 1994.
- [LanR95] **Landman, P. E., Rabaey, J. M.**, *Architectural power analysis: The dual bit type method*, IEEE Transactions on VLSI Systems, Vol. 3, No. 2, pp. 173-187, June 1995.
- [LBSV93] **Lam, W. K. C., Brayton, R. K., Sangiovanni-Vincentelli, A. L.**, *Circuit delay models and their exact computation using timed boolean functions*, 30th ACM/IEEE Design Automation Conference (DAC93), pp. 128-134, 1993.

- [LBSV94] **Lam, W. K. C., Brayton, R. K., Sangiovanni-Vincentelli, A. L.**, *Exact minimum cycle times for finite state machines*, 31st ACM/IEEE Design Automation Conference (DAC94), pp. 100-105, 1994.
- [LemS94] **Lemons, C., Shetti, S. S. M.**, *A low power 16 by 16 multiplier using transition reduction circuitry*, IWLPD94, pp. 139-142, 1994.
- [LeRS83] **Leiserson, C. E., Rose, F. M., Saxe, J. B.**, *Optimizing Synchronous circuitry by retiming*, 3rd CalTech Conference on VLSI, pp. 23-36, March 1983.
- [LeTa79] **Lengauer, T., Tarjan, R. E.**, *A fast algorithm for finding dominators in a flowgraph*, ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, pp. 121-141, July 1979.
- [Lhom93] **Lhomme, O.**, *Consistency techniques for numeric CSPs*, 13th IJCAI Conf., 1993.
- [LiLS94] **Lin, J.-Y., Liu, T.-C., Shen, W.-Z.**, *A cell-based power estimation in CMOS combinational circuits*, ICCAD94, pp.304-309, 1994.
- [LLop94] **Llopis, R. P.**, *Exact path sensitization in timing analysis*, European Design Automation Conference (EuroDAC94), 1994.
- [LLXB94] **Llopis, R. P., Xirgo, L. R., Bordoll, J. C.**, IEEE, pp. 160-163, 1994.
- [LoSy89] *logic synthesis '89 benchmark information*, MCNC, http://www.cbl.ncsu.edu/CBL_Docs/lgs89.html, 1998
- [MaaR87] **Maamari, F., Rajski, J.**, *Reconvergent fanout analysis and fault simulation complexity of combinational circuits*, IEE Electronics Letters, vol. 23, no. 21, pp. 1131-1133, Oct. 1987.

- [MaaR88a] **Maamari, F., Rajski, J.**, *A reconvergent fanout analysis for efficient exact fault simulation of combinational circuits*, 18th Fault Tolerant Computation Symposium, pp. 122-127, June 1988.
- [MaaR88b] **Maamari, F., Rajski, J.**, *A fault simulation method based on stem regions*, ICCAD-88, pp. 170-173, Nov. 1988.
- [MaaR90] **Maamari, F.**, *On the structural properties of combinational circuits and their application to fault simulation*, Ph.D. thesis, Department of Electrical engineering, McGill University, Montreal, Canada, 1990.
- [MaaR90a] **Maamari, F., Rajski, J.**, *A method of fault simulation based on stem regions*, IEEE Transactions on CAD, Vol. 9, No. 2, pp. 212-220, February 1990.
- [MaaR90b] **Maamari, F., Rajski, J.**, *The tulip fault simulation program*, user manual, Dept. of EE, McGill University, Montreal, Canada, July 1990.
- [MaaR93] **Maamari, F., Rajski, J.**, *The dynamic reduction of fault simulation*, IEEE Transactions on CAD of Integrated Circuits and Systems, Vol. 12, No. 1, pp. 137-148, January 1993.
- [MaFu92] **Matsunaga, Y., Fujita, M.**, *A fast test pattern generation for large scale circuits*, SASIMI92, Icoke, Japan, pp. 263-271, April 1992.
- [MaMP94] **Marculescu, R., Marculescu, D., Pedram, M.**, *Switching activity analysis considering spatiotemporal correlations*, ICCAD94, pp. 294-299, 1994.
- [MBOI95] **Mehta, H., Borah, M., Owens, R. M., Irwin, M. J.**, *Accurate estimation of combinational circuit activity*, 32nd ACM/IEEE Design Automation Conference (DAC95), 1995.

- [McGB91] **McGeer, P. C., Brayton, R. K.**, *Integrating functional and temporal domains in logic design*, Kluwer Academic Publishers, 1991.
- [MDGK97] **Monteiro J., Devadas S., Ghosh A., Keutzer K., White J.**, *Estimation of Average Switching Activity in Combinational Logic Circuits Using Symbolic Simulation*, IEEE Transactions on CAD of ICs and Systems, Vol. 16, No. 1, January 1997
- [MGBr89] **McGeer, P. C., Brayton, R. K.**, *Efficient algorithms for computing the longest viable path in a combinational network*, 26th ACM/IEEE Design Automation Conference, 1989.
- [MoDG93] **Monteiro, J., Devadas, S., Ghosh, A.**, *Retiming sequential circuits for low power*, ICCAD93, pp. 398-402, 1993.
- [MonD95] **Monteiro, J., Devadas, S.**, *Techniques for the power estimation of sequential logic circuits under user-specified input sequences and programs*, ISLPD95, pp. 33-38, 1995.
- [MoDL94] **Monteiro, J., Devadas, S., Lin, B.**, *A methodology for efficient estimation of switching activity in sequential logic circuits*, 31st ACM/IEEE Design Automation Conference (DAC94), pp. 12-17, 1994.
- [Mupa97] *MuPAD, Multi Processing Algebra Data Tool*, <http://www-math.uni-paderborn.de/MuPAD/>, 1997.
- [NabR92] **Nabavi-Lishi, A., Rumin, N. C.**, *Delay and bus current evaluation in CMOS logic circuits*, ICCAD92, pp. 198-203, 1992.
- [NabR94] **Nabavi-Lishi, A., Rumin, N. C.**, *Inverter-based models for current analysis of CMOS Logic Circuits*, 1994, pp. 13-16

- [Najm91] **Najm, F. N.**, *Transition density, a stochastic measure of activity in digital circuits*, 28th IEEE/ACM Design Automation Conference (DAC91), pp. 644-649, 1991.
- [Najm93] **Najm, F. N.**, *Transition density: a new measure of activity in digital circuits*, IEEE Transactions on CAD of Integrated Circuits and Systems, Vol. 12, No. 2, pp. 310-323, February 1993.
- [Najm94] **Najm, F. N.**, *A survey of power estimation techniques in VLSI circuits*, IEEE Transactions on VLSI Systems, Vol. 2, No. 4, pp. 446-455, December 1994.
- [Najm95] **Najm, F.**, *Feedback, correlation, and delay concerns in the power estimation of VLSI circuits*, 32nd ACM/IEEE Design Automation Conference (DAC95), pp. 612-617, 1995.
- [NaLi97] **Narayanan, U., Liu, C.L.**, *Low Power Logic Synthesis for XOR Based Circuits*, ICCAD97, pp. 570-574, November 1997.
- [NaGH95] **Najm, F. N., Goel, S., Hajj, I. N.**, *Power estimation in sequential circuits*, 32nd ACM/IEEE Design Automation Conference (DAC95), pp. 635-640, 1995.
- [NajZ95] **Najm, F., Zhang, M. Y.**, *Extreme delay sensitivity and the worst-case switching activity in VLSI circuits*, 32nd ACM/IEEE Design Automation Conference (DAC95), pp. 623-627, 1995.
- [NBYH90] **Najm, F. N., Burch, R., Ying, P., Hajj, I.**, *Probabilistic simulation for reliability analysis of CMOS VLSI circuits*, IEEE Trans. on Computer-Aided Design, Vol. 9, pp. 439-450, April 1990.
- [NeNa97] **Nemani, M., Najm, F.N.**, *High-Level Area and Power Estimation for VLSI Circuits*, ICCAD97, pp. 114-119, November 1997.

- [OlVexx] **Older, W., Vellino, A.**, *Constraint arithmetic on real intervals*, Computing Research Laboratory, Bell-Northern Research (BNR).
- [OngY94] **Ong, P. W., Yan, R. H.**, *Power-conscious software design - a framework for modelling software on hardware*, IEEE Symposium on Low Power Electronics, pp. 36-37, 1994.
- [Pede84] **Pederson, D. O.**, *A historical review of circuit simulation*, IEEE Transactions on Circuits and Systems, CAS-31(1): pp. 103-111, January 1984.
- [RaDJ96] **Raghunathan A., Dey S., Jha N. K.**, *Register-Transfer Level Estimation Techniques for Switching Activity and Power Consumption*, ICCAD96, pp. 158-165, 1996.
- [RagJ95] **Raghunathan, A., Jha, N.**, *ILP formulation for low power based on minimizing switched capacitance during data path allocation*, ISCAS95, 1995.
- [RaSH97] **Ramprasad, S., Shanbhag, N. R., Hajj, I. N.**, *Achievable Bounds on Signal Transition Activity*, ICCAD97, pp. 126-129, November 1997.
- [Röth94] **Röthing, W.**, *Modélisation comportementale de consommation des circuits numériques*, Ph.D. thesis, l'École Nationale Supérieure des Télécommunications, Paris, France, 1994.
- [RoyP92] **Roy, K., Prasad, S.**, *SYCLOP: Synthesis of CMOS logic for low power applications*, ICCD92, pp. 464-467, 1992.
- [Rues96] **Ruess, H.**, *Hierarchical Verification of Two-Dimensional High-Speed Multiplication in PVS: A Case Study*, FMCAD'96, pp. 79-93, 1996.

- [SaBr94] **Saldanha, A., Brayton, R. K.**, *Circuit structure relations to redundancy and delay*, IEEE Transactions on CAD of Integrated Circuits and Systems, Vol. 13, No. 7, pp. 875-883, July 1994.
- [SaVi81] **Sangiovanni-Vincentelli, A. L.**, *Circuit simulation*, In **Antognetti, P., Pederson, D. O., De Man, H.**, *Computer Design Aids for VLSI Circuits*, pages 19-112, Sijthoff & Noordhoff, Alphen aan den Rijn, The Netherlands; Rockville, MD, USA, 1981. UI : 621.3819535C739.
- [Schr97] **Schroeder, S.**, *Turning to Formal Verification*, Integrated System Design Archive, <http://www.isdmag.com/Editorial/1997/CoverStory9709.html>, September, 1997.
- [SeAg86] **Seth, S. C., Agrawal, V. D.**, *An exact analysis for efficient computation of random-pattern testability in combinational circuits*, 16th International Symposium on Fault Tolerant Computing Systems, Vienna, Austria, pp. 318-323, July 1986.
- [ShAu88] **Schulz, M. H., Auth, E.**, *Advanced automatic test pattern generation and redundancy identification techniques*, 18th International Symposium on Fault-Tolerant Computing, pp. 30-35, June 1988.
- [Shen96] **Shenoy, S.**, *Switching Activity in CMOS Digital Circuits*, M.Sc. thesis, McGill University, 1996
- [SHGB94] **Saldanha, A., Harkness, H., McGeer, P. C., Brayton, R. K., Sangiovanni-Vincentelli, A. L.**, *Performance optimization using exact sensitization*, 31st ACM/IEEE Design Automation Conference (DAC94), pp. 425-429, 1994.
- [SGDK92] **Shen, A., Ghosh, A., Devadas, S., and Keutzer, K.**, *On average power dissipation and random pattern testability of CMOS combina-*

tional logic networks, IEEE/ACM International Conference on Computer-Aided Design, pp. 402-407, Santa Clara, CA, 1992.

- [SiSa93] **Silva, J. P., Sakallah, K. A.**, *An analysis of path sensitization criteria*, ICCD-93 Conf., Cambridge, Mass., October 1993.
- [SiSa94a] **Silva, J. P., Sakallah, K. A.**, *Dynamic search-space pruning techniques in path sensitization*, 31st ACM/IEEE Design Automation Conference (DAC94), pp. 705-711, 1994.
- [SiSa94b] **Silva, J. P., Sakallah, K. A.**, *Efficient and robust test generation-based timing analysis*, ISCAS94, pp. 303-306, 1994.
- [SiSt97] **Sivaraman, M., Strojwas, A.J.**, *Timing Analysis Based on Primitive Path Delay Fault Identification*, ICCAD97, pp. 182-189, November 1997.
- [SivS93] **Sivaraman, M., Strojwas, A. J.**, *Accurate timing verification with correlated component delays*, Workshop on Design Methodologies for Microelectronics and Signal Processing, Glivice-Cracow, Poland, October 1993.
- [SPEC97] *Processor performance comparison*, `ftp://ftp.cdf.toronto.edu/pub/spectable`, May 1997.
- [StaB94] **Stan, M., Burleson, W.**, *Limited-weight codes for low-power I/O*, IWLPD94, pp. 209-214, 1994.
- [StaH90] **Stark, D., Horowitz, M.**, *Techniques for calculating currents and voltages in VLSI power supply networks*, IEEE Transactions on CAD, Vol. 9, No. 2, pp. 126-132, February 1990.

- [StBe91] **Steward, R., Benkoski, J.**, *Static timing analysis using interval constraints*, ICCAD91, pp. 308-311, 1991.
- [Syna97] *DelayMill Datasheet*, http://www.synopsys.com/products/ptg/delaymill_ds.html, Synopsys, Inc., 1997.
- [Synb97] *PrimeTime User Guide*, Synopsys, Inc., 1997.
- [Sync98] *Formality User Guide*, Synopsys, Inc., 1998.
- [TiMW94a] **Tiwary, V., Malik, S., Wolfe, A.**, *Compilation techniques for low energy: an overview*, IEEE Symposium on Low Power Electronics, pp. 38-39, 1994.
- [TiMW94b] **Tiwary, V., Malik, S., Wolfe, A.**, *Power analysis of embedded software: a first step towards software power minimization*, IEEE Transactions on VLSI Systems, Vol. 2, No. 4, pp. 437-445, December 1994.
- [TPCD94] **Tsui, C.-Y., Pedram, M., Chen, C.A., Despain, A. M.**, *Low power state assignment targeting two- and multi-level logic implementations*, ICCAD94, pp. 82-87, 1994.
- [TsPD93] **Tsui, C.-Y., Pedram, M., Despain, A. M.**, *Efficient estimation of dynamic power consumption under a real delay model*, ICCAD93, pp. 224-228, 1993.
- [VaSM93] **Vanoostende, P., Six, P., De Man, H. J.**, *PRITI: Estimation of maximal currents and current derivatives in complex CMOS circuits using activity waveforms*, EuroDAC93, pp. 347-353, 1993.
- [WaFF94] **Wang, J.-H., Fan, J.-T., Feng, W.-S.**, *An accurate time-domain current waveform simulator for VLSI circuits*, EuroDAC94, pp. 562-566, 1994.

- [XakN94] **Xakellis, M. G., Najm, F. N.**, *Statistical estimation of the switching activity in digital circuits*, 31st ACM/IEEE Design Automation Conference (DAC94), pp. 728-733, 1994.
- [Youn96] **Young, L. H.**, *Building a better Bug Trap*, Electronic Business Today, November 1996.
- [ZCMM96] **Ziegler, J. F. , Curtis, H. W., Muhlfeld, H. P., Montrose, C. J.**, *IBM experiments in soft fails in computer electronics (1978-1994)*, IBM Journal of Research and Development, Vol. 40, No. 1 - Terrestrial cosmic rays and soft errors, 0018-8646/96, 1996
- [ZCSR95] **Zejda, J., Cerny, E., Shenoy, S., Rumin, N. C.**, *Gate-level power estimation using transition analysis*, Workshop on Design Methodologies for Microelectronics (DMM95), pp. 111-119, 1995.
- [ZCSR96] **Zejda, J., Cerny, E., Shenoy, S., Rumin, N. C.**, *Bounding Switching Activity in CMOS Circuits Using Constraint Resolution*, European Design & Test Conference (EDAC), Paris, March 1996.
- [ZhWo97] **Zhou, H., Wong, D.F.**, *An Exact Gate Decomposition Algorithm for Low-Power Technology Mapping*, ICCAD97, pp. 575-580, November 1997.

Appendices

APPENDIX A

Technical Documentation and Utilization

This appendix shows the architecture of the C++ implementation of our verification method based on constraint resolution.

A-1. Architecture

A-1.1 Modules

The entire C++ implementation is divided into several modules as shown in Figure 163. The Verilog parser produces IC InCore data structures (ic module). These are then con-

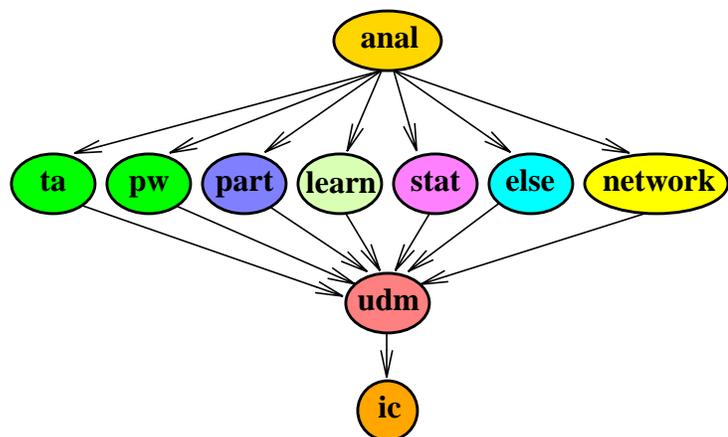


FIGURE 163: Architecture

verted into another database (`udm` module) which is accessed by all other upper layers. The `udm` module contains the translator from the IC InCore data structures, `udm` circuit database, and the constraint resolution engine. The top module (`anal`) implements the case analysis. It uses all the other modules: the `ta` module for timing verification and the `pw` module for switching activity verification. The algorithms and the data structures for partitioning are in the `part` module, and for learning in the `learn` module, for topological analysis in the `stat` module. Two other independent modules are used in the case analysis: the CPU and memory usage measurement module called `else` and the `network` module containing the necessary algorithms for the parallel implementation of the power analysis over a network of workstations.

The architecture is very closely captured by the physical organization of files in a directory tree as shown in Figure 164. The main executable `ICTest` is located in the `bin` directory.

A-1.2 Revision Control

Scripts for development and for revision control can be found in `etc/bin` directory. The RCS revision control is used; therefore, each directory has a subdirectory called `RCS` which contains the version files (`*`, `v`). To overcome one of the major problems of RCS (retrieving the version of the software rather than that of the individual files) the script `co_all` checks out all source files. It accepts the date option of the normal `co` command, hence any version of the whole software can be obtained by, e.g., `etc/bin/co_all -d 'Thu Jun 29 12:03:30 EDT 1995'`. This would recover the software as it existed on Thursday June 29, 1995 at 12:03:30. The `README` file in the root directory contains more information about the revision control scripts as well as the dates (in the format for `co_all`) of the various software versions and the description of all major changes.

A-1.3 Benchmarks

The source code of the benchmark and other test circuits is located in `etc/circuits` directory. It includes the Verilog source code and some schematics in Post-Script for small circuits, and local-file-system symbolic links to large circuits. Netlist translators are located in the `etc/translators` directory.

A-1.4 Source Code

The compilation is controlled by a single `Makefile` in the root directory. Therefore any `make` command must be issued from the root directory of the project. During compilation the object files are stored in the same directory as the source files. The libraries are created in the `lib` directory, one static and/or dynamic library for each module. To compile, do `make depend && make`.

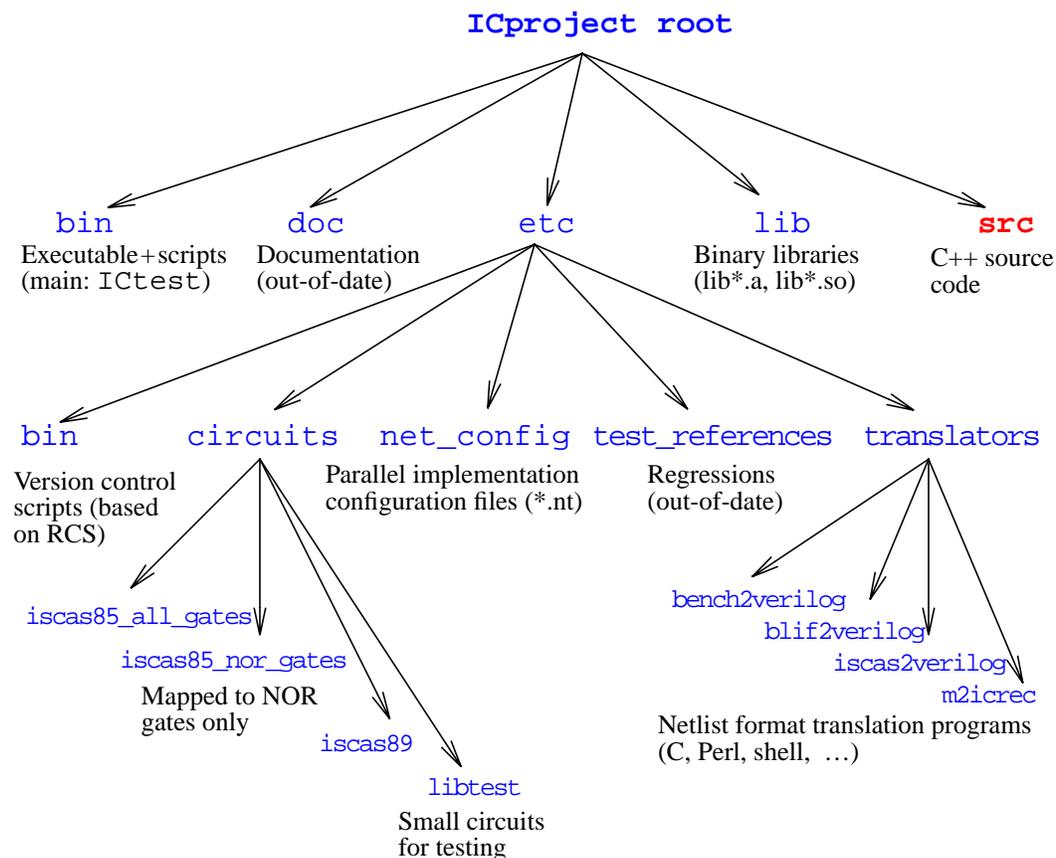


FIGURE 164: Directory tree

The source code tree located in the `src` directory. It is described in Figure 165. The Verilog/VHDL parser interface and the top-level source file `ICaction.cxx` are located in `src/ic`, our circuit representation and constraint resolution code in `src/udm`. The code modeling waveforms by intervals (for timing verification) is in `src/ta`. The code modeling waveforms as sets of transitions (for switching activity verification) is in `src/pw`.

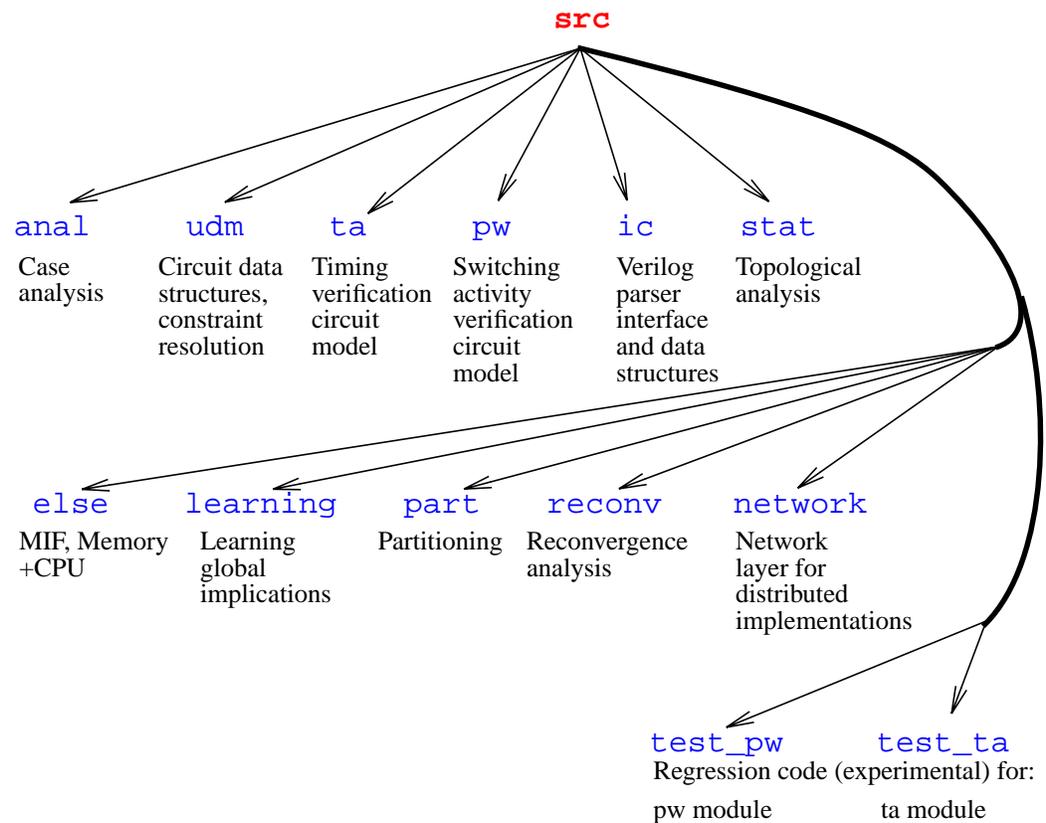


FIGURE 165: Directory tree - source files

A-2. Modules

In this section we discuss in detail implementation of each module. The author recommends to read the source code in parallel with this section or rather skip this section and return to it once you start programming inside the ICproject.

A-2.1 Parser interface and the top level (*ic*)

The parser itself and the IC InCore data structures are in a separate directory referenced from the Makefile by variable `ICBNR_DIR`. The interface to the IC InCore is in `src/ic`. The main executable is `src/ic/ICtest.cxx`. Most of the functionality is implemented in `src/ic/ICaction.cxx`. Top of the file is devoted to configuring the IC InCore, calling the parser and processing the command line arguments by calling procedures in `src/ic/ICpreProc.cxx`. The rest of the `src/ic/ICaction.cxx` has two similar parts: one (called *TA*) for the user specified `--ta` option, the other (*PW*) for `--pw`. The option influences which is the lowest level of signal representation used: `--ta` for waveforms with intervals (`src/ta/TAwaveform.cxx`) and `--pw` waveforms with transition sets (`src/ta/TAwaveform.cxx`). In each of these two blocks of code, the appropriate circuit (i.e., the constraint system) is created using the IC InCore data structures that are already in memory. Then *PW* or/and *TA* analysis or any other procedure and reporting are performed and the circuit instance is deleted from memory. Both of these blocks can be active during one execution, but they are executed in sequence, *PW* first, then *TA*.

A-2.1.1 Shared part

The IC Incore part starts in the procedure `mytestAction`. It creates data structures for measuring memory consumption and CPU time (`TimeAndSpace`), for Sandeep Shenoy's [Shen96] exact transition counting algorithm (`McG_*` variables), creates the circuit IC Incore representation (`"zdesign = new ICdesign;"`), parses the input file (`"loadFName"`) by calling `loadFunc`, binds the design (`"zmod->bind(...)"`), instantiates a working copy of the circuit (`"zmod->instantiate(doInst, "work", ...)"`), elaborates and then flattens the working instance of the design (`"zmod->addFlat();"`). The last part of the IC InCore code prints the contents of the IC InCore. That part of the code is active if the Boolean variable `udm_mode_icdb` is true, which is controlled by the `--icdb` command line option to the `src/bin/ICtest`.

The verification by constraint resolution follows. First, common data structures are created, e.g., `EXPORTmif` for graphical output in MIF format if the Boolean variable `udm_mode_mif` is true (option `--mif`). Then, the circuit netlist is exported in the Tulip format (`.m`) if the Boolean variable `udm_mode_tulip` is true (option `--tulip`). The pointers to the partitioning interface (`PARTpartitions`) and to the network interface (`NTnetwork`) are created. At this time they are left empty (`NULL`). That is the end of the common part. It is followed by two blocks, *PW* and *TA* to be described next.

A-2.1.2 Switching activity analysis (*PW*)

The switching activity analysis begins with the line `if (udm_mode_all || udm_mode_pw) // do power`. The Boolean variable `udm_mode_all` is controlled by the option `--all`, which replaces several of the most often used options. Generally, it means to perform both the timing and the switching activity analysis with the *PW* type of waveforms, and the timing analysis only with the *TA* type of waveforms. In both cases, the most advanced type of the case analysis procedure (HPCA) is used. Running `bin/ICtest` with no arguments shows the list of options and the exact definition of `--all`.

The Boolean variable `udm_mode_pw` is controlled by the `--pw` option. It selects the *PW* type of waveforms. The constraint system is built by calling the constructor of the `UDM_PW` class (`zPW = new PW_UDM(zmod, EM);`) which translates the IC InCore data structures into the constraint system (`UDM_PW`). The class `UDM_PW` is derived from the `UDM` class; therefore, most of the translation code is found in `src/udm/UDM.cxx` file, procedure `UDM::sort()`. The circuit netlist is then output in the Tulip format (if `udm_mode_tulip` is true) and the circuit name is written into the MIF output file (if `udm_mode_mif` is true).

The number of primary inputs and outputs are stored in `Results_pw.PInum_` and `Results_pw.POnum_`. Circuit topological information, i.e., the fanout and

the topological distances from the primary inputs are initialized afterwards (`zPW->circ_topol()->initialize();`).

Reconvergent region analysis follows. It is selected by the Boolean variable `udm_mode_maxrcvg` (controlled by the `--maxrcvg` option). The analysis is performed by the constructor of the reconvergent region database class (`regions = new RCVregions(zPW,udm_mode_maxrcvg);`). If the Boolean variable `udm_mode_recon` constant is true as well (controlled by `--recon`) then the reconvergent regions are saved in the file specified on the command line after the `--recon` option.

The reconvergent region analysis is followed by learning of global implications. If the Boolean variable `udm_mode_learn` is true (the `--learn` option) then the learning data structures are initialized (`LEARNlearning* learning = new LEARNlearning(zPW, regions->regions());`) and learning is performed (`learning->learn();`). Note that there is an argument `regions` in the initialization since learning is done along the reconvergent regions only in the current implementation.

The `system_info` object which is instantiated after the learning procedure gathers system information such as hostname and system type which are printed by `system_info.print();`. Then information about the circuit (such as number of gates, nets, largest fanout, etc.) is retrieved by accessing the circuit representation (object `zPW` of class `UDM_PW`) and printed.

One very important global variable is initialized at this time too, `common_delay_factor = zPW->comm_delay();`. It is the greatest common factor off all delay values. It is used to scale delay values so that, minimum number of necessary intervals of discrete time are used. E.g., a circuit with all gate delays of 10 units and the longest path of 20 levels does not cause the variables of the constraint system to have 200 intervals of discrete time but only 200

divided by the `common_delay_factor=10`. I.e., only intervals of discrete time where the signals can potentially change are maintained. From the outside, however, this is hidden, and all delays are reported as if the scaling mechanism did not exist. Note that for realistic gate delays such as after technology mapping the common factor will most likely be 1.

After printing the topological information, some experimental code enabled by various `#define` options is executed. The options must be defined by `#define` at the beginning of the `src/ic/ICaction.cxx` file, and the code must be recompiled to use these features. The `PRINT_TOPOL_DIST` prints the topological distance from the primary inputs to the inputs of each gate.

The end of the topological analysis can be identified in the source code by the call to `ts.step_ret` with the argument “`static analysis`” which prints CPU time and memory usage. Then, the initial universal waveforms (any transition possible at any time) of the constraint system are saved (`zPW->push_state();`) on the stack of circuit states. Subsequently, the test mode is invoked. Currently, only testing of equivalence of AND gate and its OR-NOT structural model is available. This test is called AOQ (And Or eQuivalency) and is invoked if the `udm_mode_test` constant contains the string “AOQ” (option `--test`). The special circuit `etc/circuits/libtest/aoq.v` is required for this test.

The AOQ test is invoked on an unconstrained circuit before setting operating conditions because it generates random waveforms different (“larger” in the sense of constraint sets) from the operating conditions. Since the constraint system is built as non-increasing (conjunctive constraints), application of any larger-than-existing set is not possible. Therefore the initial universal waveforms are saved on the stack and restored after AOQ terminated.

The operating conditions are set by the `zPW->set_standard_input()` call. This selects the appropriate primary input waveforms. The fixpoint after the con-

straint system is recalculated by `zPW->forward()`, followed by `zPW->stabilize()`. The `UDM::stabilize()` procedure recomputes the fixpoint of the constraint system. In the first fixpoint computation (when waveforms on all nets but PIs are universal) the `UDM::forward()` is called before `UDM::stabilize()`. In a general case of a constraint assigned anywhere but on PIs the `UDM::stabilize()` is more efficient than the couple `{UDM::forward(); UDM::stabilize()}`. If the constraints are assigned only on PIs then `UDM::forward()` is faster because it does not propagate events backward (from outputs of a gate to its inputs).

Once the input waveforms are assigned and the fixpoint of the constraint system is recomputed, the number of transitions is counted (“`power_forw=zPW->count_trs()`”). This is the initial upper bound on switching activity. The maximum delay is also calculated, just for information (“`max_delay_forw=zPW->calc_max_delay()`”). The waveforms obtained after setting the operating conditions are printed if the Boolean variable `udm_mode_wf` is true (option `--wf`).

Partitioning follows next. In the current implementation, there is only the initialization (“`partitions = new PARTpartitions(zPW,udm_mode_partition);`”) and a marker to indicate where to put user code (“`// do your job BELOW this line`”). The constant `udm_mode_partition` (option `--partition`) specifies which partitioning algorithm is to be used. For now, the only implemented algorithm is “one gate = one block” for testing purposes.

The code which follows the partitioning code in the `src/ic/ICaction.cxx` file was supplied by Sandeep Shenoy [Shen96]. It is an exact transition counting algorithm for a block with limited number of inputs. The algorithm finds such a feasible subset of abstract waveform on each net that the abstract waveform is only a simple waveform, and that the number of transitions in the block is maximal.

Since the code was developed apart from the system at the time when the partitioning interface did not exist yet, it uses its own partitioning. It should be integrated with the partitioning interface (future work). The algorithm is activated by the option `--maxMcGinp`.

Another experimental code follows the exact transition counting code in the `src/ic/ICaction.cxx` file. It is compiled optionally if the symbol `DO_CM_PW_NODE_LIST` is defined. It creates a list of nets for the case analysis based on a heuristics. Details can be found in the source code for the method `PWwaveforms::get_cm()`.

Another experimental code whose inclusion is controlled by the symbol `PRINT_HEADLINES_AFTER_FW` is followed by code for debugging. By default it is commented out, because it is specific to circuits `c3540`, `c499`, and `c17`. This code was made obsolete by introducing interactive constraint evaluation (option `--CAList MANUAL`). However, if you need to add any special code for debugging, this is the place to do so.

Since some of the conditionally compiled code builds its own list of nets for the case analysis, there is a flag called `NORMAL_RUN` which delimits the default (“normal”) code (`#ifdef NORMAL_RUN`). You should `#undefine` this flag if you write a code which interferes with the default (“normal”) code. The default code checks the Boolean variable `udm_mode_CAList` (`--CAList HEUR` option) and builds the list of nets where the case analysis will be performed according to the selected heuristic `HEUR`. Each heuristic has its own list of nets, but at the end produces `nlist_pw` of which only a small portion is used for the case analysis. The used portion starts at the beginning of the original list and is long enough to contain all PIs. All PIs must be contained to guarantee termination of the case analysis.

The documented heuristics accessible from the command line are FANOUT, RCVFAN, and PIFAN. There are several additional experimental heuristics which are conditionally compiled. The heuristic enabled by `"#define CM_PW_NODE_LIST"` creates a list of nets for the case analysis based on modifications to the waveforms by merging into four waveform classes during the initial forward propagation, see comments in procedure `PWwaveforms::get_cm()` for details. The heuristic enabled by `"#define TD_PW_NODE_LIST"` is based on the topological distances of each gate input from the PIs. It creates a list of nets driving gate inputs with the shortest and longest distances from PIs for each gate. The list is sorted by the sum of the differences between the shortest and longest distance for each gate. The heuristic enabled by `"#define AM_PW_NODE_LIST"` creates list of nets based on how much they overlap during merge (union of waveform classes during positive (forward) operations on abstract waveforms). The heuristic enabled by `"#define REC_PW_NODE_LIST"` is an older version of the heuristic RCVFAN. It creates a list of stems of reconvergent regions sorted by decreasing size (in gates) of the reconvergent regions.

The heuristic code is followed by the code that prepares list `nlist_ca_pw` of nets for the case analysis based on the list supplied by the active heuristic. The list can be printed if the symbol `PRINT_CA_NODE_LIST` is defined. If the symbol `SET_PO_TIME_CONTR` is defined then the true circuit delay can be interactively entered as an additional constraint in the constraint system.

The analysis starts by a simulation that determines a lower bound on switching activity. The simulation is run if `udm_mode_sim` is true (option `--sim`). First 4 special pairs of test vectors (first 4 bits of the PI vectors shown) are tried: `0000→1111`, `1111→0000`, `0101→1010`, `1010→0101`. Then an exhaustive simulation follows for circuits with up to 6 primary inputs or a random simulation for 1024 input vectors for any other circuit. A 2-D graph of the results can be printed using class `TwoDgraph`.

The next part of the `src/ic/ICAction.cxx` file is code for case analysis. The oldest version of the case analysis, *SCA* (“`new ANALcase_analysis(...)`”), is run when `udm_mode_sca` is true (option `--sca`), the *PCA* (“`new ANALpca(...)`”) when `udm_mode_pca` is true (option `--pca`), and *HPCA* (“`new ANALhpca(...)`”) if `udm_mode_hpca` is true (option `--hpca`). Actually, all three analyses can be run twice: once with the decision function returning the circuit delay when `udm_mode_verify_delay` is true (option `--verify DELAY`) or with the decision function returning the switching activity when `udm_mode_verify_power` is true (option `--verify POWER`). Both are run when the option `--verify` is set to `DELAY:POWER`. Once the case analyses are finished the instantiated objects for the case analyses and the circuit with the PW waveforms (`PW_UDM`) are deleted.

A-2.2 Circuit representation (`udm`)

The circuit netlist is translated from the IC data structures into a constraint system. The components of the constraint system and its basic functionality such as state saving and fixpoint calculation are in the *UDM* module.

The analogy between a circuit and its corresponding constraint system is shown in Figure 166. Each gate (constraint) is a `UDMgate` object which has pointers

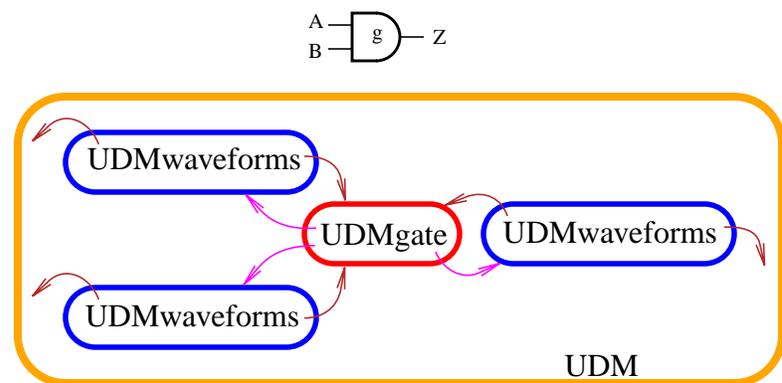


FIGURE 166: Circuit netlist and corresponding C++ classes

to sets of input and output nets. Each net (a variable of the constraint system containing an abstract waveform) is a `UDMwaveforms` object. The whole circuit is a `UDM` object. Next we describe the basic components of the constraint system and then look closely at the individual C++ classes.

A-2.2.1 Constraint system fixpoint

Circuit nets are represented by `UDMwaveforms` objects. Each holds an abstract waveform representation `UDMwaveformSet`. The constraints are implemented by methods `UDMgate::forward()` and `UDMgate::backward()`. Fixpoint calculation is achieved by calling `UDMgate::forward()` and `UDMgate::backward()` for all gates scheduled for evaluation (to be explained later). Whenever `UDMgate::forward()` is called, the gate looks at the `UDMwaveformSet` object of each of the connected input nets, computes the output abstract waveform, writes it into a local buffer `UDMwaveforms::local` and calls `UDMwaveforms::merge_local()` on each of the output nets to perform the intersection of the current abstract waveform `UDMwaveforms::output` (confusing name ...) and the contents of the local buffer. The function `UDMwaveforms::merge_local()` also detects the change in the current abstract waveform on the net and inserts pointers to all gates connected to the net in the queue `UDMgate_queue` of scheduled gates. The very first events at the beginning of the calculation (pointers to gates) are inserted by generating the appropriate primary input waveforms and by calling `UDMwaveforms::merge_local()`. The waveforms are generated by a low-level procedure `UDMwaveform::setPIconstraint(delmod)`. The parameter `delmod` describes the delay model used. Currently, input waveforms corresponding to the 2-vector transition and the floating-mode modes are supported. The data structures used representing a circuit net are described in detail in the next section.

A-2.2.2 The data structure representing a circuit net

The class `UDMwaveforms` is a net representation. It contains an abstract waveform (`UDMwaveformSet`), pointers to the sets of driving and driven gates (`UDMgateSet* fanin, fanout`), a stack of abstract waveforms (`UDMwaveformsDynStack* waveformsDynStack`) and a list of branches (`UDMbranches* branches_`) which adds annotation to individual branches reachable through the `fanout` set of driven gates.

This separation is historical - the “branches_” were added later to speed up reconvergent region analysis. The stack of constraint system variables implements incremental constraint-system state saving (explained in Section A-2.2.4). The structure of a UDMwaveforms object is shown in Figure 167.

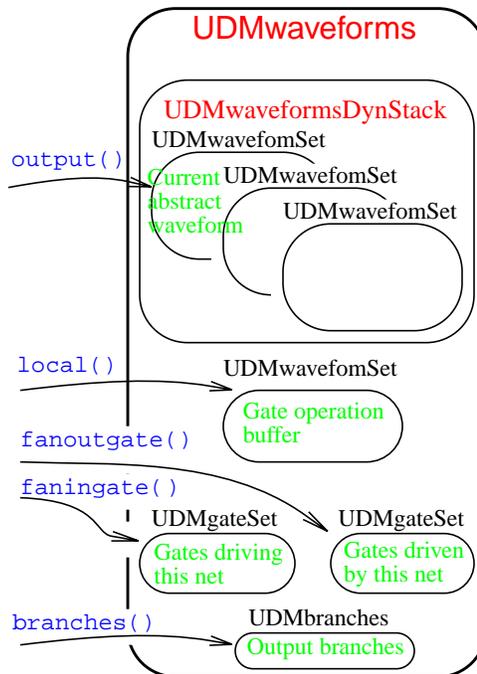


FIGURE 167: Net - UDMwaveforms

One abstract waveform or a variable of the constraint system is stored in an object of class UDMwaveformSet. The name may be confusing but it is a set of up to four objects of class UDMwaveform that implement waveform classes. Class UDMwaveform is an abstract class which provides an interface for implementing the lowest level of the constraint system. It holds the necessary variables and C++ methods for performing operations such as

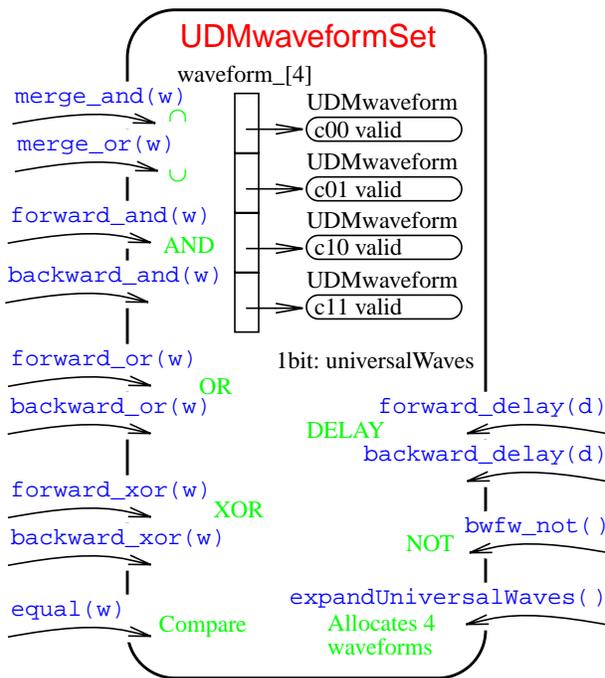


FIGURE 168: Abstract waveform - UDMwaveformSet

union, intersection, AND, OR, XOR, and NOT. Classes UDMwaveformSet and

UDMwaveform are shown in Figure 168. The implementation of a gate object is described next.

A-2.2.3 The data structure representing a gate

The class UDMgate represents gates. It contains the methods `forward()` and `backward()` which are implemented differently by derived classes to provide the desired functionality of the specific gates such as *AND*, *OR*, *XOR*, *NAND*, etc. The inputs and outputs are accessible through pointers to the sets of the input and output nets (UDMwaveforms-Set* `input`, `output`) and their corresponding access methods, as shown in Figure 169.

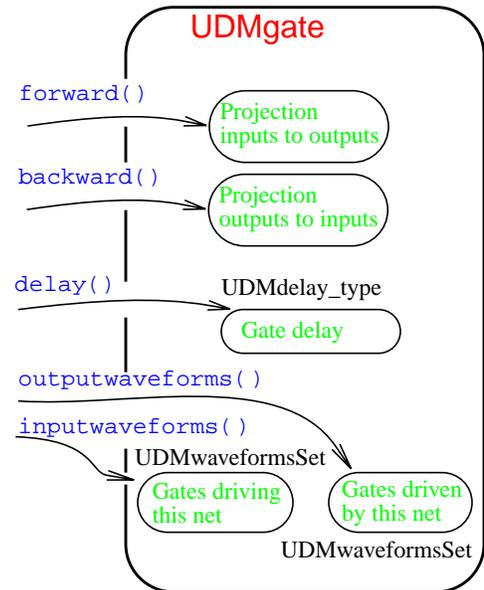


FIGURE 169: Gate - UDMgate

Several types of gates were developed. The UDMgate is a basic gate suitable for 2 or 3-input gates. Its `forward` function computes the output abstract waveform as a union of 4^N waveform classes, where N is the number of gate inputs. It builds a table of N entries, takes one waveform class from each input, computes the desired function in a loop over N -tuples and performs a union of the resulting 4^N individual waveforms.

The UDMgatem uses a structural model for symmetrical gates, as shown in Figure 170 for an N -input gate. There are $N-1$ two-input gates called “internal”. All of them perform the same function. The function is implemented by two methods, `forwardSetInt()` and `backwardSetInt()`. These are only abstract in the class UDMgatem. they are defined in the gate specific ancestor classes (UDMandm, ...).

There is one output gate which usually implements negation and delay, but any desired functionality can be written into the methods `forwardSetOut()` and `backwardSetOut()`. Class `UDMgatem` is the default class used for most gates. Only the buffer and the inverter use directly the base class `UDMgate`. To implement a symmetrical gate, one has to create a new class which inherits `UDMgatem` and implements the four

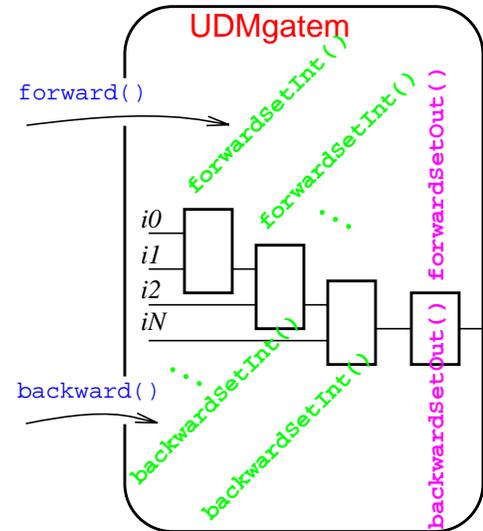


FIGURE 170: UDMgatem

A-2.2.4 The data structure representing a circuit

The UDM class is a circuit representation, i.e., it implements the constraint system. It contains sets of nets (“UDMwaveformsSet local”) and sets of gates (“UDMgateSet gates”). Lists of the primary input and the primary output nets (“UDMwaveformsSet input, output”) are subsets of the set local of all nets. The queue `UDMgate_queue` of gates for event driven evaluation is also a part of the constraint system. In the versions supporting sequential circuits, there is also a list of flip-flops which is a subset of the set gates of all gates. The constraint system provides interface functions `forward()`, `backward()`, `stabilize()` described in Section A-2.2.1, and `push_state()`, `pop_state()` to save and recover the state of the constraint system. The state saving is used for backtracking during case analysis and some other algorithms. The interface also allows to print the current state of the constraint system variables using the methods `print_*`, and to dump them in the MIF format using the methods `mif_*`. The UDM also contains the topological information `STATcircTop`, a method returning topological delay “`int topol_delay()`”, a method computing the time of the latest transition in the current state of the constraint system “`int calc_max_delay()`”, a method which computes a certain property, such as

circuit delay or switching activity “virtual int decis_func (df_property prop)”. The structure of the class UDM is shown in Figure 171. Notice differences compared to what we said earlier: the main object is not called UDM but PW_UDM, gates UDMnandm, not UDMgate, nets PWwaveforms not UDMwaveforms. This is because the figure shows a realistic instantiation of the objects during switching activity verification. As it will be described in detail in the two following sections, the PW* classes are derived from the UDM* classes for switching activity verification and the TA* classes are derived from the UDM* classes for timing verification. The rule of thumb is that many of the UDM* classes are abstract, i.e., it is not possible to instantiate an object of that class. Inheritance between the gate classes is by function, e.g., UDMnandm is derived from UDMgatem.

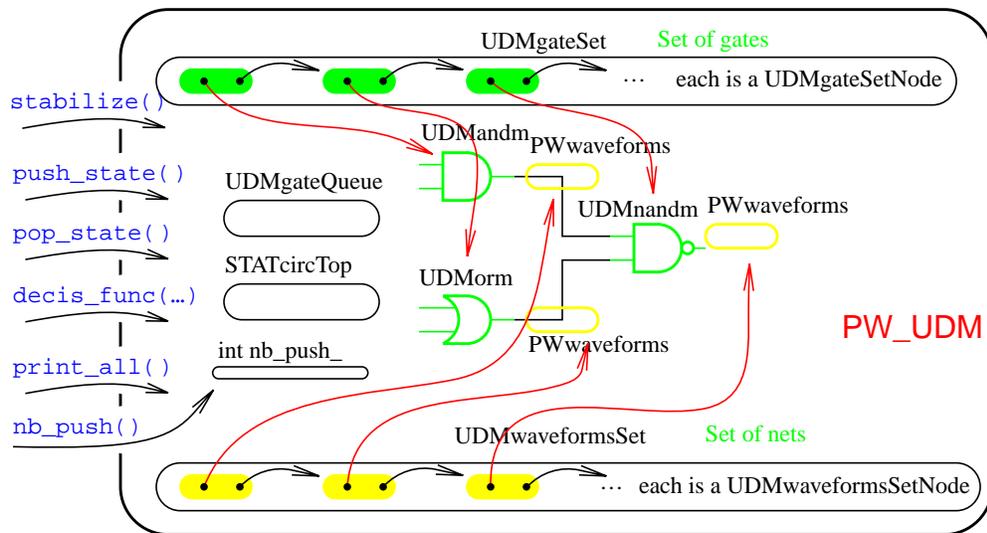


FIGURE 171: Circuit - UDM

The state saving is incremental: only the modified constraint system variables are saved. This is done using a local stack inside each net object. When the user asks to save the state by calling UDM::push_state(), only counter the nb_push_ inside UDM is incremented. When a net is to be modified by merge_local(), the stack level number inside the net is compared with the global one UDM::nb_push() and if the local one is lower then the net stack performs push operation. The method UDM::pop_state() visits all nets and compares the

stack level with the global level. If they do not match, the appropriate number (difference local minus global levels) of pop operations are performed on the net stack. The pop operation can also be made incremental by adding a code which checks for the stack level in `UDMwaveforms::output()` which is the access method to the current value of the abstract waveform variable of a given net.

A-2.3 Timing (τ_a)

The `ta` subdirectory contains code that is specific to the interval representation of waveform classes and is intended to be used for timing verification. The waveform class is `TAwaveform` which fits into the opaque data structures of `UDMwaveform` and implements all the methods of the `UDMwaveform` interface. The `TAwaveformSet` represents an abstract waveform for timing verification. The `TAwaveforms` is one net of circuit `TA_UDM`.

To use the timing waveforms and operations, one must build the data structure by calling the constructor of `TA_UDM`. This automatically instantiates all the classes `TAxxx` instead `UDMxxx`. One can also add any methods or data structures into the appropriate `TAxxx` classes rather than into the `UDMxxx` classes if they are specific to timing verification.

One `TAwaveform` is a set of four integer numbers and Boolean flag `valid` inherited from the `UDMwaveform` as shown in Figure 172. The `min` and `max` delimit an interval where transi-

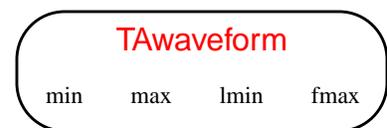


FIGURE 172: Timing class - TAwaveform

tions can occur. It is an upper envelope over any waveform expressed by this waveform class: it is guaranteed that no transition occurs before `min` and no transition occurs after `max`. The `fmax` value is the latest time when the first transition occurs if there is at least one transition. The `lmin` value is the earliest time when last transition occurs if there is a at least one.

A-2.4 Switching activity (pw)

The `pw` subdirectory contains code that is specific to the transition representation of waveform classes and is intended to be used for switching activity verification. The waveform class is `PWwaveform` which fits into the opaque data structures of `UDMwaveform` and implements all the methods of the `UDMwaveform` interface. An object of class `PWwaveformSet` represents an abstract waveform for switching activity verification. An object of class `PWwaveforms` represents one net of the circuit `PW_UDM`.

To use the *PW* waveforms and operations, the data structures must be built by calling the constructor of `PW_UDM`. This will automatically instantiate all the classes `PWxxx` instead of `UDMxxx`. One can also add any methods or data structures into the appropriate `PWxxx` class rather than `UDMxxx` if they are specific to switching activity verification.

The transition set for each interval of discrete time is expressed by a set of bits defined by the enumerated type `tr_type` which currently holds `t00`, `t01`, `t10`, `t11`. However, internally the representation is separate for each transition, i.e., there is one bit array for each transition type, where one bit is allocated for one discrete time interval. This allows to perform

most operations in parallel for n intervals on an n -bit computer. The transition bit arrays are dynamically allocated and pointed to by array `t` of four pointers. The bit array is an array of `base_type` which is defined as `unsigned long int` - the longest type the CPU can perform bit-wise operations on. This is to achieve portable scalability. Consider an OR operation on two waveform classes in a circuit where the resolution of 130 discrete timing intervals is required. If the source code is compiled on a 32-bit computer then five 32-bit words will be allocated and 5 bit-wise OR operations will be performed for each of `t00`, `t01`, `t10`, `t11`. On

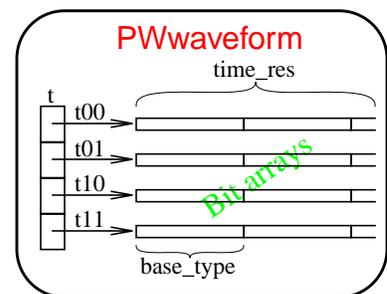


FIGURE 173: Transition set class - PWwaveform

a 64-bit computer three 64-bit words are allocated and 3 operations performed for each of t_{00} , t_{01} , t_{10} , t_{11} . The variable `time_res` tells how many bits are used in the bit array, since only whole words can be allocated (in the last example there are 192 bits allocated and `time_res=130`). The `PWwaveform` implementation is shown in Figure 173.

A-2.5 Topological analysis (`stat`)

Class `STATcircuitop` stores topological information about the circuit such as headlines and list of nets sorted by increasing fanout. It is instantiated in the `UDM` object.

Each net has its own topological information attached as well. The class `STATnodetop` contains the size of fanout, the topological distance from the primary inputs, and the reconvergent region overlap (used only for reconvergent region analysis). It is instantiated in the `UDMwaveforms` class.

A-2.6 Case analysis (`anal`)

The oldest *SCA* is implemented as the class `ANALcase_analysis`. When instantiated, the constructor takes the following arguments: a pointer to the circuit to work on, a pointer to the decision function, an ordered list of nets where to perform case analysis, a limit on the depth of the decision tree, and a limit on the number of decisions. The case analysis can be started either by the method `binary_search_using_cs` or by `desc_search_using_cs`. The former is a binary search, the latter a search from a (user-provided) upper bound down by specified increments.

The *PCA* is implemented as the class `ANALpca`. It uses a case analysis tree `ANALpca_pathSet` build as a list of paths `ANALpca_path`. The case analysis can be started by calling the method “`analyze(int max_passes, int max_level)`” where the arguments limit the size of the decision tree as in the

case of *SCA*. The results are stored in the class `ANALef_info` available from the case analysis by calling the `info()` method.

The most advanced case analysis *HPCA* is implemented as the class `ANALhpca`. It is the only case analysis which supports parallel evaluation. Therefore, it has a job scheduler which comes in two versions: `ANALhsched` is a serial one, and `ANALhsched_par` is a parallel one. Both use the same interface to communicate with the case analysis. The decision tree is stored in a heap represented by the class `ANALh_heap`.

A-2.7 Partitioning (part)

Partitioning is only an interface reserved for future development. The result of a partitioning operation can be stored in an object of class `PARTpartitions`. The algorithm which is used to partition the circuit is chosen according to the integer argument passed to the constructor. Right now the only partitioning algorithm is type *1* which means one gate is one block of the partition.

Each partition is an object of the class `PARTpartition`. It contains a list of gates, lists of entry, exit and all nets, the number of nets in the partition, and the number of gates in the partition. For the definitions see the file `src/part/PARTpartition.hxx`.

A-2.8 Learning (learn)

The class `LEARNlearning(UDM* circuit, RCVregionSet* rcv_set)` implements learning of global implications on a “circuit” along the reconvergent regions in “rcv_set”. Learning is done by calling the method `learn()`. The learned implications are attached to the netlist as virtual gates and thus automatically used in any operation over the constraint system.

All implications learned on one net are in `LEARNimplications`. The set of learned implications is returned by calling the method `implset()`. The constraint

system uses the methods “`int apply()`” and “`cl_set get_classes_to_remove (cl_set el_classes, cl_set stem_classes, unsigned int learned)`” to implement the virtual gates.

Each implication is an object of the class `LEARNimplication`. It contains a net where the implication is to be applied, the net which is the cause (where it was learned) and the type of the implication `impl_type`. It is an enumerated type of two values: `CNTL` and `RECON`. The value `CNTL` indicates that the implication (its forward complement) was caused by a controlling value, i.e., it can be deduced by the backward part of the gate projections. The value `RECON` indicates that the implication (its forward part) was caused by all gate inputs carrying non-controlling values, i.e., it cannot be deduced by the backward part of the gate projections (it was learned due to reconvergence).

A-2.9 Reconvergence (`reconv`)

The reconvergent regions are stored in `RCVregions`. It contains a set of regions `RCVregionSet` and forest of regions `RCVregionForest` to capture the inclusion of one region in another.

The region set `RCVregionSet` provides methods for sorting the regions according to several criteria, and an easy enumeration of regions through `RCVregionSetIter`.

Each reconvergent region `RCVregion` is described by the stem, the reconvergence gates, all the gates in the region, the entry nets to the region, the exit nets from the region, all the nets in the region, the closing nets of the region, and the set of subregions (to this region).

The algorithm to find the reconvergent regions is implemented as the constructor of the `RCVregions` class. It attaches five different tags to each branch of the circuit nets. The meaning of all tags is explained in the `src/reconv/RCVre-`

gions.cxx file. The tagging necessary to implement the algorithm is done by calling `tag_cone()`. A queue of gates is used for backtracking during the depth-first traversal through the cone. First a list of all primary reconvergent regions is created by tagging branches in the forward direction from the stem. Once all the primary reconvergent regions are known, each region is re-tagged in the forward direction as in the first pass. Then the region is tagged in the backward direction to mark all the nets and the gates in the region. The reconvergent region is then the intersection of the two cones. The method `identify_region()` traverses the region from the stem forward, looks at the tags, and records all nets and gates in the region data structure. Each net in the region is also checked for being a stem of another region. In this case, the region is added as a secondary reconvergence to the region currently under investigation. Regions of size exceeding the user-specified limit are forgotten, others are stored. The last step is the creation of a forest of inclusions of regions by calling the constructor of the `RCVregionForest` class with the set of regions as its argument.

A-2.10 Parallel (network)

The parallel case analysis algorithm is intended to improve performance of the case analysis. The implementation is described in detail in the chapter on parallel case analysis, in Section 5-3 on page 153.

A-2.11 Auxiliary (else)

The “else” directory is a collection of small modules which help the implementation and in some cases are also generic (i.e., used by more than one module).

The classes `Random1` and `Random2` are random number generators used for generating test vectors when the `--sim` option is used.

The class `Timer` allows to measure CPU time. The interface to that class is the `TimeAndSpace` class. One can obtain the CPU time and memory consumption by calling “`total(const char* text)`” or “`void step_ret(const`

`char* text, long int& MEM, double& CPUuser, double& CPUsystem)”. The class SysInfo provides the date and the hostname.`

The class `EXPORTmif` is an interface for producing a graphical report in the Maker Interchangeable Format (MIF) 4.0. It provides an HPGL-like user interface with primitives “`moveto(double x, double y)`”, “`line(double x, double y)`”, “`set_color(char* color)`”, etc. Many classes have the method “`mif(EXPORTmif*)`” which adds graphical representation of the object to the output MIF file. The `txt2twoDmif.cxx` file contains a stand-alone program which uses the `EXPORTmif` to translate a textual format into a 2-dimensional graph. The MIF interface was used extensively for creating this thesis.

A-3. Utilization

This section is a user manual how to use the switching activity verification system. The binary executable is called `ICtest` and is located in the `bin` directory in the project root directory. Running the `ICtest` with no arguments prints the list of available options and brief help messages.

A-3.1 Viewing the circuit netlist

Use the option `-r` to specify the top module name. The following command only parses the circuit netlist:

```
bin/ICtest etc/circuits/iscas85_all_gates/c17.v -r c17
```

To print the internal representation of a netlist stored in the IC InCore data structures use the option `--icdb` as follows:

```
bin/ICtest etc/circuits/iscas85_all_gates/c17.v -r c17 \
--icdb
```

To obtain the same information but from the UDM data structures use `--udmdb`. However, it will work only if `TA_UDM` or `PW_UDM` is instantiated, hence add the `--pw` or `--ta` option as follows:

```
bin/ICtest etc/circuits/iscas85_all_gates/c17.v -r c17 \
--udmdb --pw
```

A-3.2 Timing analysis

Timing analysis requires to specify `--ta` to use the timing waveforms. The following will build the constraint system with `TAwaveforms` and print the resulting upper bound on delay on the line denoted “Maximal circuit delay after forward:”:

```
bin/ICtest etc/circuits/iscas85_all_gates/c17.v -r c17 \
--ta
```

A-3.3 Switching activity analysis

Switching activity analysis requires to specify `--pw` to use the transition set waveforms. The following will build the constraint system with `PWwaveforms` and print the resulting upper bound on switching activity on the line denoted “Number of transitions in the circuit after forward:”:

```
bin/ICtest etc/circuits/iscas85_all_gates/c17.v -r c17 \
--pw
```

A-3.4 Decision function

The property that is being verified is set by option `--verify`. It can have one or more of the following options: `POWER`, `DELAY`, or `VDROP`. Multiple decision functions can be specified as a colon-separated string (e.g., `POWER:VDROP`). In such the case analysis (see below) is repeated for each decision function separately.

A-3.5 Case analysis

The case analysis is invoked by specifying the type of the case analysis (`--sca`, `--pca`, `--hpca`), the type of the constraint system (`--ta` or `--pw`), the property(ies) to verify (`--verify`), and the parameters for the case analysis. It can be done as follows:

```
bin/ICtest etc/circuits/iscas85_all_gates/c17.v -r c17 \
--pw --verify POWER --hpca
```

Or with using more command line arguments (see `--help`) to perform more precisely defined analysis, e.g., as follows:

```
bin/ICtest etc/circuits/iscas85_all_gates/c17.v -r c17 --pw
--verify POWER --hpca --CALim 200 --CAList FANOUT --hpcaStep 1
```

A-3.6 Parallel case analysis

The parallel case analysis is started as any other case analysis (does not work with `--pca` nor `--sca`) but with the option `--parallel` and with a configuration file. The configuration file is shared among the server and the slaves. It contains the TCP port to use, the communication time-out, the server hostname, and the option `detach` which causes the UNIX process to spawn a child detached from the terminal. A Perl script `bin/ICrun_par` is provided for easy start-up. It takes three arguments: the directory where to write the log files, a list of hosts to be used (the server's name appears in this list and the `--parallel` configuration file), and arguments for `ICtest` (with absolute paths because it is passed to UNIX `rsh`). An example follows:

```
bin/ICrun_par /tmp/run_001 machines.txt \
'cd /home/carre/zejda/work/power/ICproject; nice \
bin/ICtest_run etc/cicuits/iscas85_all_gates/c432.v \
-r c432 --hpca --verify POWER --pw --CALim 1000000 \
--hpcaStep 1 --parallel etc/net_config/carre8929_150.nt'
```

A-3.7 Manual mode

The manual mode is for playing with the constraint system. It allows the user to interactively assign constraints, to save the state of the constraint system, and to print out waveforms. A snapshot of a manual mode session follows:

```
bin/ICtest etc/circuits/iscas85_all_gates/c17.v -r c17 \
--pw --CAList MANUAL

...
Welcome to the ICtest's interactive mode. You can type "help".
ICtest> print 7
net: 7 : P=2
    |0
c00 _____

    |0
c01 __<>~~

    |0
c10 ~~<>__

    |0
c11 ~~<>~~

ICtest> tr
    There are 15 transitions in the circuit
ICtest> quit
```

The symbols used in the sample of textual output have the following meaning: “_” is a stable zero, “~” is a stable one, “<” is a set containing rising transition and stable value one. The symbol “<” is also used for a set containing stable one and rising transition. Similarly for symbol “>”. The textual output is intended mainly for debugging. The MIF graphical output provides exact detailed information about transition sets.

A-3.8 Various options

The reconvergent region analysis is invoked by, e.g., `--maxrcvg 85`, which would ignore all regions that have more gates than 85% of the total number of gates in the circuit. Learning can be added by the `--learn` option. It requires

reconvergent analysis. Partitioning is invoked by `--part 1` which uses method number *1* for partitioning. Waveforms can be printed after the constraint system is built and a fixpoint calculated (`--wf`) or during the case analysis (`--pcawaves`). Both of these options will also produce a graphical equivalent of the textual output into the MIF file when `--mif` is specified. The *HPCA* case analysis can dump its decision tree into a file periodically (`--checkpoint`) and be restored from a checkpoint file (`--recover`). The `--pw_time_resolution` can be used to limit the number of intervals of discrete time.

APPENDIX B

Conflict of N messages

Solution to this polynomial equation in t_j is needed to calculate the speedup considering conflicts of up to N messages:

$$\begin{aligned}
 & -4 t_c t_m t_j^N - 4 t_m s t_j^N - N t_j^{N+2} + 4 N t_m^3 t_j^{N-1} + t_c t_j^2 t_j^{N-1} + 4 t_c t_m^2 t_j^{N-1} + s t_j^{N+1} \\
 & + 4 t_m^2 s t_j^{N-1} - 6 N t_m^2 t_j^N + 5 N t_m t_j^{N+1} - N t_m^{N+2} 2^{N+1} + N^2 t_m^{N+2} 2^{N+1} - N^2 t_j \\
 & t_m^{N+1} 2^N = 0
 \end{aligned}$$

APPENDIX C

Decision Trees

We present here the images of the case analysis decision trees for 1000 decisions on nets sorted by decreasing fanout.

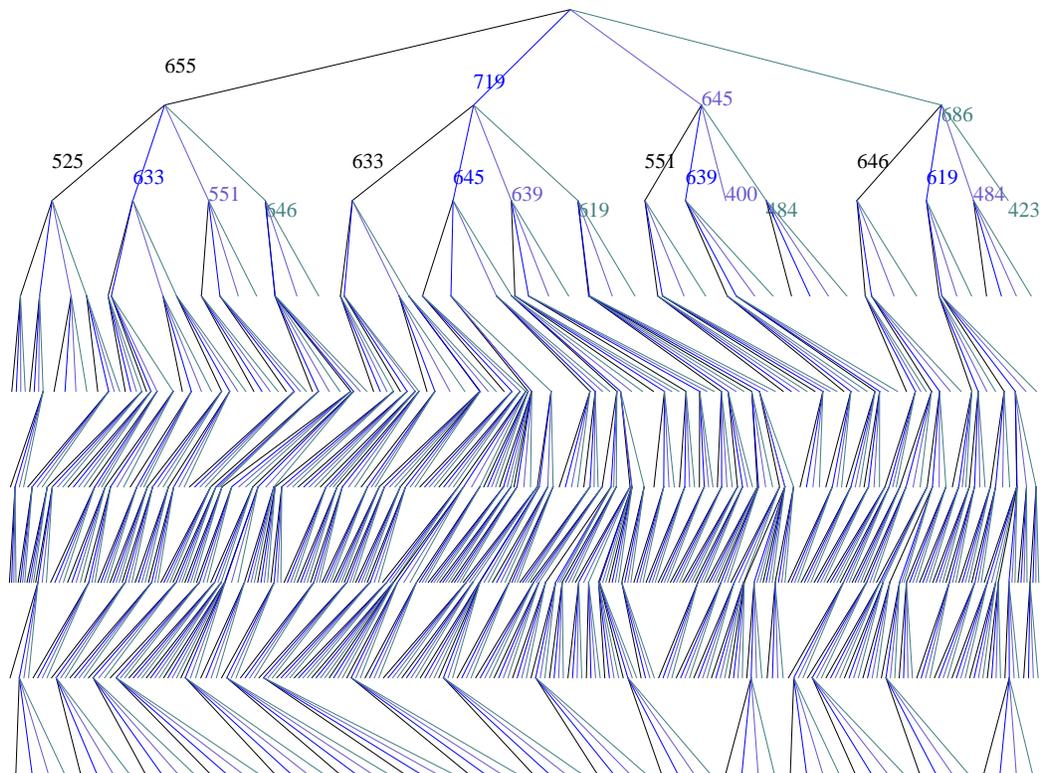


FIGURE 174: Decision tree for c499

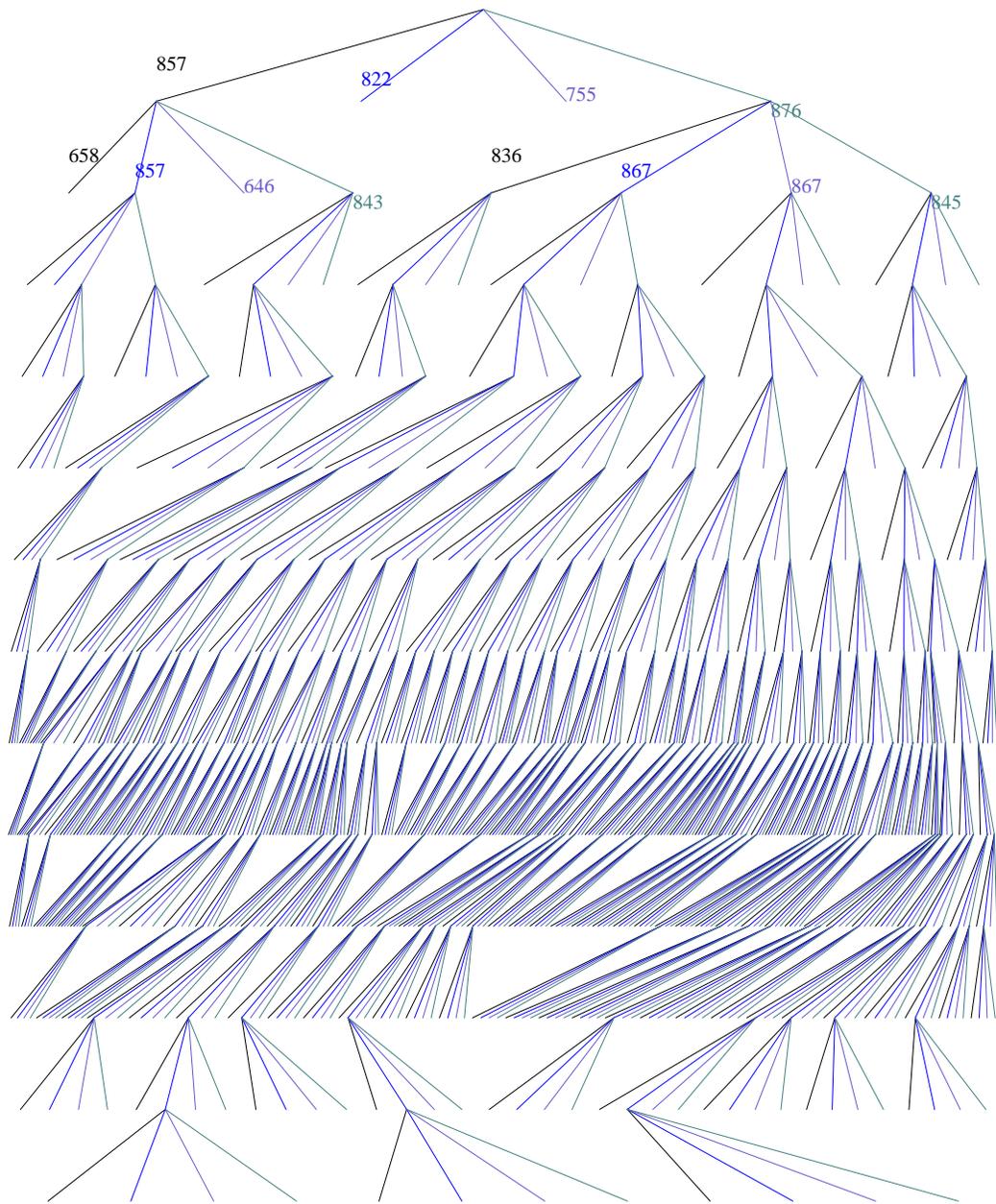


FIGURE 175: Decision tree for c432

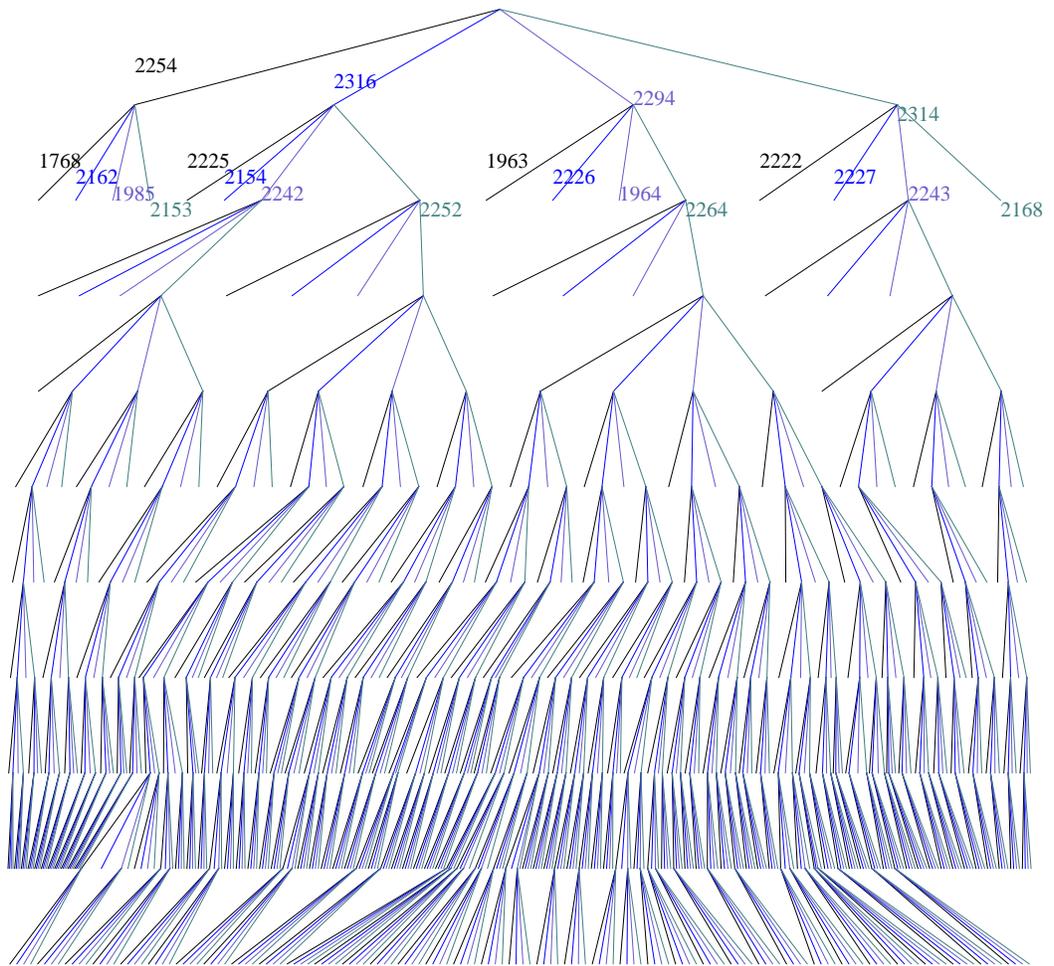


FIGURE 176: Decision tree for c880

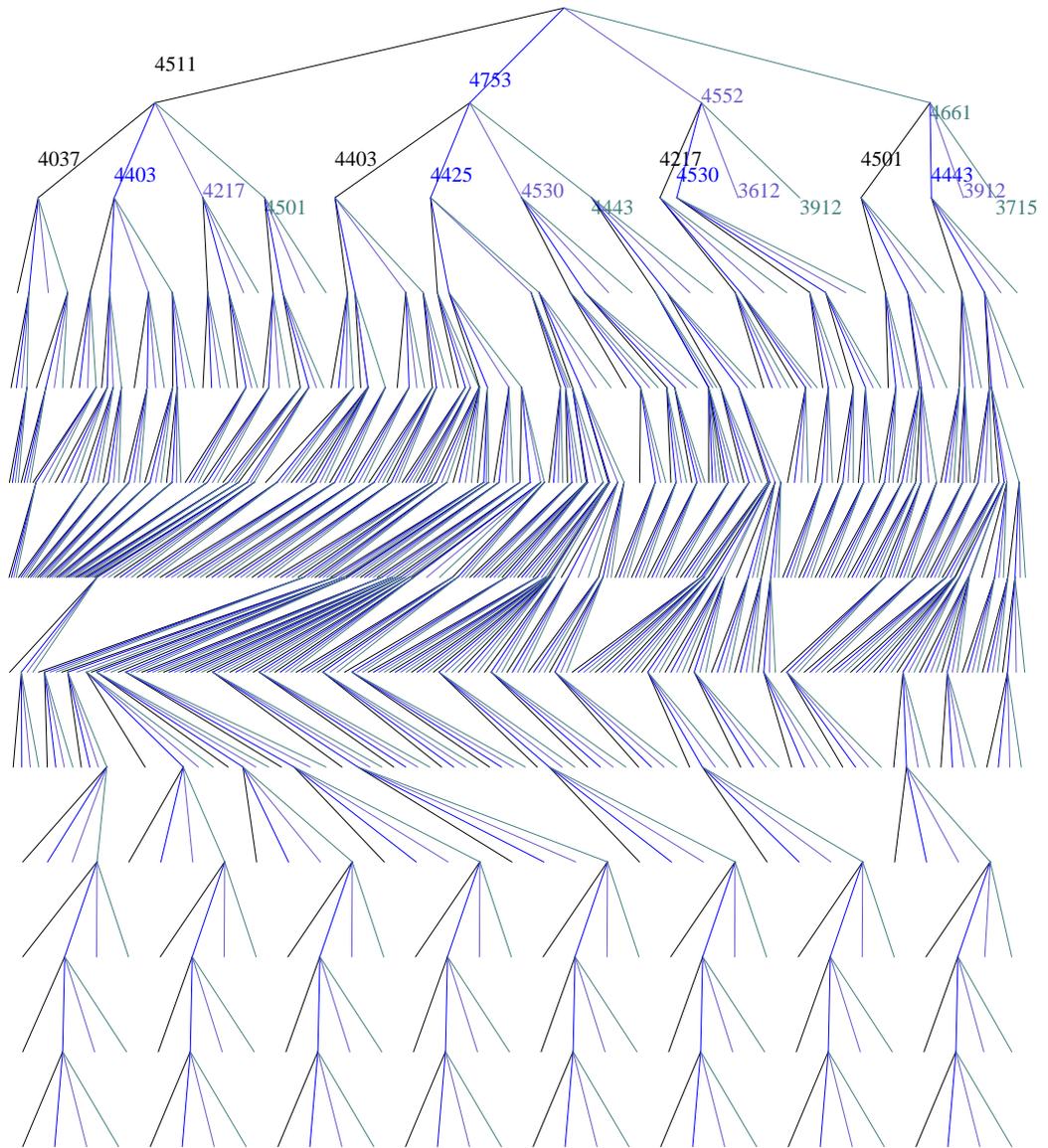


FIGURE 177: Decision tree for c1355

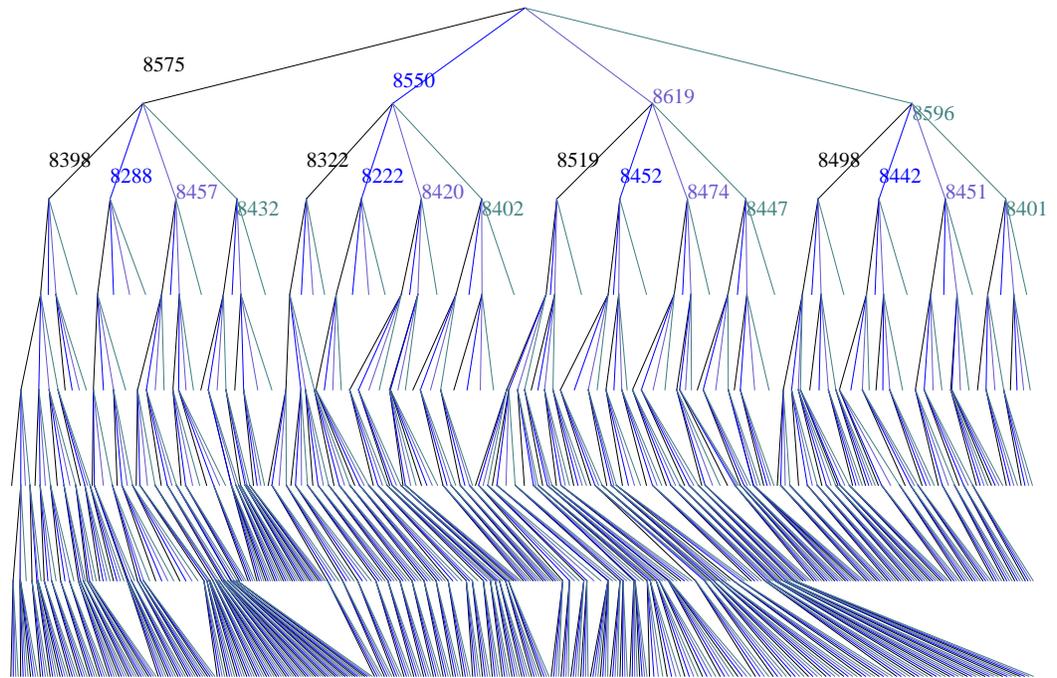


FIGURE 178: Decision tree for c1908

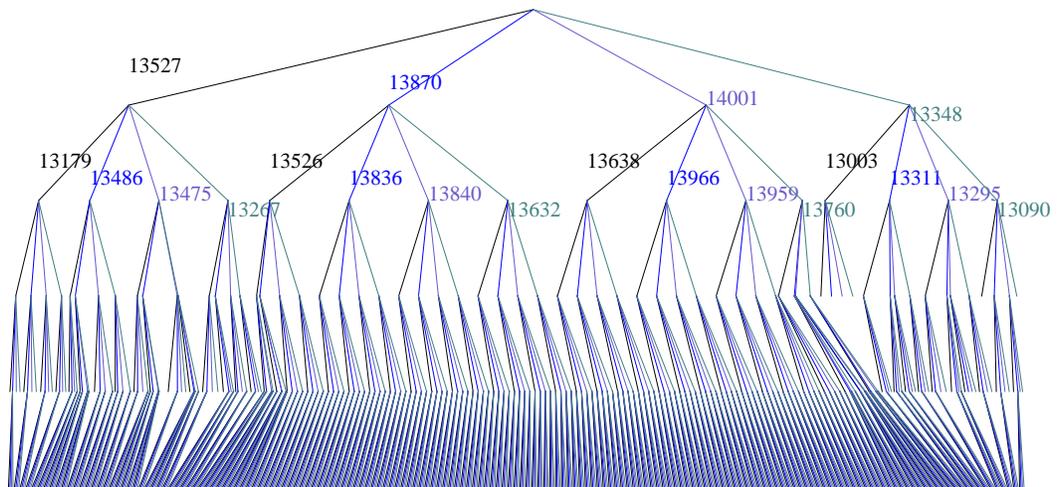


FIGURE 179: Decision tree for c3540

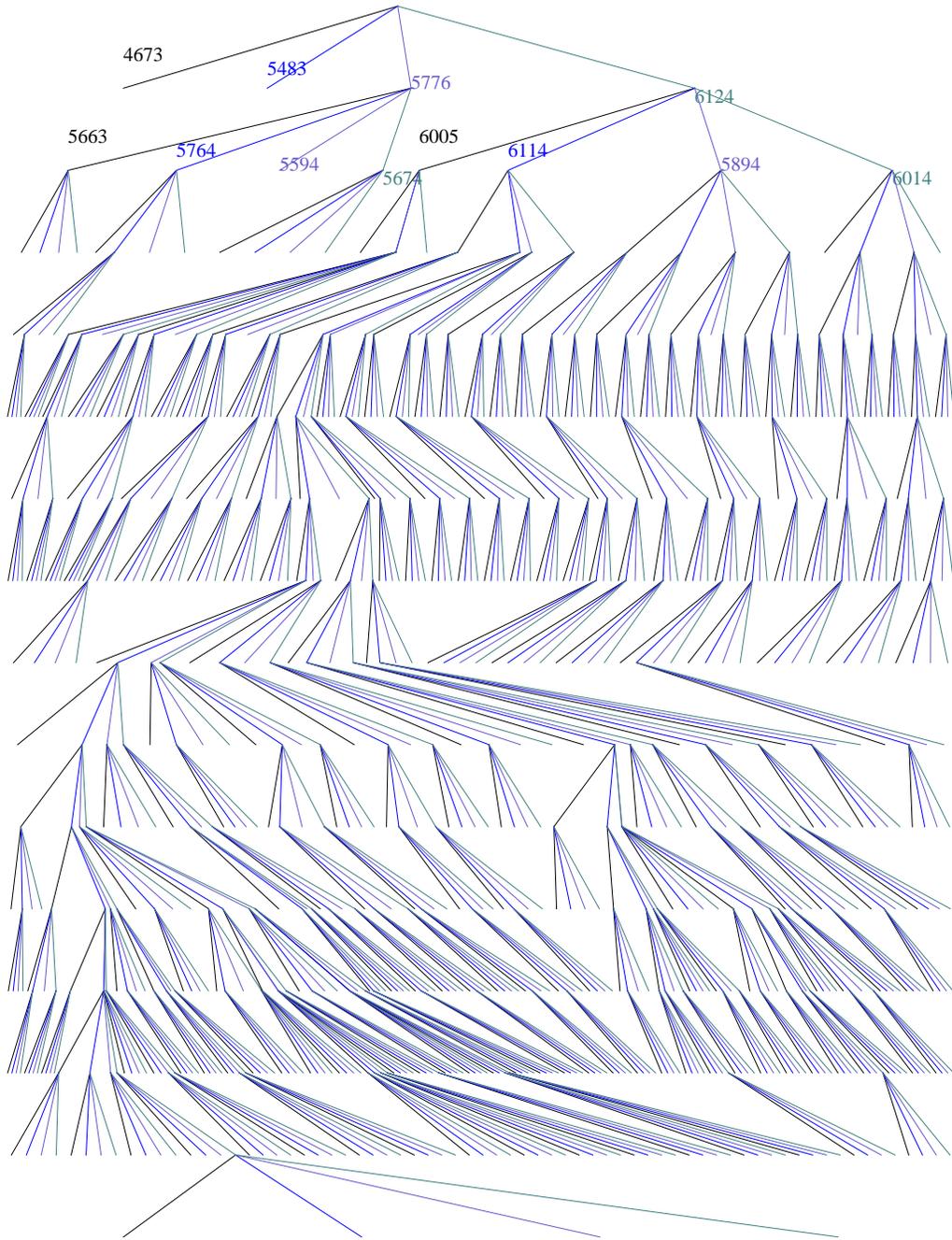


FIGURE 180: Decision tree for c2670

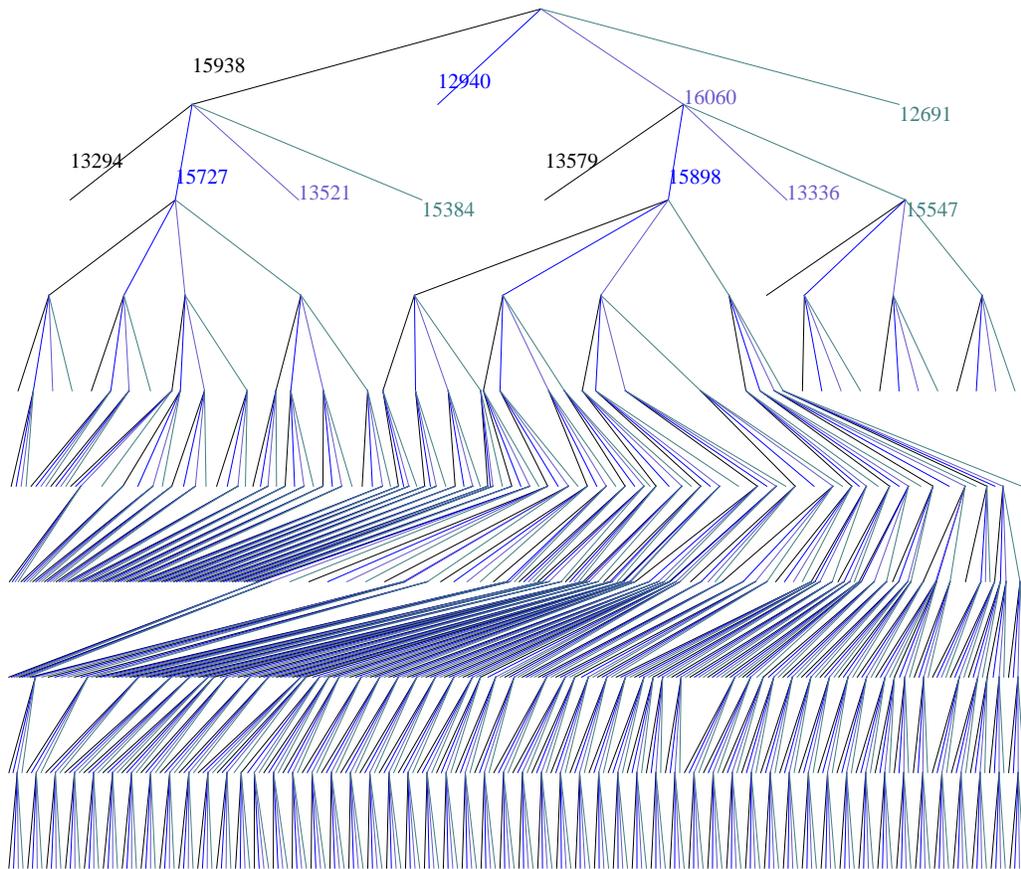


FIGURE 181: Decision tree for c5315

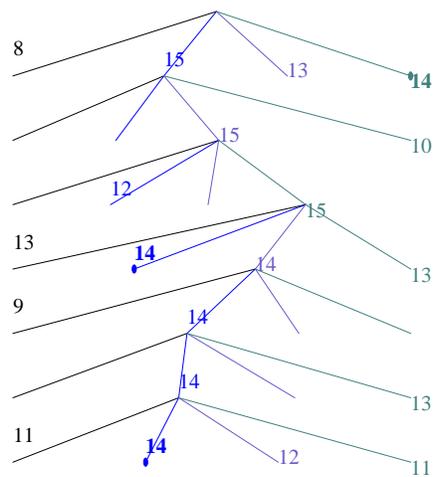


FIGURE 182: Decision tree for c17

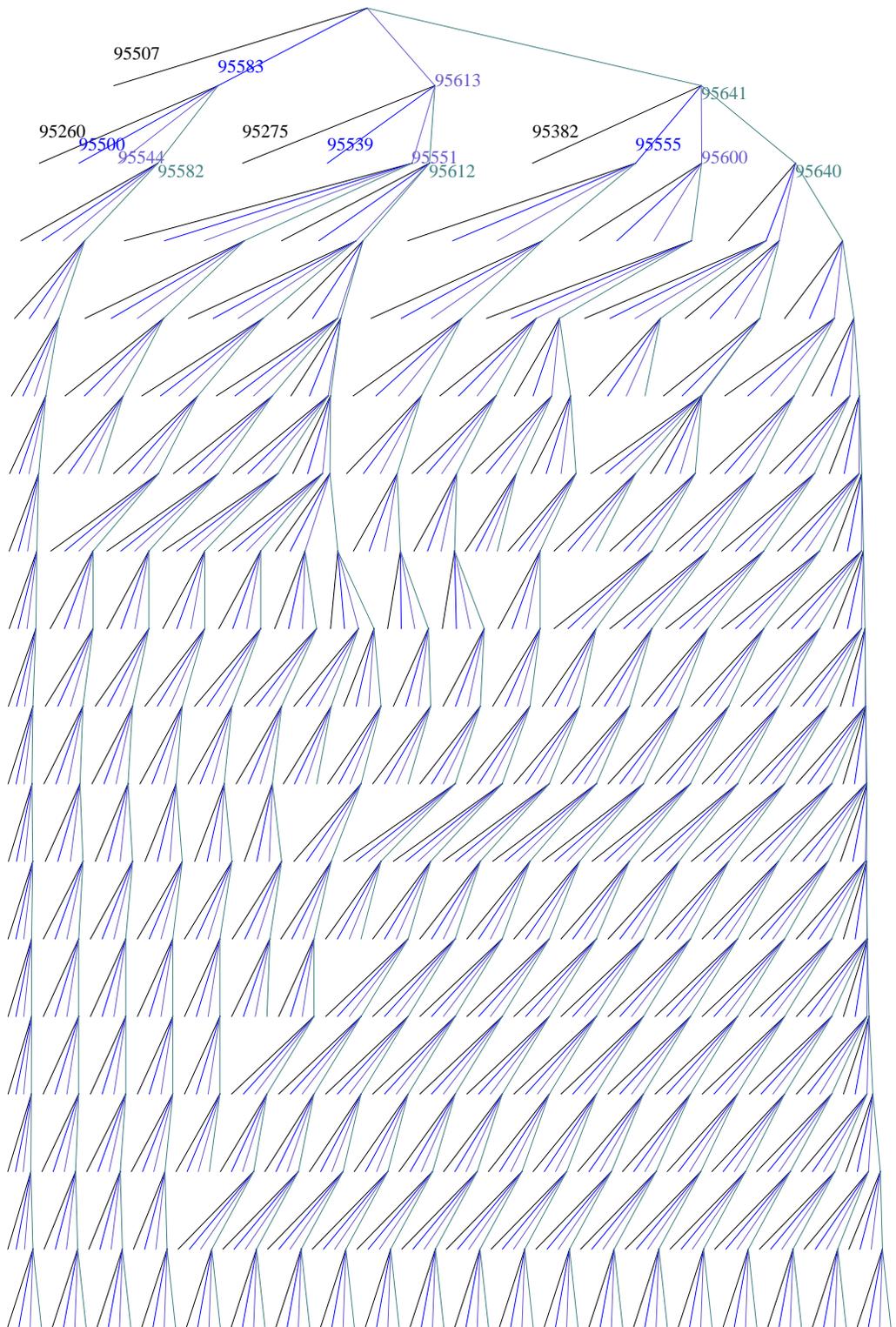


FIGURE 183: Decision tree for c6288

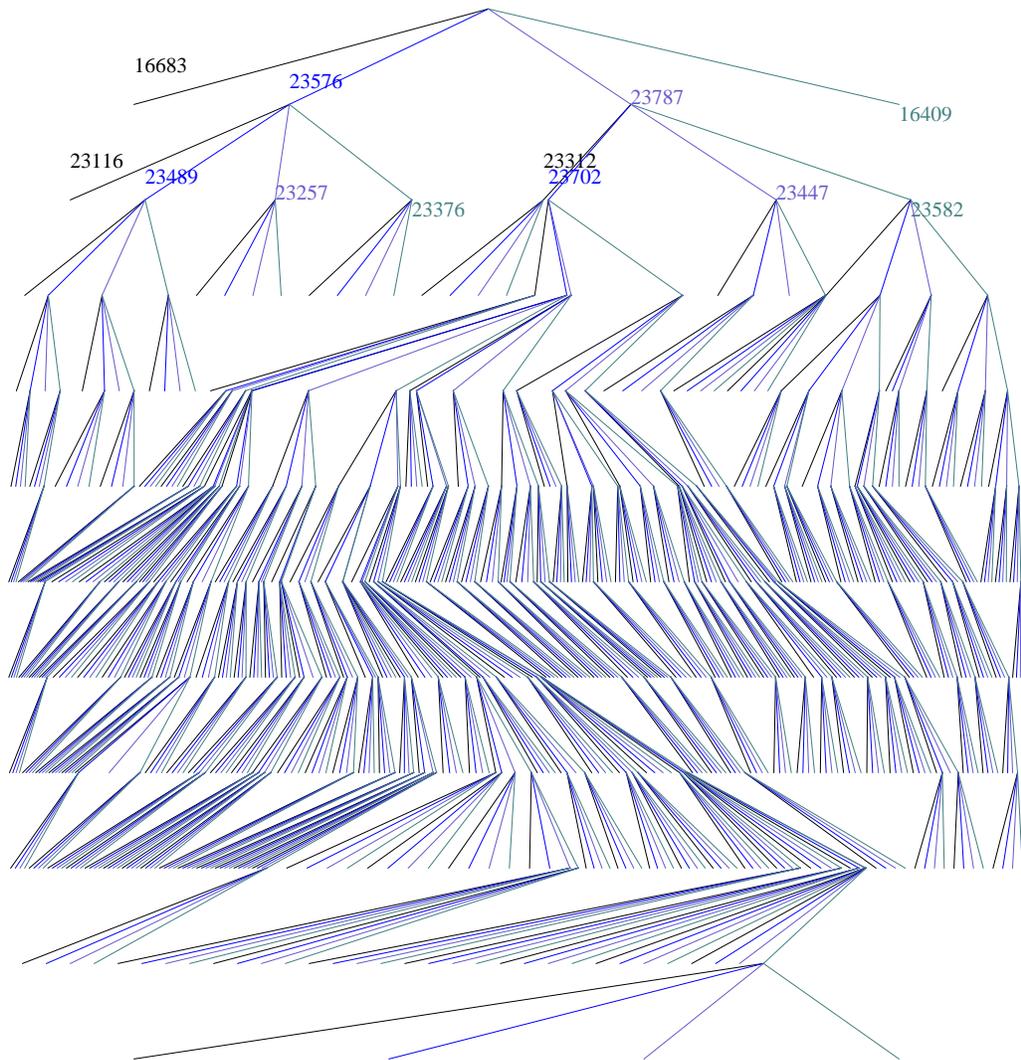


FIGURE 184: Decision tree for c7552

Acknowledgments

In the first place I would like to thank to my research director professor Eduard Černý for his many advices, analysis and deep discussions. I also appreciated his ever encouraging words and support when the things were not going very well. While working with him, I have enriched my knowledge by many theoretical basics of CAD and EDA technology. I would also like to thank my co-director, professor Nicolas C. Rumin for his invaluable reviews and advices, and sharing his vast knowledge in electrical engineering.

I would like to thank my friends and my family, especially my parents for their support and patiently accepting infrequent visits home while working hard on my research thousands miles from my homeland. I am thankful to Joyce for waiting patiently when I am working late, and to my brother for his support and advice in the field of mathematics and numerical analysis.

All the work would not be accomplished without the work of a number of undergraduate students to whom I would like extend my thanks: to Isabelle Matton who worked on benchmark analysis and testing, to Natalie Langlois for writing a part of the MIF export interface, and especially to Sebastien Charlands for implementing a considerable part of the reconvergent region analysis, and the parallel case analysis.

No work is easy to accomplish without a good work environment and resources. I appreciated the support of all people at the LASSO laboratory, and am thankful to the entire IRO department and University of Montreal whose resources were used throughout my research. Some workstations used for the research were loaned from the Canadian Microelectronics Corporation, and part of the project was funded by Northern Telecom, Ottawa. Last but not least, I would like to thank Synopsys, Inc., for providing resources during the last phase of writing the thesis.

Curriculum vitae

Jindrich Zejda, born in 1970 in Jihlava, a small city located in the beautiful highlands of East Moravia in the Czech Republic, pursued from 1988 to 1993 engineering studies in computer design at the Department of Computers at Czech Technical University in Prague. The final (M.Sc.) project in the field of mixed-signal analog-digital simulation was accomplished at GenRad, Ltd., in United Kingdom.

From 1994 he worked on gate-level timing and switching-activity verification as a Ph.D. student in the group of professor Eduard Černý at the Department of Computer Science and Operational Research (IRO) at the University of Montreal in Canada. The goal of the project was to investigate possible use of a new verification method based on constraint system resolution for static verification of dynamic power in CMOS sequential circuits.

In June 1997 he joined the static timing analysis group of Synopsys, Inc., in Mountain View, California as a senior R&D engineer. The group develops the stand-alone static timing verification tool PrimeTime and provides timing solutions and core algorithms to several Synopsys products including ChipArchitect.

His professional interest include electronic design automation (EDA), digital electronics, computer architectures, UNIX, microprocessors.

