

Functional Design Errors in Digital Circuits

Lecture Notes in Electrical Engineering

Volume 32

For other titles published in this series, go to
www.springer.com/series/7818

Kai-hui Chang · Igor L. Markov · Valeria Bertacco

Functional Design Errors in Digital Circuits

Diagnosis, Correction and Repair

 Springer

Dr. Kai-hui Chang
University of Michigan
Dept. Electrical Engineering &
Computer Science
Ann Arbor MI 48109-2122
USA
changkh@umich.edu

Dr. Valeria Bertacco
University of Michigan
Dept. Electrical Engineering &
Computer Science
Ann Arbor MI 48109-2122
USA
valeria@umich.edu

Dr. Igor L. Markov
University of Michigan
Dept. Electrical Engineering &
Computer Science
Ann Arbor MI 48109-2122
USA
imarkov@umich.edu

ISBN: 978-1-4020-9364-7

e-ISBN: 978-1-4020-9365-4

DOI 10.1007/978-1-4020-9365-4

Library of Congress Control Number: 2008937577

© Springer Science+Business Media B.V. 2009

No part of this work may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, microfilming, recording or otherwise, without written permission from the Publisher, with the exception of any material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work.

Printed on acid-free paper

9 8 7 6 5 4 3 2 1

springer.com

*To the synergy between
science and engineering*

Contents

Dedication	v
List of Figures	xiii
List of Tables	xxi
Preface	xxiii

Part I Background and Prior Art

1. INTRODUCTION	3
1.1 Design Trends and Challenges	4
1.2 State of the Art	6
1.3 New Opportunities	8
1.4 Key Innovations and Book Outline	10
2. CURRENT LANDSCAPE IN DESIGN AND VERIFICATION	13
2.1 Front-End Design	13
2.2 Back-End Logic Design	16
2.3 Back-End Physical Design	20
2.4 Post-Silicon Debugging	21
3. FINDING BUGS AND REPAIRING CIRCUITS	25
3.1 Simulation-Based Verification	25
3.1.1 Logic Simulation Algorithms	26
3.1.2 Improving Test Generation and Verification	27
3.2 Formal Verification	28
3.2.1 The Boolean Satisfiability Problem	28
3.2.2 Bounded Model Checking	29
3.2.3 Symbolic Simulation	29

3.2.4	Reachability Analysis	30
3.2.5	Equivalence Checking	31
3.3	Design for Debugging and Post-Silicon Metal Fix	31
3.3.1	Scan Chains	31
3.3.2	Post-Silicon Metal Fix via Focused Ion Beam	32

Part II FogClear Methodologies and Theoretical Advances in Error Repair

4.	CIRCUIT DESIGN AND VERIFICATION METHODOLOGIES	37
4.1	Front-End Design	37
4.2	Back-End Logic Design	37
4.3	Back-End Physical Design	38
4.4	Post-Silicon Debugging	40
5.	COUNTEREXAMPLE-GUIDED ERROR-REPAIR FRAMEWORK	43
5.1	Background	43
5.1.1	Bit Signatures	43
5.1.2	Don't-Cares	44
5.1.3	SAT-Based Error Diagnosis	44
5.1.4	Error Model	45
5.2	Error-Correction Framework for Combinational Circuits	46
5.2.1	The CoRé Framework	46
5.2.2	Analysis of CoRé	48
5.2.3	Discussions	49
5.2.4	Applications	49
6.	SIGNATURE-BASED RESYNTHESIS TECHNIQUES	51
6.1	Pairs of Bits to be Distinguished (PBDs)	51
6.1.1	PBDs and Distinguishing Power	51
6.1.2	Related Work	52
6.2	Resynthesis Using Distinguishing-Power Search	53
6.2.1	Absolute Distinguishing Power of a Signature	53
6.2.2	Distinguishing-Power Search	53
6.3	Resynthesis Using Goal-Directed Search	54
7.	SYMMETRY-BASED REWIRING	57
7.1	Background	58
7.1.1	Symmetries in Boolean Functions	58

7.1.2	Semantic and Syntactic Symmetry Detection	60
7.1.3	Graph-Automorphism Algorithms	62
7.1.4	Post-Placement Rewiring	62
7.2	Exhaustive Search for Functional Symmetries	63
7.2.1	Problem Mapping	63
7.2.2	Proof of Correctness	64
7.2.3	Generating Symmetries from Symmetry Generators	66
7.2.4	Discussion	66
7.3	Post-Placement Rewiring	67
7.3.1	Permutative Rewiring	67
7.3.2	Implementation Insights	68
7.3.3	Discussion	69
7.4	Experimental Results	69
7.4.1	Symmetries Detected	70
7.4.2	Rewiring	71
7.5	Summary	73

Part III FogClear Components

8.	BUG TRACE MINIMIZATION	77
8.1	Background and Previous Work	77
8.1.1	Anatomy of a Bug Trace	77
8.1.2	Known Techniques in Hardware Verification	79
8.1.3	Techniques in Software Verification	81
8.2	Analysis of Bug Traces	81
8.2.1	Making Traces Shorter	82
8.2.2	Making Traces Simpler	83
8.3	Proposed Techniques	84
8.3.1	Single-Cycle Elimination	84
8.3.2	Input-Event Elimination	86
8.3.3	Alternative Path to Bug	86
8.3.4	State Skip	86
8.3.5	Essential Variable Identification	87
8.3.6	BMC-Based Refinement	88
8.4	Implementation Insights	90
8.4.1	System Architecture	90

8.4.2	Algorithmic Analysis and Performance Optimizations	90
8.4.3	Use Model	92
8.5	Experimental Results	93
8.5.1	Simulation-Based Experiments	93
8.5.2	Performance Analysis	97
8.5.3	Essential Variable Identification	98
8.5.4	Generation of High-Coverage Traces	99
8.5.5	BMC-Based Experiments	100
8.5.6	Evaluation of Experimental Results	101
8.6	Summary	102
9.	FUNCTIONAL ERROR DIAGNOSIS AND CORRECTION	105
9.1	Gate-Level Error Repair for Sequential Circuits	105
9.2	Register-Transfer-Level Error Repair	106
9.2.1	Background	107
9.2.2	RTL Error Diagnosis	108
9.2.3	RTL Error Correction	114
9.3	Experimental Results	117
9.3.1	Gate-Level Error Repair	117
9.3.2	RTL Error Repair	122
9.4	Summary	130
10.	INCREMENTAL VERIFICATION FOR PHYSICAL SYNTHESIS	133
10.1	Background	133
10.1.1	The Current Physical Synthesis Flow	133
10.1.2	Retiming	134
10.2	Incremental Verification	135
10.2.1	New Metric: Similarity Factor	135
10.2.2	Verification of Retiming	136
10.2.3	Overall Verification Methodology	137
10.3	Experimental Results	140
10.3.1	Verification of Combinational Optimizations	140
10.3.2	Sequential Verification of Retiming	144
10.4	Summary	145
11.	POST-SILICON DEBUGGING AND LAYOUT REPAIR	147
11.1	Physical Safeness and Logical Soundness	148
11.1.1	Physically Safe Techniques	148
11.1.2	Physically Unsafe Techniques	149

11.2	New Resynthesis Technique – SafeResynth	151
11.2.1	Terminology	152
11.2.2	SafeResynth Framework	152
11.2.3	Search-Space Pruning Techniques	152
11.3	Physically-Aware Functional Error Repair	155
11.3.1	The PAFER Framework	155
11.3.2	The PARSyn Algorithm	156
11.4	Automating Electrical Error Repair	158
11.4.1	The SymWire Rewiring Technique	159
11.4.2	Adapting SafeResynth to Perform Metal Fix	159
11.4.3	Case Studies	160
11.5	Experimental Results	161
11.5.1	Functional Error Repair	162
11.5.2	Electrical Error Repair	164
11.6	Summary	165
12.	METHODOLOGIES FOR SPARE-CELL INSERTION	167
12.1	Existing Spare-Cell Insertion Methods	168
12.2	Cell Type Analysis	170
12.2.1	The SimSynth Technique	170
12.2.2	Experimental Setup	172
12.2.3	Empirical Results	173
12.2.4	Discussion	173
12.3	Placement Analysis	174
12.4	Our Methodology	176
12.5	Experimental Results	177
12.5.1	Cell-Type Selection	177
12.5.2	Spare-Cell Placement	178
12.6	Summary	181
13.	CONCLUSIONS	183
	References	187
	Index	199

List of Figures

1.1	Relative delay due to gate and interconnect at different technology nodes. Delay due to interconnect becomes larger than the gate delay at the 90 nm technology node.	5
1.2	Estimated mask costs at different technology nodes. Source: ITRS'05 [153].	6
2.1	The current front-end design flow.	14
2.2	The current back-end logic design flow.	17
2.3	The current back-end physical design flow.	20
2.4	The current post-silicon debugging flow.	21
2.5	Post-silicon error-repair example. (a) The original buggy layout with a weak driver (INV). (b) A traditional resynthesis technique finds a “simple” fix that sizes up the driving gate, but it requires expensive remanufacturing of the silicon die to change the transistors. (c) Our physically-aware techniques find a more “complex” fix using symmetry-based rewiring, and the fix can be implemented simply with a metal fix and has smaller physical impact.	23
3.1	Lewis’ event-driven simulation algorithm.	27
3.2	Pseudo-code for bounded model checking.	29
3.3	The algorithmic flow of reachability analysis.	30

3.4	Schematic showing the process to connect to a lower-level wire through an upper-level wire: (a) a large hole is milled through the upper level; (b) the hole is filled with SiO_2 ; (c) a smaller hole is milled to the lower-level wire; and (d) the hole is filled with new metal. In the figure, whitespace is filled with SiO_2 , and the dark blocks are metal wires.	32
4.1	The FogClear front-end design flow.	38
4.2	The FogClear back-end logic design flow.	39
4.3	The FogClear back-end physical design flow.	40
4.4	The FogClear post-silicon debugging flow.	41
5.1	Error diagnosis. In (a) a multiplexer is added to model the correction of an error, while (b) shows the error cardinality constraints that limit the number of asserted select lines to N .	45
5.2	Errors modeled by Abadir et al. [1].	46
5.3	The algorithmic flow of CoRé.	47
5.4	Execution example of CoRé. Signatures are shown above the wires, where underlined bits correspond to <i>error-sensitizing vectors</i> . (1) The gate was meant to be AND but is erroneously an OR. Error diagnosis finds that the output of the 2nd pattern should be 0 instead of 1; (2) the first resynthesized netlist fixes the 2nd pattern, but fails further verification (the output of the 3rd pattern should be 1); (3) the counterexample from step 2 refines the signatures, and a resynthesized netlist that fixes all the test patterns is found.	48
6.1	The truth table on the right is constructed from the signatures on the left. Signature s_t is the target signature, while signatures s_1 to s_4 are candidate signatures. The minimized truth table suggests that s_t can be resynthesized by an INVERTER with its input set to s_1 .	55
6.2	Given a constraint imposed on a gate's output and the gate type, this table calculates the constraint of the gate's inputs. The output constraints are given in the first row, the gate types are given in the first column, and their intersection is the input constraint. "S.C." means "signature complemented."	56
6.3	The GDS algorithm.	56

7.1	Rewiring examples: (a) multiple inputs and outputs are rewired simultaneously using pin-permutation symmetry, (b) inputs to a multiplexer are rewired by inverting one of the select signals. Bold lines represent changes made in the circuit.	58
7.2	Representing the 2-input XOR function by (a) the truth table, (b) the full graph, and (c) the simplified graph for faster symmetry detection.	64
7.3	Our symmetry generation algorithm.	67
7.4	Rewiring opportunities for p and q cannot be detected by only considering the subcircuit shown in this figure. To rewire p and q , a subcircuit with p and q as inputs must be extracted.	68
7.5	Flow chart of our symmetry detection and rewiring experiments.	70
8.1	An illustration of two types of bugs, based on whether one or many states expose a given bug. The x-axis represents FSM-X and the y-axis represents FSM-Y. A specific bug configuration contains only one state, while a general bug configuration contains many states.	78
8.2	A bug trace example. The boxes represent input variable assignments to the circuit at each cycle, shaded boxes represent input events. This trace has three cycles, four input events and twelve input variable assignments.	79
8.3	Another view of a bug trace. Three bug states are shown. Formal methods often find the minimal length bug trace, while semi-formal and constrained-random techniques often generate longer traces.	79
8.4	A bug trace may contain sequential loops, which can be eliminated to obtain an equivalent but more compact trace.	82
8.5	Arrow 1 shows a shortcut between two states on the bug trace. Arrows marked “2” show paths to easier-to-reach bug states in the same bug configuration (that violate the same property).	83
8.6	Single-cycle elimination attempts to remove individual trace cycles, generating reduced traces which still expose the bug. This example shows a reduced trace where cycle 1 has been removed.	85

8.7	Input-event elimination removes pairs of events. In the example, the input events on signal c at cycle 1 and 2 are removed.	86
8.8	Alternative path to bug: the variant trace at the bottom hits the bug at step t_2 . The new trace replaces the old one, and simulation is stopped.	87
8.9	State skip: if state $s_{j_2} = s_{i_4}$, cycles t_3 and t_4 can be removed, obtaining a new trace which includes the sequence "... $s_{j_1}, s_{j_2}, s_{i_5}, \dots$ ".	87
8.10	BMC-based shortcut detection algorithm.	88
8.11	BMC-based refinement finds a shortcut between states S_1 and S_4 , reducing the overall trace length by one cycle.	89
8.12	A shortest-path algorithm is used to find the shortest sequence from the initial state to the bug state. The edges are labeled by the number of cycles needed to go from the source vertex to the sink. The shortest path from states 0 to 4 in the figure uses 2 cycles.	90
8.13	Butramin system architecture.	91
8.14	Early exit. If the current state s_{j_2} matches a state s_{i_2} from the original trace, we can guarantee that the bug will eventually be hit. Therefore, simulation can be terminated earlier.	92
8.15	Percentage of cycles removed using different simulation-based techniques. For benchmarks like B15 and ICU, state skip is the most effective technique because they contain small numbers of state variables and state repetition is more likely to occur. For large benchmarks with long traces like FPU and picoJava, cycle elimination is the most effective technique.	95
8.16	Number of input events eliminated with simulation-based techniques. The distributions are similar to cycle elimination because removing cycles also removes input events. However, input-event elimination works the most effectively for some benchmarks like S38584 and DES, showing that some redundant input events can only be removed by this technique.	96
8.17	Comparison of Butramin's impact when applied to traces generated in three different modes. The graph shows the fraction of cycles and input events eliminated and the average runtime.	99

9.1	Sequential signature construction example. The signature of a node is built by concatenating the simulated values of each cycle for all the bug traces. In this example, trace1 is 4 cycles and trace2 is 3 cycles long. The final signature is then 0110101.	106
9.2	REDIR framework. Inputs to the tool are an RTL design (which includes one or more errors), test vectors exposing the bug(s), and correct output responses for those vectors obtained from high-level simulation. Outputs of the tool include REDIR <i>symptom core</i> (a minimum cardinality set of RTL signals which need to be modified in order to correct the design), as well as suggestions to fix the errors.	107
9.3	An RTL error-modeling code example: module half_adder shows the original code, where <i>c</i> is erroneously driven by “ <i>a b</i> ” instead of “ <i>a & b</i> ”; and module half_adder_MUX_enriched shows the MUX-enriched version. The differences are marked in boldface.	110
9.4	Procedure to insert a conditional assignment for a signal in an RTL description for error modeling.	110
9.5	Procedure to perform error diagnosis using synthesis and circuit unrolling. PI/PO means primary inputs and primary outputs.	110
9.6	Procedure to perform error diagnosis using symbolic simulation. The boldfaced variables are symbolic variables or expressions, while all others are scalar values.	112
9.7	Design for the example. Wire g1 should be driven by “r1 & r2”, but it is erroneously driven by “r1 r2”. The changes made during MUX-enrichment are marked in boldface.	113
9.8	Signature-construction example. Simulation values of variables created from the same RTL variable at all cycles should be concatenated for error correction.	116
10.1	The current post-layout optimization flow. Verification is performed after the layout has undergone a large number of optimizations, which makes debugging difficult.	134
10.2	Similarity factor example. Note that the signatures in the fanout cone of the corrupted signal are different.	136

10.3	Resynthesis examples: (a) the gates in the rectangle are resynthesized correctly, and their signatures may be different from the original netlist; (b) an error is introduced during resynthesis, and the signatures in the output cone of the resynthesized region are also different, causing a significant drop in similarity factor.	136
10.4	A retiming example: (a) is the original circuit, and (b) is its retimed version. The tables above the wires show their signatures, where the n th row is for the n th cycle. Four traces are used to generate the signatures, producing four bits per signature. Registers are represented by black rectangles, and their initial states are 0. As wire w shows, retiming may change the cycle that signatures appear, but it does not change the signatures (signatures shown in boldface are identical).	138
10.5	Circuits in Figure 10.4 unrolled three times. The cycle in which a signal appears is denoted using subscript “@”. Retiming affects gates in the first and the last cycles (<i>marked in dark gray</i>), while the rest of the gates are structurally identical (<i>marked in light gray</i>). Therefore, only the signatures of the dark-gray gates will be different.	139
10.6	Our InVerS verification methodology. It monitors every layout modification to identify potential errors and calls equivalence checking when necessary. Our functional error repair techniques can be used to correct the errors when verification fails.	139
10.7	The relationship between cell count and the difference factor. The linear regression lines of the datapoints are also shown.	142
10.8	The relationship between the number of levels of logic and the difference factor in benchmark DES _{perf} . The x-axis is the level of logic that the circuit is modified. The logarithmic regression line for the error-injection tests is also shown.	143
11.1	Several distinct physical synthesis techniques. Newly-introduced overlaps are removed by legalizers after the optimization phase has completed.	150
11.2	The SafeResynth framework.	153

11.3	A restructuring example. Input vectors to the circuit are shown on the left. Each wire is annotated with its signature computed by simulation on those test vectors. We seek to compute signal w_1 by a different gate, e.g., in terms of signals w_2 and w_3 . Two such restructuring options (with new gates) are shown as g_{n1} and g_{n2} . Since g_{n1} produces the required signature, equivalence checking is performed between w_{n1} and w_1 to verify the correctness of this restructuring. Another option, g_{n2} , is abandoned because it fails our compatibility test.	153
11.4	Conditions to determine compatibility: $wire_t$ is the target wire, and $wire_1$ is the potential new input of the resynthesized gate.	154
11.5	The pruned_search algorithm.	154
11.6	Algorithm for function get_potential_wires. XOR and XNOR are considered separately because the required signature can be calculated uniquely from $wire_t$ and $wire_1$.	155
11.7	The algorithmic flow of the PAFER framework.	156
11.8	The PARSyn algorithm.	157
11.9	The exhaustiveSearch function.	158
11.10	The SymWire algorithm.	159
11.11	Case studies: (a) g_1 has insufficient driving strength, and SafeResynth uses a new cell, g_{new} , to drive a fraction of g_1 's fanouts; (b) SymWire reduces coupling between parallel long wires by changing their connections using symmetries, which also changes metal layers and can alleviate the antenna effect (electric charge accumulated in partially-connected wire segments during the manufacturing process).	161
12.1	A design where an XOR gate must be replaced by a NAND using spare cells. (a) A high-quality fix with little perturbation of the layout. (b) A low-quality fix that requiring long wires due to poor spare-cell placement. (c) Another low-quality fix using several cells due to a poor selection of cell types.	167
12.2	The SimSynth algorithm.	170
12.3	SimSynth example using a full adder.	172
12.4	Using single gates of different types to generate desired signals. The success rates are shown in percent.	174

12.5	Illustration of different placement methods. Dark cells are spare cells. PostSpare inserts spare cells after design placement. Since design cells may be clustered in some regions, spare-cell distribution is typically non-uniform. ClusterSpare inserts spare-cell islands on a uniform grid before design placement, while UniSpare inserts single spare cells.	175
12.6	Our spare-cell insertion flow.	176
12.7	Delay and wirelength increase <i>after</i> metal fix when using three different sets of spare-cell selections. Ours has 23 and 4% smaller delay increase compared to Yee and Giles, while the wirelength increase is approximately the same.	178
12.8	Impact of spare-cell placement methods on circuit parameters: (a) before metal fix; (b)(c) after metal fix. Ours has 24% smaller delay increase before metal fix compared with ClusterSpare. The delay increase after metal fix is 37 and 17% better than the PostSpare and ClusterSpare methods, respectively.	179
12.9	Average numbers of cells used when fixing bugs in the benchmarks. By contrasting with Figure 12.4 we show that SimSynth can help determine spare-cell density. For example, Alpha has smaller success rate in Figure 12.4 than its EX block, followed by its ID and IF blocks. This figure shows that the Alpha design requires more cells than its EX, ID and IF blocks.	181

List of Tables

2.1	Distribution of design errors (in %) in seven microprocessor projects.	15
2.2	A comparison of gate-level error diagnosis and correction techniques.	19
7.1	A comparison of different symmetry-detection methods.	61
7.2	Number of symmetries found in benchmark circuits.	71
7.3	Wirelength reduction and runtime comparisons between rewiring, detailed placement and global placement.	72
7.4	The impact of rewiring before and after detailed placement.	72
7.5	The impact of the number of inputs allowed in symmetry detection on performance and runtime.	73
8.1	Characteristics of benchmarks.	94
8.2	Bugs injected and assertions for trace generation.	94
8.3	Cycles and input events removed by simulation-based techniques of Butramin on traces generated by semi-formal verification.	95
8.4	Cycles and input events removed by simulation-based techniques of Butramin on traces generated by a compact-mode semi-formal verification tool.	97
8.5	Cycles and input events removed by simulation-based methods of Butramin on traces generated by constrained-random simulation.	97
8.6	Impact of the various simulation-based techniques on Butramin's runtime.	98
8.7	Essential variable assignments identified in X-mode.	99

8.8	Cycles and input events removed by simulation-based methods of Butramin on traces that violate multiple properties.	100
8.9	Cycles removed by the BMC-based method.	101
8.10	Analysis of a pure BMC-based minimization technique.	102
8.11	Analysis of the impact of a bug radius on Butramin effectiveness.	103
9.1	Error-correction experiment for combinational gate-level netlists.	119
9.2	Error-correction experiment for combinational gate-level netlists with reduced number of initial patterns.	119
9.3	Multiple error experiment for combinational gate-level netlists.	120
9.4	Error correction for combinational gate-level netlists in the context of simulation-based verification.	121
9.5	Error-repair results for sequential circuits.	122
9.6	Description of bugs in benchmarks.	124
9.7	Characteristics of benchmarks.	125
9.8	RTL synthesis-based error-diagnosis results.	126
9.9	Gate-level error-diagnosis results.	127
9.10	Error-correction results for RTL designs	129
10.1	Characteristics of benchmarks.	140
10.2	Statistics of similarity factors for different types of circuit modifications.	141
10.3	The accuracy of our incremental verification methodology.	144
10.4	Statistics of sequential similarity factors for retiming with and without errors.	145
10.5	Runtime of sequential similarity factor calculation (SSF) and sequential equivalence checking (SEC).	145
11.1	Comparison of a range of physical synthesis techniques in terms of physical safeness and optimization potential.	151
11.2	Characteristics of benchmarks.	162
11.3	Post-silicon functional error repair results.	163
11.4	Results of post-silicon electrical error repair.	165
12.1	A summary of existing spare-cell insertion techniques described in US patents. Major contributions are marked in boldface.	169
12.2	Characteristics of benchmarks	173

Preface

The dramatic increase in design complexity of modern circuits challenges our ability to verify their functional correctness. Therefore, circuits are often taped-out with functional errors, which may cause critical system failures and huge financial loss. While improvements in verification allow engineers to find more errors, fixing these errors remains a manual and challenging task, consuming valuable engineering resources that could have otherwise been used to improve verification and design quality. In this book we solve this problem by proposing innovative methods to automate the debugging process throughout the design flow. We first observe that existing verification tools often focus exclusively on error detection, without considering the effort required by error repair. Therefore, they tend to generate tremendously long bug traces, making the debugging process extremely challenging. Hence, our first innovation is a bug trace minimizer that can remove most redundant information from a trace, thus facilitating debugging. To automate the error-repair process itself, we develop a novel framework that uses simulation to abstract the functionality of the circuit, and then rely on bug traces to guide the refinement of the abstraction. To strengthen the framework, we also propose a compact abstraction encoding using simulated values. This innovation not only integrates verification and debugging but also scales much further than existing solutions. We apply this framework to fix bugs both in gate-level and register-transfer-level circuits. However, we note that this solution is not directly applicable to post-silicon debugging because of the highly-restrictive physical constraints at this design stage which allow only minimal perturbations of the silicon die. To address this challenge, we propose a set of comprehensive physically-aware algorithms to generate a range of viable netlist and layout transformations. We then select the most promising transformations according to the physical constraints. Finally, we integrate all these scalable error-repair techniques into a framework called *FogClear*. Our empirical evaluation shows that FogClear can repair errors in a broad range of designs, demonstrating its ability to greatly reduce

debugging effort, enhance design quality, and ultimately enable the design and manufacture of more reliable electronic devices.

This book is divided into three parts. In Part I we provide necessary background to understand this book and illustrate prior art. In Part II we present our FogClear methodologies and describe theoretical advances in error repair, including a counterexample-guided error-repair framework and signature-based resynthesis techniques. In Part III we explain different components used in the FogClear flow in detail, including bug trace minimization, functional error diagnosis and correction, an incremental verification system for physical synthesis, post-silicon debugging and layout repair, as well as methodologies for spare-cell insertion. Finally, we conclude this book and summarize our key techniques in the last chapter.