ENHANCED LAYOUT-AWARE CIRCUIT/SYSTEM SYNTHESIS

by

Ahmet Unutulmaz

B.S., Electrical-Electronics Engineering, Boğaziçi University, 2006

M.S., Electrical-Electronics Engineering, Boğaziçi University, 2009

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Graduate Program in Electrical-Electronics Engineering

Boğaziçi University

2015

ENHANCED LAYOUT-AWARE CIRCUIT/SYSTEM SYNTHESIS

APPROVED BY:

Prof. Günhan Dündar (Thesis Supervisor)	.....
Assist. Prof. İsmail F. Başkaya	.....
Prof. Francisco V. Fernandez	.....
Prof. Can Özturan	.....
Assist. Prof. Hamdi Torun	.....

DATE OF APPROVAL: 19.01.2015

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Günhan Dündar, Francisco Fernandez and İsmail F. Başkaya for their invaluable guidance and help during the preparation of this dissertation. I would like to mention their patience, giving me inspiration and hope when I was stuck at dead-ends.

I also would like to thank to my family for their infinite patience, support and love. Besides, I thank them very much for everything they sacrificed.

Finally, I would like to thank Turkish Scientific and Technical Research Council (TUBITAK) for its support through the project 112E005.

ABSTRACT

ENHANCED LAYOUT-AWARE CIRCUIT/SYSTEM SYNTHESIS

This thesis presents the layout description script (LDS) which is suitable to be used within a template synthesizing layout aware circuit synthesis loop. LDS is a hybrid programming language supporting both sequential and constraint programming capabilities. Through constraint programming, representations used in the layout optimizers may be converted to LDS which enables the integration of LDS with layout optimizers. The language allows to code not only static devices but also parametrized device generators. LDS is compared with available layout representations and code samples are given to highlight the superiority of LDS over sequential programming. An integrated development environment to code LDS, capable of keyword highlighting and syntax checking, is built and some screen shots are shared. A tool is coded to extract an LDS template from an expert drawn layout and sample templates and their instances are shown. A sequential convex programming based methodology is developed and used to minimize the layout area, which is shown to be a convex function of the layout width and height under realistic assumption. Thus, the area of an LDS template may be efficiently minimized which increases the flexibility of an LDS template.

ÖZET

SERİM-FARKINDALIKLI GELİŞMİŞ DEVRE/SİSTEM SENTEZİ

Bu tez serim farkındalıklı tasarım döngüsüne uyumlu serim tanımlama dili LDS'yi sunmaktadır. LDS hibrit bir programlama dilidir ve hem ardışık hem de kısıt programlamayı desteklemektedir. Kısıt programlama kapasitesi sayesinde serim yerleşimlerini tutmakta kullanılan gösterimler LDS'de gösterilebilir. Bu sayede farklı gösterimler, LDS ile birleştirilebilir. Bu gösterimleri desteklemesi sayesinde LDS bir yerleşim aracı tarafından kolaylıkla sentezlenebilir. Bu dil ile statik ve dinamik serimler kolaylıkla oluşturulabilir. LDS dili, serim programlamada kullanılan diğer dillerle karşılaştırılmıştır. Bazı kod örnekleri verilerek LDS dilinin ardışık dillere üstünlüğü gösterilmiştir. LDS dilini kodlanmasının kolaylaştırmak için anahtar kelimeleri ve hataları vurgulayan tümleşik bir kod yazım ortamı geliştirilmiştir. Tecrübeli tasarımcılar çizdikleri serimlerden şablon çıkartmak için bir araç geliştirilmiştir ve örnek şablonlar oluşturulmuştur. Analog devre serimlerinin toplam alanın serimin yüksekliğinin ve genişliğinin konveks fonksiyonları olduğu gösterilmiştir. Serim alanını minimize etmek için konveks problemlerin art arda çözülmesine dayanan bir metot geliştirilmiştir. Bu metot LDS dili ile entegre edilerek esnek şablonların kodlanabilmesine imkan sağlanmıştır.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZET v
LIST OF FIGURES ix
LIST OF TABLES xvi
LIST OF SYMBOLS xvii
LIST OF ACRONYMS/ABBREVIATIONS xviii
1. INTRODUCTION 1
2. LAYOUT SYNTHESIS 6
2.1. Related Literature 8
2.2. Syntax 10
2.3. LDS Problems 14
2.3.1. Pre-Solving 18
2.3.2. Hierarchical Solver 18
2.3.3. Finding the Absolute Positions 23
2.4. Layout Constraints 24
2.5. Data Structures 25
2.5.1. Double, Integer and Long 25
2.5.2. Term 26
2.5.3. Rectangle 30
2.5.4. Layer 32
2.5.5. Port 34
2.5.6. Rule 35
2.5.7. Net 35
2.5.8. Solver 36
2.5.9. Transformations 36
2.6. Code Generator 40
2.6.1. Lexer and Parser 40
2.6.2. Library 41

2.7. Integrated Development Environment 42
2.7.1. Project Manager 42
2.7.2. Keyword Highlighting 45
2.7.3. Syntax Checking 45
3. TEMPLATE SYNTHESIS 47
3.1. Capturing a Template 47
3.1.1. Methodology 47
3.1.2. Extracted Templates 49
3.2. Topological Representations 49
3.2.1. Slicing Tree Representation 51
3.2.2. Sequence Pair Representation 51
3.2.3. B*-Tree Representation 54
3.3. Device Generators 54
3.3.1. Guard-ring Generator 54
3.3.2. Transistor Generator 56
3.3.3. Routed Transistor Generator 57
3.3.4. Resistor 59
3.3.5. Capacitor 60
3.3.6. Template for a Folded Amplifier 63
4. AREA OPTIMIZATION 65
4.1. Related Literature 66
4.2. Convex Models for Devices 73
4.2.1. Transistor 74
4.2.2. Capacitor 76
4.2.3. Resistor 81
4.3. Convexity of a Layout 82
4.3.1. Convexity of Slicing Floorplans 83
4.3.2. Convexity of General Floorplans 99
4.4. Optimization Technique 105
4.4.1. Methodology 107
4.4.2. Convergence 109

4.4.3. Results 110
4.5. Folded-Amplifier Layout 114
5. CONCLUSION 116
REFERENCES 119

LIST OF FIGURES

Figure 1.1. Layout Synthesized through Optimization. 1

Figure 1.2. Layout Aware Circuit Synthesis Loops (a) Layout Synthesized through Optimization, (b) Parasitic Model used, (c) Template Instanced. . 3

Figure 1.3. Template Synthesized and Reused. 4

Figure 2.1. Order of Assignments (a) Placement, (b) Code to Align Right Edges as in (a), (c) Order of Assignment Changed, (d) Placement After the First Assignment in (d), (e) Placement after the second assignment in (d). 7

Figure 2.2. Symmetry with (a) Sequential Code, (b) LDS. 8

Figure 2.3. An LDS Template. 12

Figure 2.4. Constraints and Objectives. 14

Figure 2.5. LDS Codes (a) without Constraint, (b) with Constraint. (c) Relation between LDS Code, Layout, and Linear Program. 15

Figure 2.6. Problem (a) Nested Problems, (b) Sample Code with the `final` Keyword, (c) New Problem. 16

Figure 2.7. Redundant Constraints Removed (a) the Constraints, (b) the Constraint Graph. 18

Figure 2.8. Flexible Layouts, Port Shown in (a) Enlarged in (b), Spacing Rule Violation Shown in (c) Corrected in (d), Space Added to the Device in (e) as Shown in (f). 20

Figure 2.9. Hierarchical Problem (a) Constraints for *P1*, *P2* and *P3*, (b) Tables Stored in *P1*, *P2* and *P3*, (c) Constraint Graph for *P3*, (d) Constraint Graph for *P2*, (e) White Nodes in (d) Eliminated, (f) Constraint Graph for *P2* after Solving *P1*. 21

Figure 2.10. Comparison of Run Time Costs. 23

Figure 2.11. Overlap-free Layout with LDS (a) Initial Placement and Sizes, (b) Rectangles *b* and *c* Expanded, Updates for `lds(x){lds(y){...}}` Given in (c) and (d), Updates for `lds(y){lds(x){...}}` Given in (e) and (f). 24

Figure 2.12. Placement, Abutment, Alignment, and Symmetry Constraints in LDS. 25

Figure 2.13. Numerical Classes (a) Pointed Values Not Modified, (b) Explicit Casting. 26

Figure 2.14. Symbolic Calculations (a) Sample Code to Instance `x`, `y` and, `v` Typed *Term*s, (b) A Constraint is Composed of *Term*s. 28

Figure 2.15. Calculations on *Term*s Handled via Special Data Structures. . . . 29

Figure 2.16. Properties of Rectangles (a) `left` is Aware of Transformation, (b) `width` Defined in Rectangle, (c) `x.enclose` Horizontally Constrains, (d) Equivalent of Code in (c), (e) User Defined Function `center` Aligns Centers of Rectangles. 31

Figure 2.17. Instances of *Rule* Class May Have Hierarchy. 35

Figure 2.18. Transformations on Constraints (a) Constraint Defined in Rectangle *f*, (b) Rectangle *f* Rotated, Constraint Transformed, (c) Constraint Defined in Rectangle *g*, (b) Rectangle *f* Rotated, Constraint Not Transformed. 39

Figure 2.19. Abstract Syntax Tree for the Expression `x(a) + 3`. 41

Figure 2.20. The *JLDS* Project (a) Generating New *JLDS UMC130nm* Project, (b) Sample Code Automatically Added and New Menu Item added. 44

Figure 2.21. The *JLDS* Editor (a) Keywords, Strings and Comments Highlighted, (b) Syntax Errors Reported. 46

Figure 3.1. Capture Extracts the Template in Two Steps (a) Module, Port, and Connectivity Extracted via a GUI, (b) Synthesized LDS Code. 48

Figure 3.2. Capture Tool (a) Schematic of Amplifier (b) Schematic of Voltage Doubler, (c) Extracted Data for (a), and (d) Extracted Data for (b), (e) Instance of the Template for (c), and (f) Instance of the Template for (d). 50

Figure 3.3. Slicing Tree to LDS (a) Placement, (b) Slicing Tree Representation for (a), (c) Rectangles Generated, (d) Algorithm to Generate LDS Constraints for a Slicing Tree Representation. 52

Figure 3.4. Sequence Pair to LDS (a) Placement, (b) Extracted Constraints, (c) Algorithm to Generate LDS Constraints from a Sequence Pair Representation. 53

Figure 3.5. B*-Tree to LDS (a) Placement, (b) B*-Tree (Horizontal) Representation for (a), (c) Rectangles Generated and Placed for Each Node, (d) Algorithm to Constrain a Placement for B*-Tree Representation. 55

Figure 3.6. Guard-Ring (a) for *n-well* with $m = 1$, (b) for *p-substrate* with $m = 2$. Ports Shown in (a). 56

Figure 3.7. MOSFET (a) NMOS with $m = 3$, $L = 0.13\mu m$ and $W = 1.65\mu m$, (b) PMOS with Guard-Ring and $m = 3$, $L = 0.13\mu m$, $W = 0.48\mu m$, (c) PMOS with $L = \{3\mu m, 2\mu m, 1\mu m, 0.5\mu m, 0.5\mu m\}$, $W = 5\mu m$, $metalwidth = 1\mu m$. 58

Figure 3.8. MOSFET with Source-Drain-Gate Connections (a) NMOS with $width = 1\mu m$ and $layer = \{tech.PO1, tech.MET1, tech.MET1\}$, (b) PMOS with Guard-Ring and All Rows in *MET2*. 59

Figure 3.9. Resistor (a) $ns = 3$, $length = 6\mu m$ and $width = 1\mu m$ (b) $ns = 5$. . 60

Figure 3.10. Capacitor with $W = 10\mu m$ and $H = 10\mu m$, Ports *IN* and *OUT* Shown. 61

Figure 3.11. Schematic of the Folded Amplifier Circuit. 61

Figure 3.12. Instance of the Folded Amplifier Template. 62

Figure 3.13. Simulation Results (a) Transient Analysis, (b) AC Analysis. 64

Figure 4.1. Variant Selection (a) Floorplan, (b) Variants for Components in (a), (c) Possible Placement where Variant B_1 is Selected, (d) Another Possibility where Variant B_2 is Selected. 66

Figure 4.2. Area of a Transistor Changes as Number of Fingers m Changes. Each Circle Corresponds to a Different m Value for a Transistor in the UMC 0.13μm Technology with W=10μm and L=0.5μm. . . . 67

Figure 4.3. Components $B1$ and $B2$ are Symmetrically Placed with Respect to the Tick Dashed Line and the Center of the Component A Aligns with Symmetry Axis. 68

Figure 4.4. Linear Approximation (a) Height h vs. Width w, Number of Fingers m in $[5 : 20]$. Approximation for h with Three Lines Shown. (b) Area $\psi(w)$ of Transistor in (a) Plotted in Black, h Linearly Approximated with 15 Lines Plotted in Gray. 70

Figure 4.5. (a) Horizontal Addition of Two ESF where Placement of first ESF Represented by B*-Tree η_1 and Placement of Second One Represented by B*-Tree η_2. Possible Placements for η_1 Given in (b) and (c), and Placement for η_2 Given in (d). 71

Figure 4.6. Transistor Parameters Shown on Two Different Transistors (a) Simple Transistor, (b) Transistor with Guard-Ring and Routing. . . . 75

Figure 4.7. Parameters of Different Capacitor Shown (a) Diffusion Capacitor, (b) Metal-Insulator-Metal Capacitor, (c) Metal-Oxide-Metal Capacitor. 78

Figure 4.8. Parameters of a Resistor. 82

Figure 4.9. (a) Placement and Functions for Dimensions, (b) Area Plot: Obtained by Varying All ms in (a) from 1 to 5, Pareto-Front (w/h) in (b) Observed to be Convex. 83

Figure 4.10. (a) A Placement of *ami33* from *MCNC* Benchmark, (b) Pareto Frontier for the Placement in (a). 84

Figure 4.11. Layout (a) Not Compact, If Possible to Obtain Shorter Layout as in (b). Points Representing Areas of (a) and (b) Shown in (c). Filled Region in (c) Represents Area of Any Layout where Dark Solid Line is for a Compact Layout. 85

Figure 4.12. Horizontal Combination without Dead Space. Heights of the Blocks Equal, Configuration Denoted by H_E (horizontal-equal-height). . . 87

Figure 4.13. Vertical Combination without Dead Space. Widths of Blocks Equal, Configuration Denoted by V_E (vertical-equal-width). 89

Figure 4.14. Horizontal Combination with Dead Space (a) Block Reaches Minimum Height (H_{N1}), (b) Both Blocks Reach Minimum Height (H_{N2}), (c) One Block Reaches Maximum Height (H_{X1}), (d) Both Blocks Reach Maximum Height (H_{X2}). 91

Figure 4.15. Vertical Combination (a) Block Reaches Minimum Width (V_{N1}), (b) Block Reaches Maximum Width (V_{X1}), (c) Both Blocks Reach Maximum Width (V_{X2}). 93

Figure 4.16. Area as a function of w Obtained by Taking Maximum of Extended Functions (a) Horizontal Combination, Area ψ_H is the Maximum of $\tilde{\psi}_{H_{X1}}$, $\tilde{\psi}_{H_E}$, $\tilde{\psi}_{H_{N1}}$. (b) Vertical Combination, Area ψ_V is the Maximum of $\tilde{\psi}_{V_E}$, $\tilde{\psi}_{V_{X1}}$, $\tilde{\psi}_{V_{X2}}$. 94

Figure 4.17. (a) Combination Composed of Symmetric Blocks, (b) Proximity Constraints Between the Block A and its neighbors Shown. 97

Figure 4.18. (a) Basic Non-Slicing Floorplan, (b) A Realistic Case, (c) An Unrealistic Case. In (b) and (c), Gray Area Represents Overhead, The White Area Represent Flexible Regions. 99

Figure 4.19. Change in Objective Function. 108

Figure 4.20. (a) Convergence of r to $r_{opt} = 3$, (b) Percentage Error. 109

Figure 4.21. Sample Layouts (a) Placement Op-Amp , (b) Placement for ami-33 from MCNC Bench., (c) and (d) Placements for ami-49 from MCNC Bench. Flexible Modules are Filled: Transistors and Resistors in Light Gray, Capacitors in Dark Gray. 111

Figure 4.22. Op-Amp Circuit (a) Schematic, (b) Floorplan (c) Hierarchy. . . . 112

Figure 4.23. Folded Amplifier with Minimum Area (a) Instance I, (b) Instance II. Template Presented in Chapter 3.3.6 and Values of Parameter Given in (a) and (b). 115

LIST OF TABLES

Table 2.1. Comparison of LDS with Other Representations. 11

Table 2.2. Comparison of Run Times Costs for Solving Constraints for Transistor Arrays of Sizes 1x1, 2x2, 4x4, 6x6, 8x8. 22

Table 2.3. Resultant Types of the Operations Between Two Numeric Types. . 27

Table 2.4. Operations Supported by the *Term* Class. 28

Table 2.5. Methods Defined in the *Rectangle* Class. 33

Table 2.6. Vertically/Horizontally Constraining Methods. 33

Table 2.7. Functions for Transformation. 38

Table 2.8. Methods Defined in the LDS Library. 43

Table 4.1. Comparison of the Available Methods with the Method Presented in This Chapter. 69

Table 4.2. Comparison of LDS with Other Area Minimization Methods. . . . 110

LIST OF SYMBOLS

h	Height of a rectangle
h_p	Height of a capacitor plate
k_i	Geometric model parameter of a device
m	Number of fingers
r	Aspect ratio
w	Width of a rectangle
w_p	Width of a capacitor plate
C	Capacitance
L	Length of a transistor
R	Resistance
W	Width of a transistor
α	Geometric model parameter of a device
β	Geometric model parameter of a device
η	B*-Tree representation
γ_a	Capacitance per unit area
γ_p	Fringing capacitance per unit length
ψ	Area of a rectangle
Γ_+	First sequence in a sequence pair representation
Γ_-	Second sequence in a sequence pair representation

LIST OF ACRONYMS/ABBREVIATIONS

AST	Abstract Syntax Tree
CAD	Computer Aided Design
DSL	Domain Specific Language
EDA	Electronic Design Automation
ESF	Enhanced Shape Function
GP	Geometric Programming
GUI	Guided User Interface
IDE	Integrated Development Environment
LDS	Layout Description Script
LP	Linear Programming
MIM	Metal Insulator Metal
MOM	Metal Oxide Metal
SA	Simulated Annealing
SCP	Sequential Convex Programming
SF	Shape Function
SLP	Sequential Linear Programming
SP	Sequence Pair Representation

1. INTRODUCTION

Nowadays, most ICs contain both digital and analog circuitry. Synthesis or re-mapping of digital blocks to new technologies is almost fully automated. However, most analog blocks are manually designed and require much higher development time than digital blocks. Today's trend is to replace analog processing by digital computations, but some functions must necessarily remain analog and require numerous iterations between different design levels [1]. Reducing these iterations has been suggested by the International Technology Roadmap for Semiconductors as a major contribution towards the reduction of design cost. Towards this end, the integration of physical (layout) and electrical (circuit) synthesis in one single step has been proposed, yielding the so-called layout-aware circuit synthesis approaches.

Most successful analog circuit synthesis approaches are based on the formulation of the sizing problem as an optimization problem, commonly solved by iterative processes that imply hundreds or thousands of circuit performance evaluations. It becomes obvious that integration of physical synthesis into such circuit synthesis process is only practical if the circuit layout can be instanced very fast.

Many approaches have been proposed to automate analog circuit and layout synthesis. According to the methodology followed to combine circuit and layout synthesis,

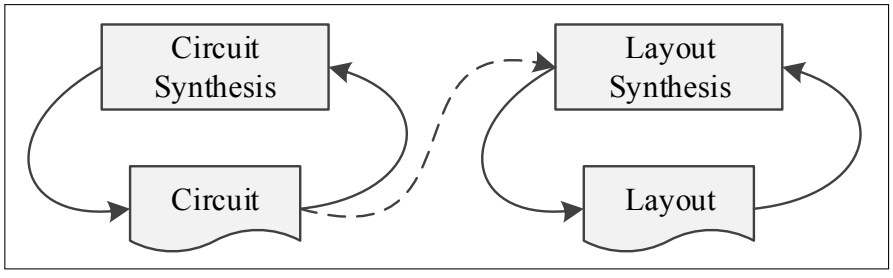


Figure 1.1. Layout Synthesized through Optimization.

these tools may be classified in three categories, namely: *traditional synthesis* [2, 3], *performance-driven layout synthesis* [4, 5] and the *layout-aware circuit synthesis* [6–14].

In traditional approaches, circuit synthesis is not connected to the layout synthesis quantitatively. During layout synthesis, matching and coupling are qualitatively considered [3], but performance degradation due to the parasitic effects, such as: resistance or capacitance of interconnects, is not considered. Thus, the synthesized circuit is expected to deviate from the original specifications, especially at high frequencies. The second category, performance-driven layout synthesis, feeds performance constraints, obtained from the sized circuit, to the layout generator. The placement tool in [4] tries to keep the performance degradation within imposed margins where sensitivity analysis is used to bound parasitics in [5]. Performance-driven layout synthesis approaches may suffer when the performance margins are not enough for the layout generator. Layouts for the traditional and the performance-driven layout synthesis approaches are generally generated by an optimization engine which tries to place and route the layout, as shown in Figure 1.1. These synthesis approaches do not guarantee to fully satisfy the circuit performance specifications after post-layout simulation. Moreover, the optimal sizing provided by the previous sizing step is not guaranteed to be optimal once accurate parasitics are considered. Furthermore, as designers shift to nanometric technologies, parasitic effects have become more significant. To completely fulfill the desired specifications, synthesized layouts need to be modified and iterations between circuit and physical synthesis are required, thus diminishing the advantages of automatic synthesis.

Recent works are based on layout-aware circuit synthesis which aims to combine physical and circuit synthesis into a single design loop. Three different techniques are used to integrate layout induced effects into the design loop:

(i) Optimization-based layout synthesis is performed within the global circuit synthesis loop and the parasitics [6, 7] are extracted as shown in Figure 1.2a. To be usable in a synthesis loop, the high computational time required for optimization may be decreased by strictly constraining the solution space [7, 8]. As more

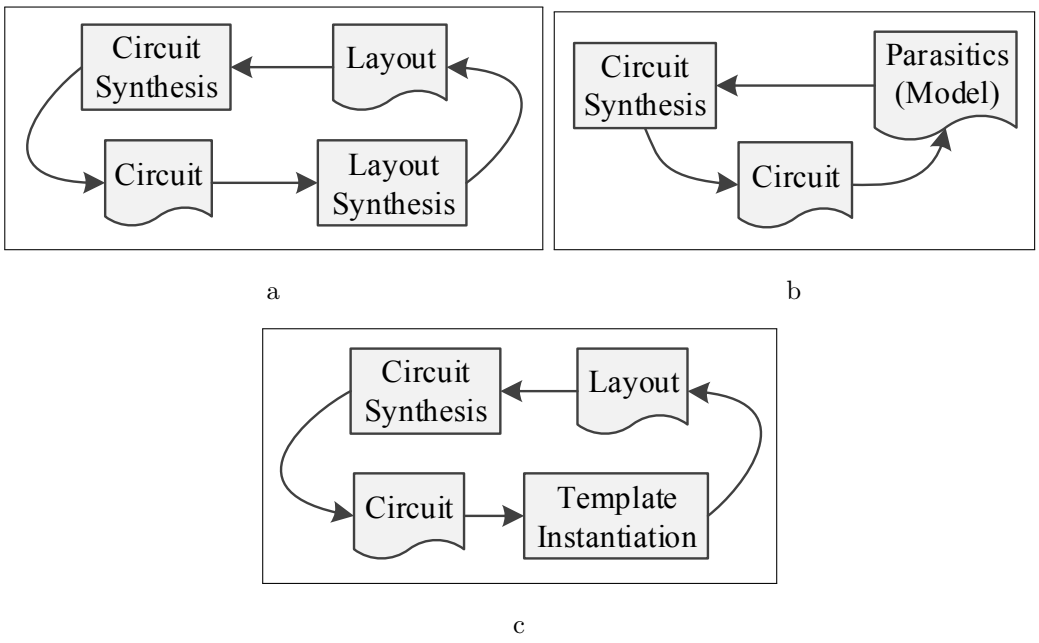


Figure 1.2. Layout Aware Circuit Synthesis Loops (a) Layout Synthesized through Optimization, (b) Parasitic Model used, (c) Template Instanced.

constraints are added, the solution space becomes smaller and the optimization approach turns into template instancing.

(ii) A layout is not generated in the synthesis loop, but parasitic effects are estimated through a model, as shown in Figure 1.2b. The model is generated by using a template and a parasitic extractor [9]. This approach annihilates the need for layout synthesis and extraction in the loop. Thus, it is fast, although the necessity for a template and parasitic models significantly limit this approach. On the other hand, the parasitic models are usually build from an interpolation of several layout instances. This is either very expensive if many samples are used or very inaccurate if few samples are used.

(iii) A template, a code including the placement and the routing information for a circuit, is used to instantiate a layout and then the parasitics are extracted [10–13] as shown in Figure 1.2c. Template-based synthesis makes use of stored design knowledge and its advantage is fast layout instancing, making it appealing for the layout-in-the-loop circuit synthesis approaches. A template is required for this approach. However, low flexibility of the templates and hardness of template coding limits its usage. Template-based layout synthesis approaches have palli-

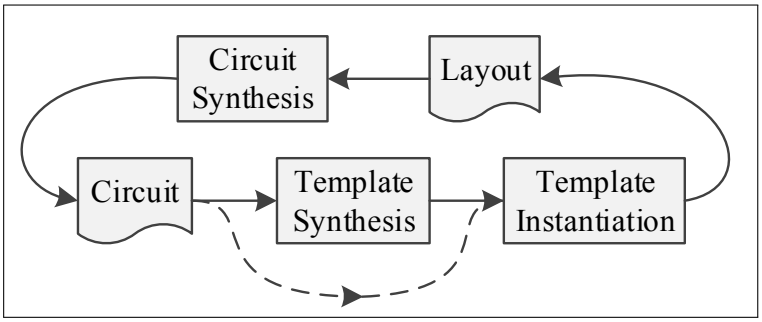


Figure 1.3. Template Synthesized and Reused.

ated the low flexibility of templates by structured template construction and, in some cases, by including the quality of the layout instance as an optimization parameter [14]. However, a single template cannot usually provide high quality layouts for the complete search space of design parameters.

From the approaches reviewed above, optimization-based sizing approaches with instantiation of layout templates in the loop provide the best speed / performance trade-off. The drawback of these approaches is two fold:

(i) Templates have been traditionally considered to have low flexibility. Even when detailed parameterization and optimization of the layout quality is performed during the optimization process, as in [14], a single layout template can hardly offer good performance in all regions of the search space of the design variables.
(ii) Template construction requires deep expertise and considerable effort.

A novel design automation paradigm is proposed in Figure 1.3, which enjoys the advantages of both optimization-based and template-based layout synthesis methods. The method is more flexible than conventional template-based approaches because a new template is synthesized when available ones do not provide a good enough quality of the layout instance at the current region of the search space in a given iteration of the circuit sizing process. The efficiency is much better than optimization-based approaches, and close to template-based approaches, because sizing relies on existing

templates in most iterations. However, to enable the design automation paradigm in Figure 1.3, the following problems must be addressed:

(i) Placement and routing tools must be able to synthesize templates, whereas available tools [15–26] were coded to output just individual layouts.
(ii) Templates must be flexible, at least in some part of the solution space.

This thesis introduces a template language, Layout Description Script (LDS), which supports the above statements. In addition, LDS supports analog layout constraints, e.g. symmetric placement constraints, and an LDS code may be automatically captured from an expert drawn layout. Also, device generators may be coded in LDS, thus LDS is capable of describing a layout completely.

Summing up, this thesis aims to develop a layout language to enable the design automation paradigm in Figure 1.3 and to completely represent a layout.

2. LAYOUT SYNTHESIS

This chapter introduces the layout description script (LDS) which is a domain specific language (DSL) intended to ease template coding. LDS is a hybrid language, that combines the advantages of sequential and declarative programming. A declarative programming language defines *what to achieve* without describing the control flow, whereas a sequential one describes *how to achieve a task*. Commonly sequential programming languages [11, 32–38] were used to code layout templates. These languages allow the designer to code parameterized device generators, which accepts some device parameters and outputs a layout. For instance, a transistor generator accepts the number of fingers as one of its input and outputs a transistor layout. However, also declarative programming capabilities are needed for a template language and the following discussion tries to answer the question "Why the declarative programming capabilities of LDS are needed?"

The order of constraints in LDS does not affect the layout. In a sense, assignments are automatically ordered during constraint satisfaction. For instance, assume that a designer wants to align the right edges of three rectangles as shown in Figure 2.1a, where these rectangles may be metal pieces or devices, such as transistors. This can be achieved by using the sequential code in Figure 2.1b. However, if the order of assignments is reversed as in Figure 2.1c, the designer will end up with an incorrect layout. The layout after executing the first assignment is depicted in Figure 2.1d, whereas the layout after the second assignment is provided in Figure 2.1e which does not correspond to the initially intended layout. If LDS were used instead of sequential assignments in these figures, the order would not affect the result.

If LDS is used, the designer is not responsible for taking care of different conditions during template coding; in other words, LDS constraints abolish the necessity for conditional statements required for coding flexible templates. Assume that the designer wants to align centers of two rectangles, namely: a and b. If the designer does not know which rectangle is longer, every case must be checked by if-else conditionals

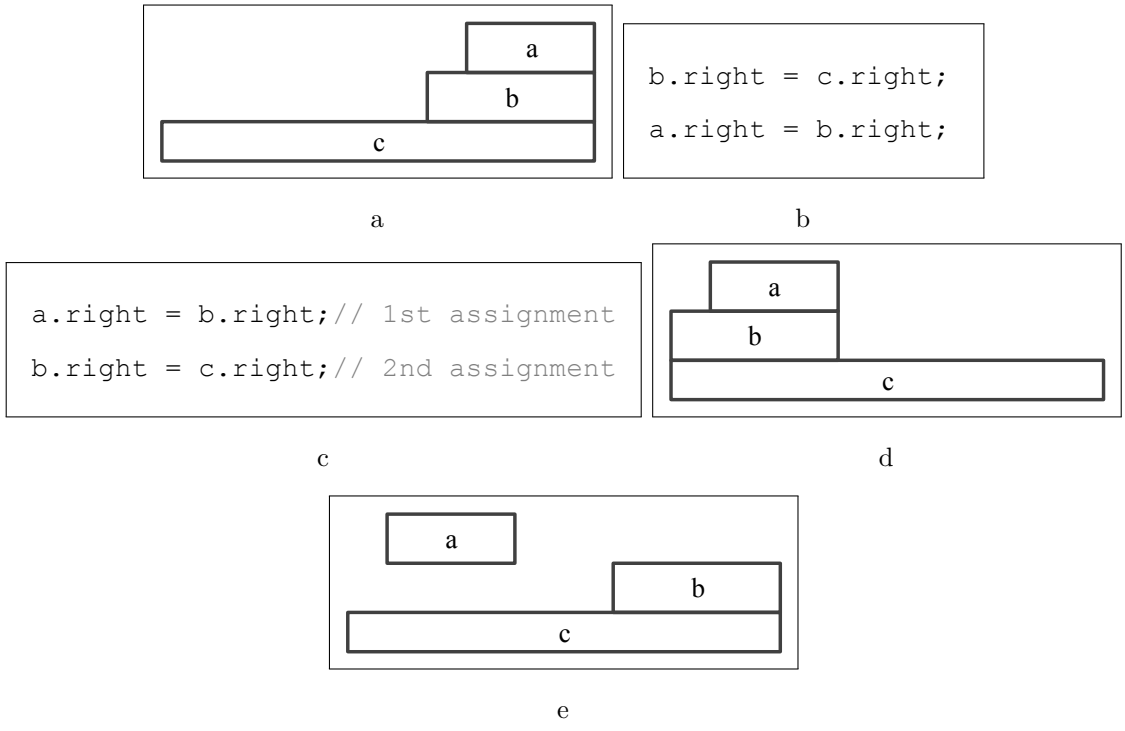


Figure 2.1. Order of Assignments (a) Placement, (b) Code to Align Right Edges as in (a), (c) Order of Assignment Changed, (d) Placement After the First Assignment in (d), (e) Placement after the second assignment in (d).

or min/max functions. Note that the length of the rectangle may be undetermined in a template. For instance, the length of a connection is not known a priori and it is going to be determined after sizing the devices, e.g., transistors. A simple sequential code to align the centers of rectangles a and b is given in Figure 2.2a. However by using LDS, the constraint in Figure 2.2b would suffice. Considering the case where the number of blocks is high and the layout is complicated, which is often the case for templates containing routing, template coding is going to be very time consuming and prone to errors by using a sequential language.

Alignment, boundary, symmetry, and similar constraints such as those mentioned in [42], may be coded without any concern about the type of graph representation, the kind of solver. This drastically reduces the time required to manually code a template and offers a straightforward procedure for EDA tools.

```
if (a.width >= b.width) {
        b.left = a.left + (a.right - a.left)/2;
} else {
        a.left = b.left + (b.right - b.left)/2;
}
```

a

```
(right(b) + left(b))/2 == (right(a) + left(a))/2;
```

b

Figure 2.2. Symmetry with (a) Sequential Code, (b) LDS.

In addition to declarative programming, LDS also supports sequential programming. It is possible to call a parameterized device generator from a declarative program; however, it is not possible to code one with a declarative language. This is due to the fact that the number of elements of a parameterized device is unknown. For instance, the number of fingers is an input to a transistor generator and it is not known a priori. Thus, it is not possible to define the required constraints for each finger without using a loop, which controls the program flow.

2.1. Related Literature

Available layout representations may be grouped into five classes, namely: structured, topological, sequential, declarative and hybrid.

Structured representations [30, 31], where only the absolute position of each rectangle is used, are specialized in storing the layout information. They have been commonly used for interfacing designers and fabrication foundries. Topological ones [27–29, 63–67] are built with tree structures or name sequences. Fast algorithms are available to convert these representations to absolute placements. Topological representations are used with optimization-based placement tools, where the layout constraints are imposed by these representations. They are usually handled by sequential programs and they do not contain any code themselves. Sequential languages [11, 32–38] can

also represent a layout. Not only absolute positions, but also templates or parameterized cells may be coded using these. Some of these sequential languages [32, 33] are general programming languages and are also integrated with commercial EDA tools. Some of them [34–38] are specific languages for layout description. The work in [11] is an application programming interface (API) offering functions to construct a Slicing Tree representation. Declarative languages in the table [39, 40] are designed for layout coding through constraint programming. The hybrid language in [41] generates a set of inequalities and then solves these.

Being a hybrid language, LDS supports constraint programming and allows coding parametrized device generators. CAIRO [11] and ALI [41] also have these properties. CAIRO [11] does not support general floorplans and is limited to slicing ones. Also, ALI [41] only accepts simple inequality constraints, and these constraints cannot be defined in functions. Moreover, it is possible to define an unlimited number of constraint problems with LDS. Thus, it is possible to code different parts of a layout with different representations and glue them.

Declarative programming capabilities in LDS, more specifically, constraint programming, not only simplifies template coding, but also yields more readable and cleaner codes than sequential programs. Constraint programming may be used to code a transistor generator for a fixed number of fingers. However, pure constraint programming is not enough to code a parametrized device generator (parametrized cell). LDS not only supports constraint programming but also supports sequential programming. The designer can freely control the flow of an LDS program and can add independent constraint blocks (constraint programs) into the flow, which makes LDS a hybrid programming language. This allows the designer to constrain topological representations and to satisfy different representations at the same time. For instance, it is possible to merge two or more Sequence Pair [28] representations, defined for different metal layers. Note that topological representations only allow constraining a single layer.

A comparison of LDS against the available layout representations is given in Table 2.1. The comparison is made in terms of *constraint programming*, *number of con-*

straint satisfactions, and *capability to handle parameterized cells*. The first parameter indicates the support for constraint programming. The parameter "number of constraint satisfaction" indicates the constraint satisfaction capabilities of the language. A "0" constraint satisfaction value means the language does not support constraint programming; a "1" value means the language can solve a single constraint problem. The capability to handle parameterized cells in column three of the table, indicates whether a parameterized device generator can be coded with the listed languages.

LDS is a combination of sequential and declarative programming and allows both constraint programming and coding parameterized device generators. Furthermore, the designer may solve an unlimited number of constraint satisfaction problems with LDS. Thus, the designer may exactly define the behavior of a rectangle when the layout is resized. This way, topological representations may be coded. Due to its support for constraint programming, EDA tools may easily generate layout templates in LDS.

2.2. Syntax

An LDS template starts with the keyword `layout`. A template may contain constructors to initialize the layout and it may also contain methods. Each method may contain conditional and loop blocks as well as `param` and `lds` blocks. A block is a piece of code, which starts with the symbol "{" and ends with the symbol "}". A sample LDS program is given in Figure 2.3. Conditional and loop blocks are common in sequential languages and are used to control the flow of the program. Code blocks `param` and `lds` are specific for LDS.

A layout inherits the methods and fields of another layout, if the `extends` keyword is used. For instance, the code "`layout MyLayout{}`" defines a new layout, named *MyLayout*, whereas the code "`layout MyLayout2 extends MyLayout{}`" defines a new layout which inherits from the layout named *MyLayout*.

The `param` block is used to ease parameterization, since layouts for analog devices may have many parameters. For instance, a transistor has poly width, poly length, and

Table 2.1. Comparison of LDS with Other Representations.

	Representation	Constraint Programming	# Constraint Satisfaction	Parametrized Cells
Structured	CIF [30] GDSII [31], etc.	NO	0	NO
Topological	Slicing Tree [27] Sequence Pair [28] B*-Tree [29], etc.	YES	1	NO
Sequential	SKILL [32] PYTHON [33] MSL [34] MOGLAN [35] BALLISTIC [36] TDL [37] LAYLA [38], etc.	NO	0	YES
	CAIRO [11]	YES (Slicing Tree)	1	YES
Declarative	PDI [39] RPL [40]	YES (No Symmetries)	1	NO
Hybrid	ALI [41]	YES (No Symmetries)	1	YES
	LDS	YES	∞	YES

```
layout nmos {
        param {
                double width = 1e-6;
                double length = 0.35e-6;
                int m = 1;
                // more parameters…

                // check parameters
                constrain width >= length;
                constrain width <= 100e-6;

                // override methods
                public length(double v) {
                        length = Math.max(v, 0.13e-6);
                }

                public constrain() {
                        // more cheks
                }
        }

        // constructor
        public nmos(...) {
                lds (x) {
                        // constraints
                        constrain lds : x(a.right) == x(b.right);
                        // or func(a, b);

                        // some code

                        for (int i=1;i<param.m;i++)
                                func(a, c[n].right);

                        // some code
                }

                // more code

        }

        private void func(Rectangle r1, Rectangle r2) {
                constrain lds : x(r1.right) == x(r2.right);
        }

        // more methods or constructors

        public constrain() {
                // more constraints
        }

}
```

Figure 2.3. An LDS Template.

number of fingers as the primary parameters and lots of secondary parameters such as: metal length, dummies, etc. Most of the time, a transistor is instanced by assigning its width, and the remaining parameters are set to their default values. LDS supports template constructors with variable arguments. If values are not specified for some parameters, the default values are used. Default values for the parameters are set in the `param` block placed at the beginning of the layout description. If a layout inherits from another one, its parameters also inherit from the parameters of that layout. For instance, the parameter *width* is also a parameter of a layout which inherits from the description in Figure 2.3. The parameters are represented by the symbol "..." in a method definition. The constructor definition in Figure 2.3 accepts the parameters as inputs, and stores their values. The methods, which accept the parameters, are called using the assign operator "`->`". As an example, the constructor in Figure 2.3 may be called by using the code "`nmos( ...->{length=1e-6, width=1e-6} )`". In a `param` block, the keyword `constrain` may be used to check the parameter ranges, also the parameters may be checked in the method named *constrain*. If any of the parameters violates its limits, an error is output. Methods, having the same names as the parameters, may be defined to do advanced assignments. The *length* method in the `param` block of Figure 2.3, sets the value of the *length* parameter if the assigned value is greater then a predefined value.

Declarative code in LDS must be coded in `lds` blocks. During run time, the constraints in an `lds` block are used to construct a constraint satisfaction problem and it is solved at the end of the block. The types of the constraints in an `lds` block are specified in parenthesis: `(x)` for horizontal, `(y)` for vertical, and `(x,y)` or `(y,x)` for both horizontal and vertical constraints. For instance, the `lds` block in the sample code in Figure 2.3, defines a horizontal constraint problem, which is defined as "`lds(x)`". It is possible to define an `lds` block within another one.

The constraints in a problem are defined using the `constrain` keyword. It is also possible to define objectives for a problem using the `objective`, the `minimize` or the `maximize` keywords. Figure 2.4 shows the usage of these. If these keywords are used without specifying the problem, the related objectives or constraints are added

```
constrain [(lds|Problem):] (Constraint|Problem)
minimize [(lds|Problem):] expression
maximize [(lds|Problem):] expression
objective [(lds|Problem):] (Objective|Problem)
constrain final [Problem:] expression
```

Figure 2.4. Constraints and Objectives.

to the layout in which they are defined. Note that, every layout in LDS is a constraint satisfaction problem. If the problem is specified, the constraints and the objectives are added into the specified problem. It is also possible to use the `lds` keyword after the `constrain`, `objective`, `minimize`, and `maximize` keywords; in this case, the related constraint or the objective is added to the problem of the `lds` block, where every `lds` block is a constraint satisfaction problem. After the `constrain` and the `objective` keywords, instead of a constraint or an objective, a problem may be specified. In this case, that problem is added as a sub problem, which results in a hierarchy of problems.

Also the `final` keyword may be used after the `constrain` keyword. A final constraint is used in sub problems and it is satisfied in that problem. An example is given in the following section.

2.3. LDS Problems

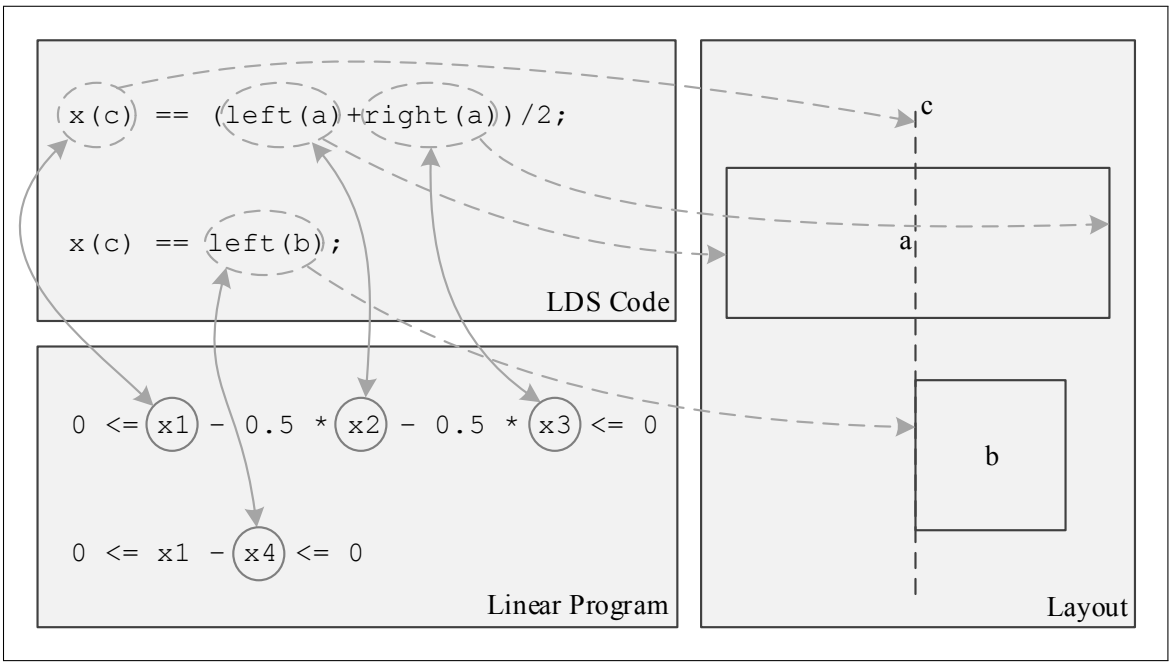
LDS problems are described in `lds` blocks. The codes given in Figure 2.5a and in Figure 2.5b are used to explain the concept of an LDS problem. The `lds` blocks in these code samples are defined as `lds(x)`, which tells the compiler to generate a constraint program and to add a variable to this constraint program when a variable is input to function `x`. The code in Figure 2.5a contains assignments and there is no constraint in this block. First, the value "3" is going to be assigned to the variable a, and then the value "5" is going to be assigned to a. At last, value stored in a is going to be assigned to b. At the end, a and b are going to have the value "5". However, the code in Figure 2.5b contains a constraint, which is going to change the behavior

```
lds (x) {
        a = 3;
        a = 5; // assignment
        b = a;
}
```

a

```
lds (x) {
        a = 3;
        constrain lds:x(a) == 5;
        b = a;
}
```

b



c

Figure 2.5. LDS Codes (a) without Constraint, (b) with Constraint. (c) Relation between LDS Code, Layout, and Linear Program.

of the program. After the first assignment, a is going to have the value "3". The following statement is a constraint and will not change the value of a immediately. This constraint is added into the generated linear program. After this constraint, the second assignment sets b to the value of a, which is "3". At the end of the block, the linear program is going to be solved and the values of the variables are going to be updated. Thus, the resulting value for a is going to be "5", and that of b is going to be "3". If there were more constraints, they would be satisfied together. This way, LDS supports constraint programming.

Layouts have predefined *top*, *right*, *bottom* and *left* methods. These represent the edges of a layout and the user is free to define more methods.

```
Double n, m;
lds (x,y) { //1st problem
      lds (y) { //2nd problem
            lds (x) { // 3rd problem
                  constrain lds : x(n) >= 0; // into 3rd
                  constrain lds : x(n) + y(m) >= 0; // into 1st
                  constrain lds : y(m) >= 0; // into 2nd
            }
      }
}
```

a

```
// constraint 1st
constrain x(a) + x(b) >= 5;
//objective 1st
minimize x(b);
```

Super Problem

```
constrain x(a) >= 3; //2nd
constrain x(b) >= 3; //3rd
minimize x(a); // obj. 2nd
constrain final x(a);//4th
```

Sub Problem

b

```
// update as 3 + x(b) >= 5
constrain x(a) + x(b) >= 5;
//objective 1st
minimize x(b);
```

Super Problem

```
constrain x(a) >= 3; //rmv
constrain x(b) >= 3; //3rd
minimize x(a); // rmv
constrain x(a) == 3; //rmv
```

Sub Problem

c

Figure 2.6. Problem (a) Nested Problems, (b) Sample Code with the `final` Keyword, (c) New Problem.

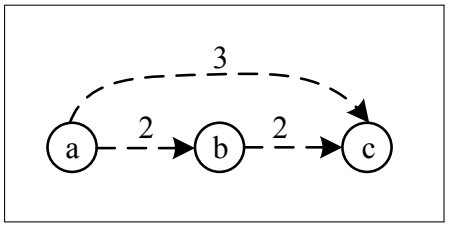
An `lds` block is converted to a constraint program, i.e. a linear program, and it is solved. Figure 2.5c illustrates the relationship between an LDS code, the corresponding layout, and the generated constraint program.

An `lds` block may be defined within another `lds` block. A constraint or an objective is added to the closest problem, accepting the constraint or the objective. In Figure 2.6a, three `lds` blocks are defined. The first one is for horizontal (`x`) and/or vertical (`y`) constraints or objectives, the second one handles only vertical (`y`) constraints or objectives, whereas the third block takes care of only horizontal (`x`) constraints. There are also three constrains defined in these blocks. The first constraint is a horizontal (`x`) constraint, thus it is added to the third problem, the closest problem which accepts horizontal (`x`) constraints. The second constraint is both a horizontal and a vertical (`x`,`y`) constraint and this constraint cannot be stored in the second or the third problems. But, the first problem accepts it. The third constraint is a vertical (`y`) constraint, thus it is stored in the second problem.

The `final` keyword after the `constrain` keyword may be used to isolate some part of a sub-problem from a super problem. The sample code in Figure 2.6b contains nested problems, and in the sub-problem, the `final` keyword is used in the fourth constraint. In this case, a local problem is constructed where only the constraints and objectives related to variable a are included. Thus, the second constraint and the second objective are included in this local problem. The local problem is solved and the result is added as a new constraint to the original problem. For the case of the problem in Figure 2.6b, the new constraint is going to fix the variable a to the value "3". The new problem is given in Figure 2.6c. The problem is then simplified, the variable a is replaced with the value "3"; thus, the underlined constraints and objectives in Figure 2.6c are going to be either removed or updated as commented in the figure.

```
constrain x(b) >= x(a) + 2;
constrain x(c) >= x(b) + 2;
constrain x(c) >= x(a) + 3;
```

a



b

Figure 2.7. Redundant Constraints Removed (a) the Constraints, (b) the Constraint Graph.

2.3.1. Pre-Solving

Every layout, even a contact, in LDS, has constraints; thus, analog layouts, having more than 10 devices, result in a large constraint problem. A pre-solver is used to reduce the number of constraints. The following techniques are applied:

- The equalities are used to reduce the problem size. For instance, if variable a is constrained to have the same value as the variable b, in all constraints and in all objectives, the variable b is replaced with the variable a.
- A constraint graph is constructed and the redundant edges of this graph are removed. For instance, for the constraints given in Figure 2.7a, the constraint graph in Figure 2.7b is constructed. The longest distances between the nodes are calculated, the distance from node a to node c is found to be "4". Then, the edge having the weight "3" is detected to be redundant and the corresponding constraint in Figure 2.7a is removed. If a constraint can not be represented in a constraint graph, it is not added to the graph.

2.3.2. Hierarchical Solver

The coder is free to nest several `lds` blocks. This allows solving the constraint for individual devices in separate `lds` blocks. The relations between the individual devices may be constrained in the outermost `lds` block. However, sometimes it may be useful to solve the layout constraints in a single `lds` block. Consider the following cases:

(i) In Figure 2.8a, metals carrying high current are connected to the port of device B. In this case, the port of device B is used to connect ports of the devices A and C. If the constraints of the devices are solved in separate `lds` blocks, the height of the port of device B is fixed. However, if all the constraints are solved in a single `lds` block, the layout in Figure 2.8b is obtained where the resistance of the path from device A to device C is reduced.

(ii) In Figure 2.8c, a transistor is instanced in an `lds` block, then connections -the dashed rectangles- are added. In this layout, a spacing rule is violated where the metals -light gray rectangles- are too close. However, if related constraints are coded in a single `lds` block, the resulting layout is the one in Figure 2.8d. The transistor is now capable to modify itself to cope with the minimum spacing rules. Note that, this is case if the device generator allows enlargement.

(iii) In Figure 2.8e, a transistor with bulk connections is given. If the transistor is fixed, solved in a separate `lds` block, it is not possible to add the connection in Figure 2.8f. However, if the layout is solved in a single `lds` block, the generated device is more flexible.

A hierarchical approach is developed. The global problem is separated into sub-problems and a problem is generated for every layout in the hierarchy, and also a table, counting the variables, is built for each problem. After building the tables, constraint graphs are constructed for each problem, where nodes corresponding to the local variables are colored in white and the others are colored in gray. If a constraint or objective can not be added to the constrained graph, the nodes corresponding to the variables of that constraint are also colored in gray. Then, the approach in [57] is used to eliminate the white nodes and the remaining graph is combined with the graph of the parent problem.

A problem, with three levels of hierarchy, is given in Figure 2.9a where the variable a is used only once in the problem *P3* and an entry {a, "1"} is added to the table stored in that problem as shown in Figure 2.9b. The same variable a is also used in the problem *P2* and the related row of the corresponding table has the entry {a, "2"},

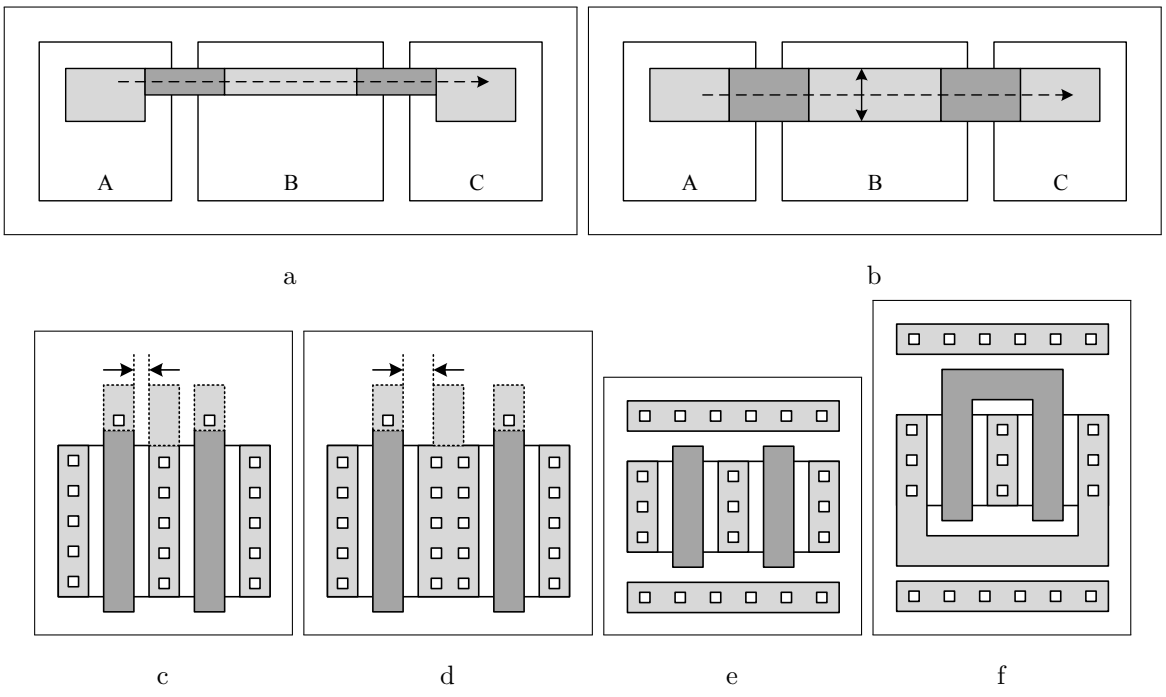


Figure 2.8. Flexible Layouts, Port Shown in (a) Enlarged in (b), Spacing Rule Violation Shown in (c) Corrected in (d), Space Added to the Device in (e) as Shown in (f).

which means the variable a is referenced two times in the problem *P2* where one of the references is given in the problem *P3* and the other reference is given in the problem *P2*. If the entry for a variable in a problem is same as the entry for that variable in the parent of the problem, that variable is called a *local* variable in that problem. In Figure 2.9b, the table entries corresponding to the local variable, are filled with white and the non-local ones are filled with gray.

Building the tables, constraint graphs are constructed for each problem. The constraint graph for the problem *P3* is shown in Figure 2.9c, where all the nodes are gray and non of them can be eliminated. Then, the constraint graph of the problem *P3* is combined with the constraint graph of the problem *P2*. The resulting graph is shown in Figure 2.9d, where the node corresponding to the variable a is a white node and can be eliminated as shown in Figure 2.9e.

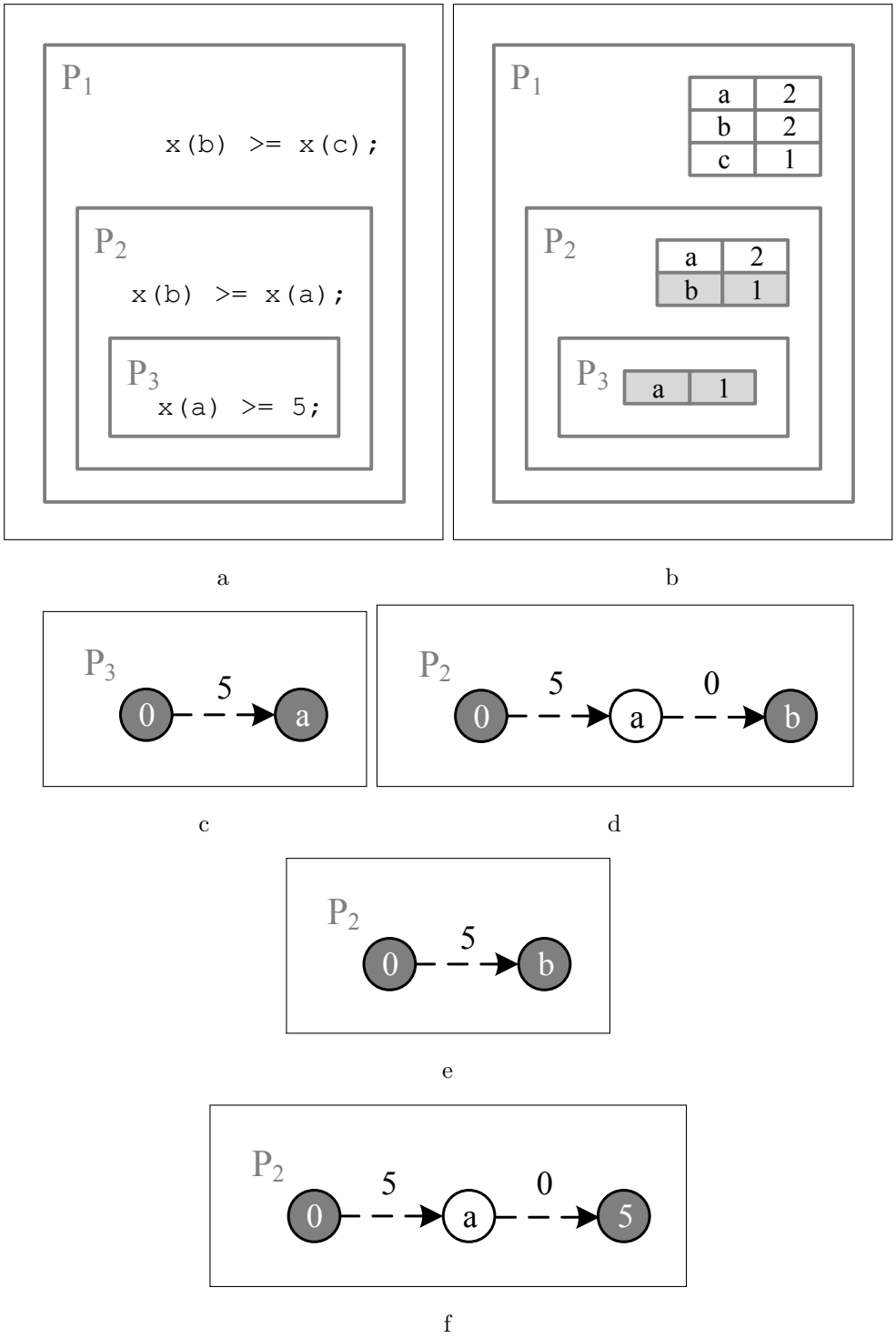


Figure 2.9. Hierarchical Problem (a) Constraints for *P1*, *P2* and *P3*, (b) Tables Stored in *P1*, *P2* and *P3*, (c) Constraint Graph for *P3*, (d) Constraint Graph for *P2*, (e) White Nodes in (d) Eliminated, (f) Constraint Graph for *P2* after Solving *P1*.

Table 2.2. Comparison of Run Times Costs for Solving Constraints for Transistor Arrays of Sizes 1x1, 2x2, 4x4, 6x6, 8x8.

# Devices	1	4	16	36	64
Single	0.014s	0.128s	1.299s	5.890s	20.22s
Separated	0.021s	0.080s	0.332s	0.820s	1.453s
Reduced	0.028s	0.145s	0.622s	1.410s	2.920s

The problem *P1* is the top level problem and a new constraint "`x(b) >= 5`", originating from the graph in Figure 2.9e, is added to it. The new problem is solved and the value of the variable b is found to be "5". Then, the value of this variable is used in the sub-problem *P2*, where its constraint graph is given in Figure 2.9e. The new graph for the problem *P2*, after replacing the variable b with its value, is given in Figure 2.9f. The new sub problem for *P2* is solved and the value of the variable a is found to be "5".

By making use of the hierarchy in the layout, the hierarchical solver automatically reduces the problem size. The run time impact of the automatic reduction algorithm is tested on transistor arrays of size 1x1, 2x2, 4x4, 6x6, 8x8. The results are compared with the run times of the following cases:

- A single problem is generated for the complete layout and solved.
- Independent problems are generated for each transistor and they are separately solved.

The results are given in Table 2.2, and also a log-log scaled plot is given in Figure 2.10. The solution time required for the reduced problem is close to that for the separated problem and much better than the solution time required for the single problem. Thus, using the hierarchical solver, flexible device generators/layouts like the ones in Figure 2.9b, Figure 2.9d, and Figure 2.9f may be coded and the run time cost is acceptable.

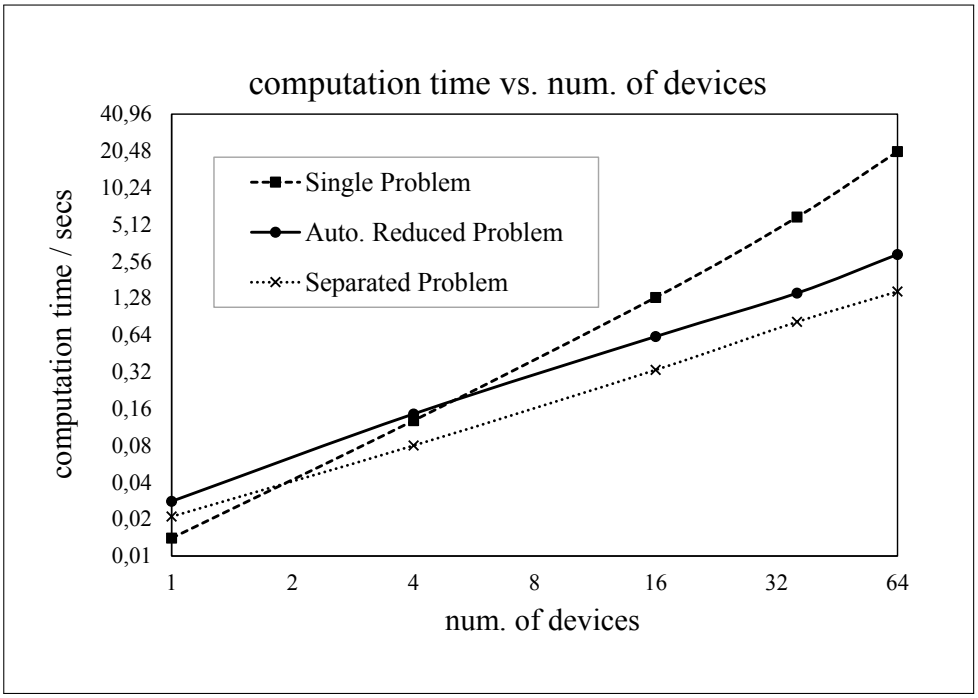


Figure 2.10. Comparison of Run Time Costs.

2.3.3. Finding the Absolute Positions

If the relative positions are only defined in one dimension, it is not possible to size a template in horizontal and vertical dimensions simultaneously. This is the case if the layout is not defined by a topological representation such as a Slicing Tree [27] or a Sequence Pair [28]. An example is shown Fig.2.11a, where the rectangles b and c are going to be expanded as in Fig. 2.11b. To prevent any overlap between b and c, the constraints must be defined in concurrent `lds` blocks: either `lds(x){lds(y){...}}` or `lds(y){lds(x){...}}`. If the blocks are written as `lds(x){lds(y){...}}`, the layout is going to be first resized in the vertical dimension, and then in the horizontal dimension. In this case, the layout in Fig.2.11a is going to be updated first as in Fig.2.11c, and then as in Fig.2.11d. For the other case, `lds(y){lds(x){...}}`, the layout is going to be first resized in the horizontal dimension and then in the vertical one. Layouts corresponding to horizontal and vertical updates are given in Fig.2.11e and Fig.2.11f, respectively.

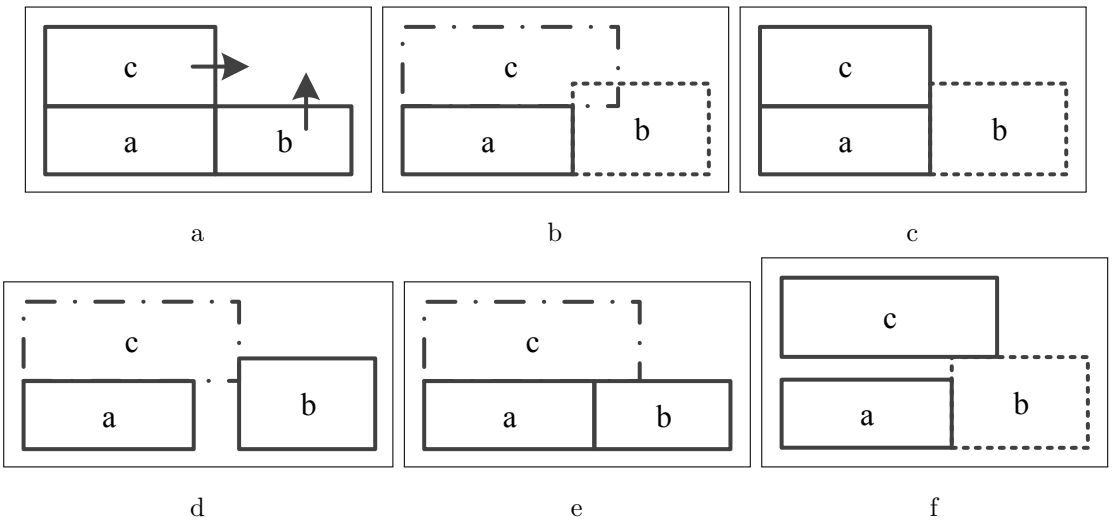


Figure 2.11. Overlap-free Layout with LDS (a) Initial Placement and Sizes, (b) Rectangles b and c Expanded, Updates for `lds(x){lds(y){...}}` Given in (c) and (d), Updates for `lds(y){lds(x){...}}` Given in (e) and (f).

2.4. Layout Constraints

Constraint programming in LDS eases coding layout constraints. A constraint is defined via the `constrain` keyword, and its usage is shown in Figure 2.4. A constraint is composed of three parts, namely: left hand side expression, comparator, and right hand side expression, where the comparator is one of the following symbol pairs: "==", ">=" and "<=". Placement, alignment, symmetry, and similar constraints [42] can be easily coded in LDS. Examples for placement, alignment, abutment, and symmetry constraints are given in Figure 2.12. Placement constraints are required to code the relative positions of the rectangles in a layout, and these constraints are coded using symbols for the inequalities, ">=" or "<=". Alignment and abutment constraints are similar to placement, but equality symbols "==" are used. Symmetry constraints, essential for analog layouts, are coded with equalities and the designer can locate two rectangles symmetrically with respect to a symmetry axis. Details for the data structures, supported by LDS, are presented in the following section.

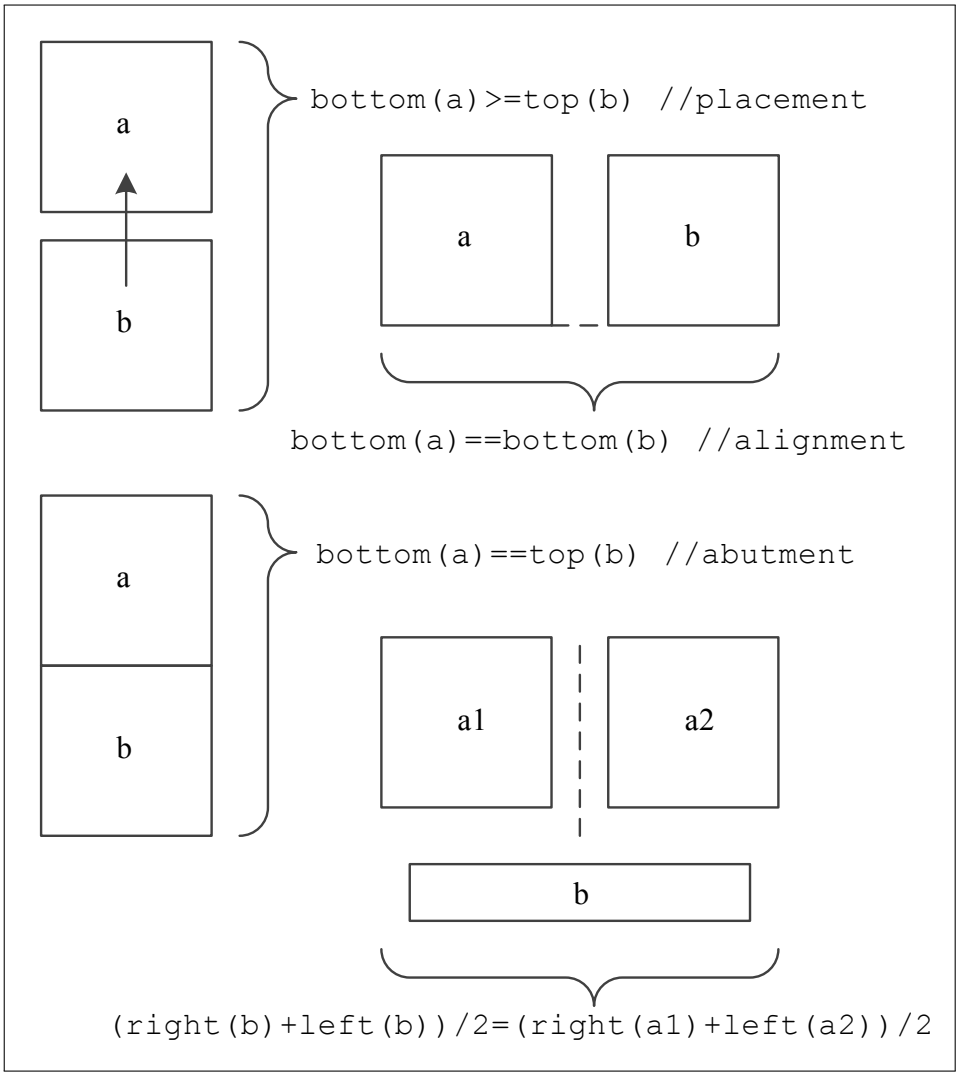


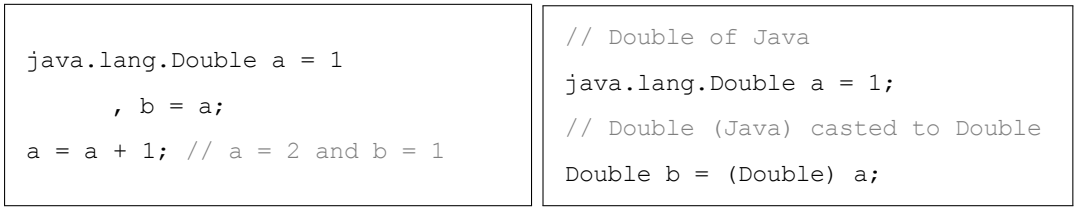
Figure 2.12. Placement, Abutment, Alignment, and Symmetry Constraints in LDS.

2.5. Data Structures

This section describes the data structures used in the implementation of LDS. Symbolic expression and the layout components may be represented with these structures.

2.5.1. Double, Integer and Long

The *Double*, *Integer* and *Long* classes of LDS inherit from the *Number* class. These classes are pointers to Java classes *Double*, *Integer* and *Long*, respectively. They



a b

Figure 2.13. Numerical Classes (a) Pointed Values Not Modified, (b) Explicit Casting.

differ from the corresponding Java classes, such that the pointed value may be changed. Numerical Java classes do not allow the coder to modify the pointed values. Consider the code in Figure 2.13a, the reference a first points to an address which stores "1", then it points to an address which stores "2". This code does not modify the memory position in which the number "1" is stored, thus the variable b has the value "1". This is a good property and prevents erroneous changes in the values of the variables. However, if it is required to modify the memory address which stores the number "1", there is no way to modify it using the numerical classes of Java. The *Double* class of LDS behaves similar to the *Double* class of Java, it also has the method *set* which allows to modify the pointed value. This method is hidden from the users, only the LDS libraries can reach it and use it. Similarly the *Integer* and the *Long* class of LDS has this *set* method, to allow the libraries to modify the pointed values. The operators, "+", "-", "*", "/", ">", "<", ">=", "<=" and "==", are overloaded for the numerical classes of LDS.

Table 2.3 lists the resulting data types when two numeric classes are added, subtracted, multiplied, or divided. In this table, type (class) names in italic are the Java types, whereas the others are the LDS types. Explicitly, casting one of these types to another one is supported. A sample cast from a Java type to an LDS type is shown in Figure 2.13b.

2.5.2. Term

Symbolic expressions are held in the *Term* class. This class works with references to *Double* and *Integer* classes and stores them in a set. There are two term types: x

Table 2.3. Resultant Types of the Operations Between Two Numeric Types.

	Double	*Integer*	*Long*	Double	Integer	Long
Double	*Double*	*Double*	*Double*	Double	Double	Double
Integer	*Double*	*Integer*	*Long*	Double	Integer	Long
Long	*Double*	*Long*	*Long*	Double	Long	Long
Double	Double	Double	Double	Double	Double	Double
Integer	Double	Integer	Long	Double	Integer	Long
Long	Double	Long	Long	Double	Long	Long

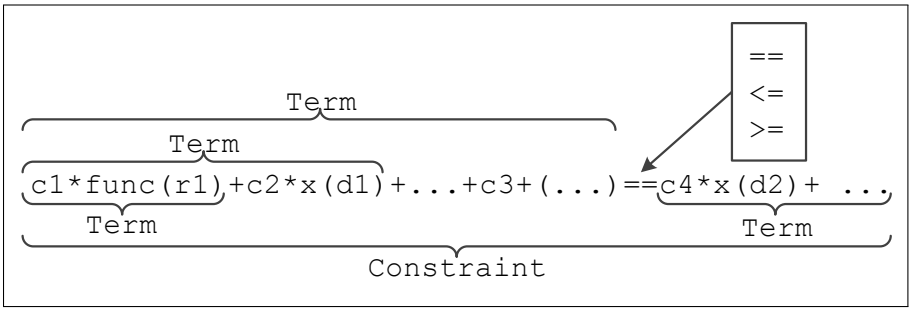
and y. There are also super terms, v typed terms, which may be used instead of x or y typed terms. These types are used to separate the constraints and the objectives of horizontal (x) and vertical (y) problems. The necessary codes to instance x, y and v typed terms are shown in Figure 2.14a.

Constraints are composed of terms as shown in Figure 2.14b. Every constraint holds a term, which is obtained by subtracting the right hand side expression of the constraint from the left hand side expression. In Figure 2.15, the code in the dashed rectangle is a constraint. The steps to find the term for this constraint are also shown in the same figure. The variables in *Term* structure are considered as constants if x, y methods or a method returning an x or a y typed term is not used. Otherwise, the addresses pointed by these variables are added into the set. Operation with a *Term* structure results in a new *Term* structure, thus the operand *Term*s are not affected. *Term*s are designed to operate fast and a set is used to store the pointers to the *Double* or *Integer* typed variables.

Term structure supports addition, subtraction, division, and multiplication operations. Addition and subtraction operations can be performed between two *Term* structures or between a *Term* structure and a numerical type, such as *Double*. Multiplication and devision operations can be performed between a *Term* structure and a numerical type. List of possible operations and the types of the operands are listed in Table 2.4.

```
Double n;
// this generates an x typed Term which refers to n
Term a = new Term(n, Term.x);
// this generates a y typed Term which refers to n
Term a = new Term(n, Term.y);
// this generates a v typed Term which refers to n
Term a = new Term(n, Term.v);
```

a



b

Figure 2.14. Symbolic Calculations (a) Sample Code to Instance `x`, `y` and, `v` Typed *Term*s, (b) A Constraint is Composed of *Term*s.

Table 2.4. Operations Supported by the *Term* Class.

Operation	First Operant	Second Operant
Addition and Subtraction	*Term*	*Number* (Java)
	Number (Java)	*Term*
	Term	*Number*
	Number	*Term*
	Term	*Term*
Multiplication	*Term*	*Number* (Java)
	Number (Java)	*Term*
	Term	*Number*
	Number	*Term*
Division	*Term*	*Number* (Java)
	Term	*Number*

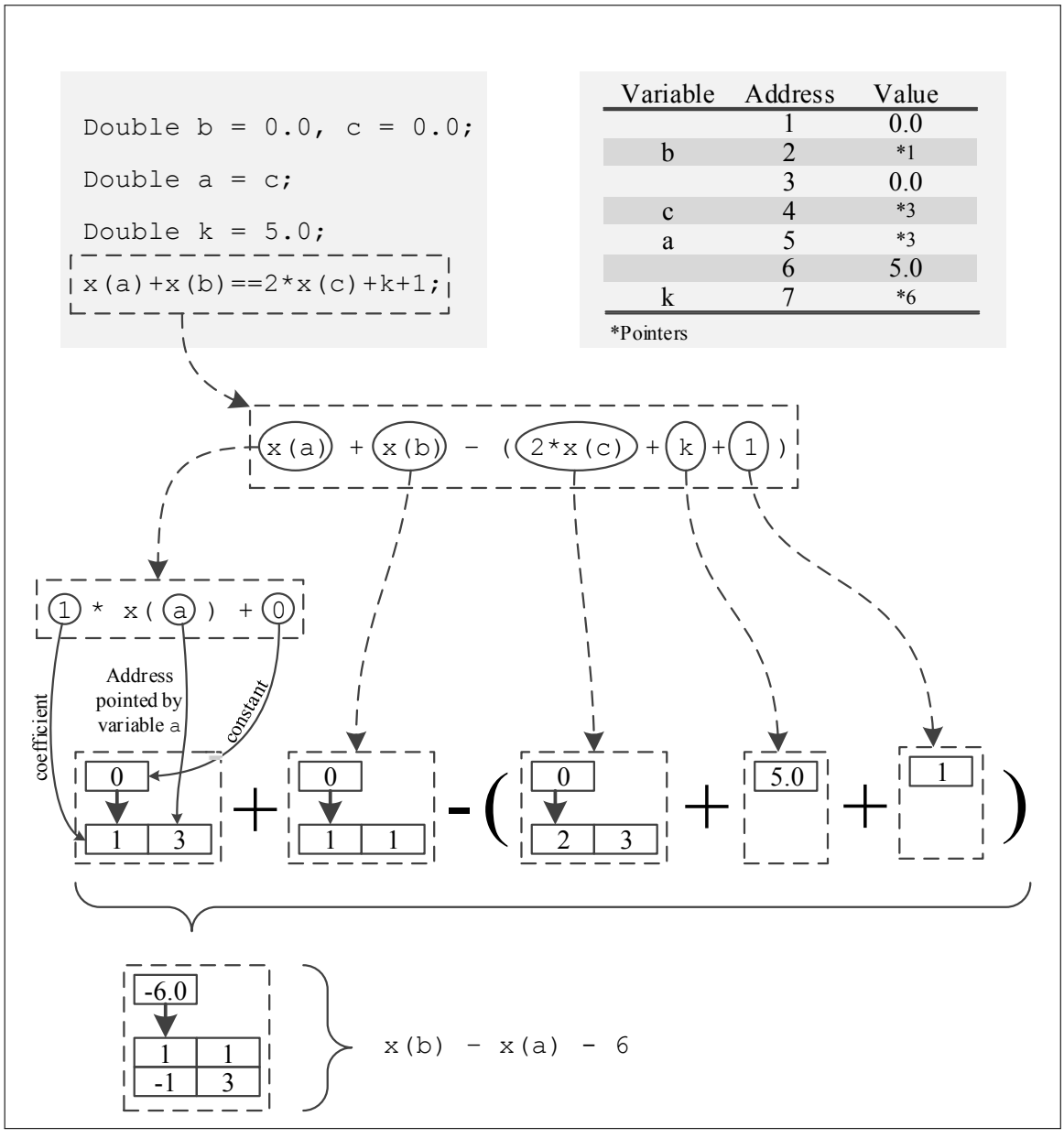


Figure 2.15. Calculations on *Terms* Handled via Special Data Structures.

By applying the following strategies, the performance of the *Term* class is improved:

- Ordered sets are used to hold the list of the variables and their coefficients. Thus, the variables in addition and subtraction operations, are matched very fast.
- When a *Term* structure is multiplied with a constant, the coefficient of the new *Term* is updated and the pointer to the set of the operand *Term* is copied. The set is not copied, just the reference to the set is copied.
- If two *Term* structures are added, the one containing more references is cloned and the set of the shorter one is added onto the set of the cloned *Term*.
- *Term* structures are not modified in the operations, new *Term* structures are generated. Thus, variables set in a *Term* instance are processed once and the results may be reused, several times.

2.5.3. Rectangle

*Rectangle*s are the basic shapes in LDS. When a new layout is defined using the `layout` keyword, its type (class) is the *Rectangle* class. A rectangle has three basic relationships with other rectangles, namely:

- A parent is assigned to every rectangle, whereas a *null* parent means the substrate is the parent. A rectangle may have children and all the constraints defined in a rectangle must be related to its children. A parent rectangle is responsible for its children, in the cases of: transformations, collecting the constraints, exporting and painting. When a rectangle is a super problem, it asks its children for their constraints and objectives.
- Rectangles also have supervisors, shortly named as supers. Although, a rectangle may only have a single parent, it may have several supers. Similarly, a super may have several members. A member rectangle owns the properties of its supers. Consider a contact between diffusion and metal layers in a CMOS process. The contact, itself is in the *contact* layer and one of its supers is the *diffusion* layer, and the other super is the *metal* layer. This super-member relationship is used

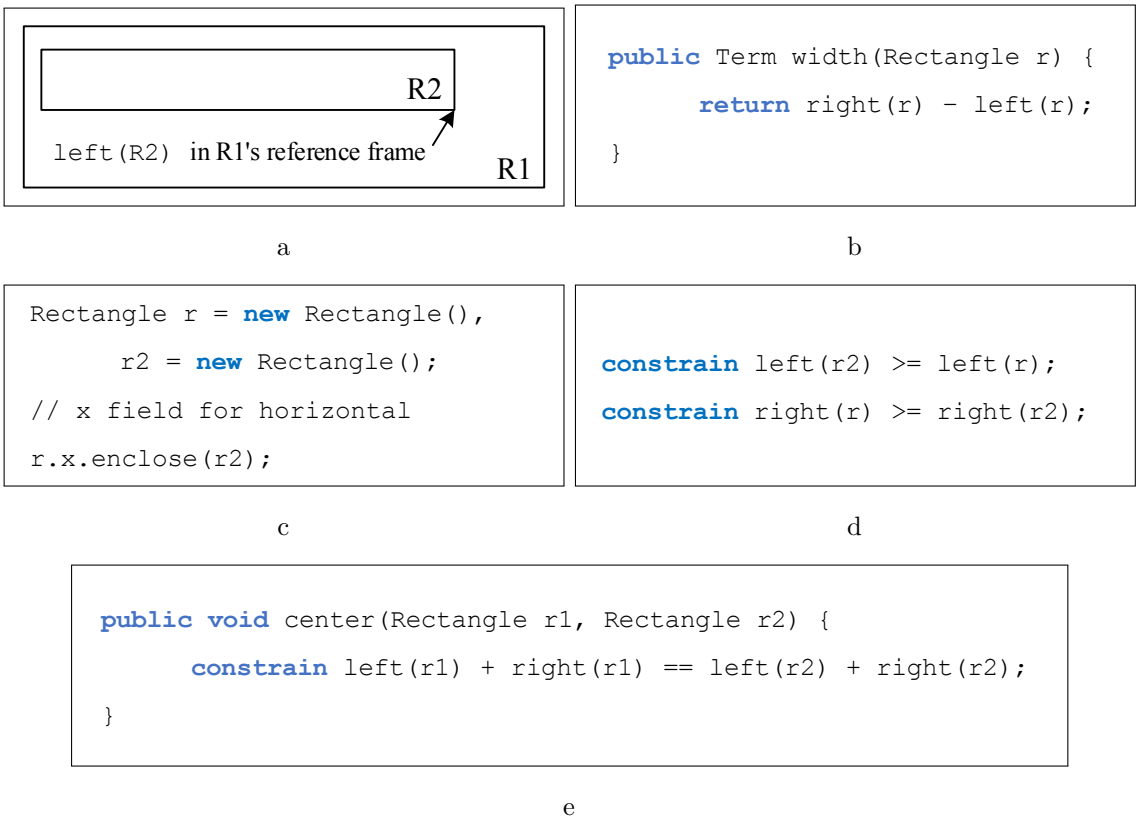


Figure 2.16. Properties of Rectangles (a) `left` is Aware of Transformation, (b) `width` Defined in Rectangle, (c) `x.enclose` Horizontally Constrains, (d) Equivalent of Code in (c), (e) User Defined Function `center` Aligns Centers of Rectangles.

to find the most appropriate design rules between rectangles. For instance, the design rules for a *polysilicon contact* may be different than a *diffusion contact*.

- Rectangles also have ports, which are the inputs/outputs to/from a rectangle. Rectangles can see each other's ports. For instance, ports of a transistor in a CMOS process are the drain, source, gate, and the bulk rectangles. Although, a transistor has several contacts and layers, these are private and need not be seen by the outer rectangles.

Each instance of the *Rectangle* class contains four *Double* variables which hold the positions of its edges. Four methods are defined in the *Rectangle* class, these are: *top*, *right*, *bottom*, and *left*. Each of these functions returns a *Term* which refers to the corresponding variable. Returned *Term* structures are context dependent, they are aware of the transformations. Consider the case in Figure 2.16a, where R1 is

horizontally flipped. Thus, the term "`left(R2)`", defined in R1, refers to the right edge of R2 with respect to the substrate frame. The transformations are considered at the last step, when the problem is being solved. Thus, the user is free to do the transformations before calling the methods -*top*, *right*, *bottom*, and *left*- or after calling them.

Rectangles also have the predefined *width* and *height* methods to ease coding. The *width* method is predefined as in Figure 2.16b. Similarly, the `height` method is a predefined method of the *Rectangle* class. Some of the methods of the *Rectangle* class and the descriptions of these methods are listed in Table 2.5.

The rectangle also has methods to constrain other rectangles, horizontally and vertically. These may be called using the x and y fields of a rectangle. A sample code is given in Figure 2.16c, where rectangle r horizontally encloses rectangle r2. Assuming the code in Figure 2.16c belongs to rectangle r, simplified version of this code is given in Figure 2.16d. The one-dimensional constraining methods are listed in the Table 2.6. These methods are only added to ease coding and to reduce the code size. The user is free to define his/her own methods. For instance, the method in Figure 2.16e aligns the centers of two rectangles, horizontally.

2.5.4. Layer

Layer is an interface and represents physical, virtual or derived layers. A physical layer represents the layers in the process, such as the *diffusion* or the *metal* layers for a CMOS process. The virtual layers are not real layers, e.g. module layer which does not correspond to a real layer and is not fabricated. Derived layers are generated when an operation is performed on a physical/virtual layer.

All classes implementing the *Layer* interface have signatures to store their layers. Physical and virtual layers are represented by a single bit. Operations on layers are binary operations, which are very efficient. If an operation such as and/or/xor, is performed on a physical layer or on a virtual layer, a derived layer is generated. Every

Table 2.5. Methods Defined in the *Rectangle* Class.

Method	Description
setTop, etc.	Sets the value of the variable.
setNet	Assigns a net to the rectangle.
setLayer	A layer is assigned.
getLayer	Returns the layer assigned to this rectangle.
setParent	The rectangle is assigned as a child of a parent.
addChild	A child rectangle is added.
getChildren	Returns all children of the rectangle.
setPort	Sets a port type, such as: GATE, IN, OUT, etc.
getPort	Returns the port type of the rectangle.
addPort	Adds port rectangles to this rectangle.
getPorts	Returns all the ports registered to the rectangle.
addSuper	The rectangle is assigned as a member.
getSupers	Returns the supervisor rectangles.
superLayer	Concatenates the layer of the rectangle with its supervisor's layers .
addMember	This rectangle is assigned as the supervisor.
getMembers	All the members of this rectangle are reported.
transform	Applies a transformation to the rectangle.
fill	Calls the fill function of its children.
enclose	The rectangle encloses another rectangle.
eqclose	The rectangle exactly encloses another one.

Table 2.6. Vertically/Horizontally Constraining Methods.

Method	Description
abut	Abuts the rectangle after/below another rectangle
connect	Aligns and abuts two rectangles
enclose	The rectangle encloses another one
eqclose	Similar to enclose but exact
place	Places a rectangle after/below the rectangle

operation on derived layers results in a new derived layer. All layers are defined in technology files and they are specific to the technology. A layer has five methods, these are *and*, *or*, *xor*, *contains*, *equals*.

The binary operations are performed on the signatures of the layers. The signature is a sequence of bits, where each bit corresponds to a physical or virtual layer. The lengths of the bit sequences in different instances of the *Layer* may not be same, however any binary operation pads the shorter one with zeros. For instance, if the operation is "10" & "001", the shorter signature is padded with zeros to obtain "010". Then, the binary operation is performed.

The method `contains` in the *Layout* interface returns "true" if the input layers exists in the instance layer. And the operation equals returns "true" if two layers are exactly same. Also the operators "+", "<", ">" and the operator "==" are overloaded. The plus operator "+" logically ORs the signatures of the operand layers. The greater ">" and the smaller "<" operators check if the layer contains another one. The equality "==" operator or the `equals` method checks whether the operant layers contains the same physical and virtual layers.

2.5.5. Port

Port structures are used to represent the types of rectangles. There are two types of ports: derived and basic ports. A basic port is represented by an integer, whereas a derived port is represented by a list of integers. Ports have the *add* method, which unites two ports and returns a derived port. A derived port may be generated by adding two basic port types, e.g. *SENSITIVE* and *GATE*. The operator "+" is overloaded and works with the ports and the operator ">" is used to check if a port contains another one. Also, the operator "==" may be used to check if two ports are composed of the same basic ports. There are several methods in the *Rectangle* class, related to ports. A port type is assigned to a rectangle with the *setPort* method and the assigned port type is read by the *getPort* method.

```
// Assume rule r1 is the minimum metal enclosure of contact
Rule r1 = (Rule) 1e-6;
// r2 metal enclosure of contact at the end of metal
Rule r2 = (Rule) 1.1e-6 extends r1;
```

Figure 2.17. Instances of *Rule* Class May Have Hierarchy.

2.5.6. Rule

Rule is the data structure used to handle the design rules. The value of a rule is assigned when it is constructed and it cannot be changed. The value of a *Rule* is in *double* type and may be read by calling its *getValue* method. A rule may extend, in other words, may override other rules. Extended rules are assigned an ordinal which is higher than its base. When two or more rules are applicable, the one with the higher ordinal is applied. The `extends` keyword is used to assign a higher ordinal. Note that, this does not necessarily mean that the rule, having the highest value, wins, but the rule having the highest ordinal wins.

A sample code is given in Figure 2.17. For this code, assume the first rule *r1* defines the required minimum space between the *metal* and the *contact* layers. However, if the *contact* layer is placed at the end of the *metal* layer, it may be necessary to apply another rule *r2*. For the cases where both rules, r1 and r2, are valid, the second one will override the first one.

2.5.7. Net

Net is an interface and is used to handle the connectivity between the layout components, where components which refer to the same *Net* must be connected. Every *Net* object has a different identifier, unless it is connected to another net. The *Net* interface, contains the method `connect`, which equates the identifiers of two nets, thus

connects them. The *Net* structures override the operator "==", which checks if the two nets are connected.

2.5.8. Solver

Solver is an interface which contains *solve* and *set* methods. The *solve* method accepts constraints and objectives as inputs and tries to solve the constraint problem. The solver, is automatically called at the end of an LDS block. Two implementations of the solver are currently available: a linear programming solver, which is the default solver, and a convex programming solver. The user may change the solver by means of the `extends` keyword. The linear programming solver uses QSopt [59] package, whereas the convex programming solver uses the MOSEK [58] package.

2.5.9. Transformations

Transformations are handled with the *Transformation* class. Affine transformation -rotation, vertical/horizontal flipping and translation- are natively supported in LDS. Each *Transformation* class stores a transformation matrix A:

$$A = \begin{bmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

where a_{mn} are constants. A counter clockwise rotation multiplies the following matrix:

$$R(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

with the stored transformation matrix A where θ is the rotation angle and *Rectangle* class in LDS is allowed to have θ values which are multiple of 90 degrees. Otherwise, the methods: `left`, `right`, `top` and `bottom`, defined in the *Rectangle* class, are going

to be meaningless. However, this does not necessarily limit the usage of any angle rotation for other shapes. The user is free to define his/her own shapes.

Vertical flipping is applied by multiplying the V matrix, defined as:

$$V = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

with the transformation matrix A. Similarly, a horizontal flipping is done by multiplying the matrix of horizontal flipping H that is defined as:

$$H = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

with the transformation matrix A. A translation is done by multiplying the following translation matrix T:

$$T(t_x, t_y) = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.5}$$

with the transformation matrix A, where t_x and t_y are the amounts of shifts in the horizontal and vertical directions, respectively.

A *Transformation* class also contains functions to transform a point. If a point with the coordinates (x, y) is transformed, the new coordinates are (x', y'), calculated by the following operation:

$$\begin{bmatrix} x' \\ y' \\ 0 \end{bmatrix} = A \cdot \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} \tag{2.6}$$

Table 2.7. Functions for Transformation.

Function	Description
inverse()	A^{-1}
rotate(double angle)	$R(\theta) \cdot A$, θ is the angle.
transform(Transformation B)	$B \cdot A$
translate(double tx, double ty)	$T(t_x, t_y) \cdot A$, t_x and t_y are the amount of shifts
x(double x, double y)	$A \cdot [\ x \quad y \quad 0\ ]^T$, x component
xflip()	$H \cdot A$
x(double x, double y)	$A \cdot [\ x \quad y \quad 0\ ]^T$, y component
yflip()	$V \cdot A$

where the matrix A is the transformation matrix. As an example, if a translation T, rotation R and vertical flipping V is applied to a transformation A, in the given order, the resulting transformation matrix A' is going to be:

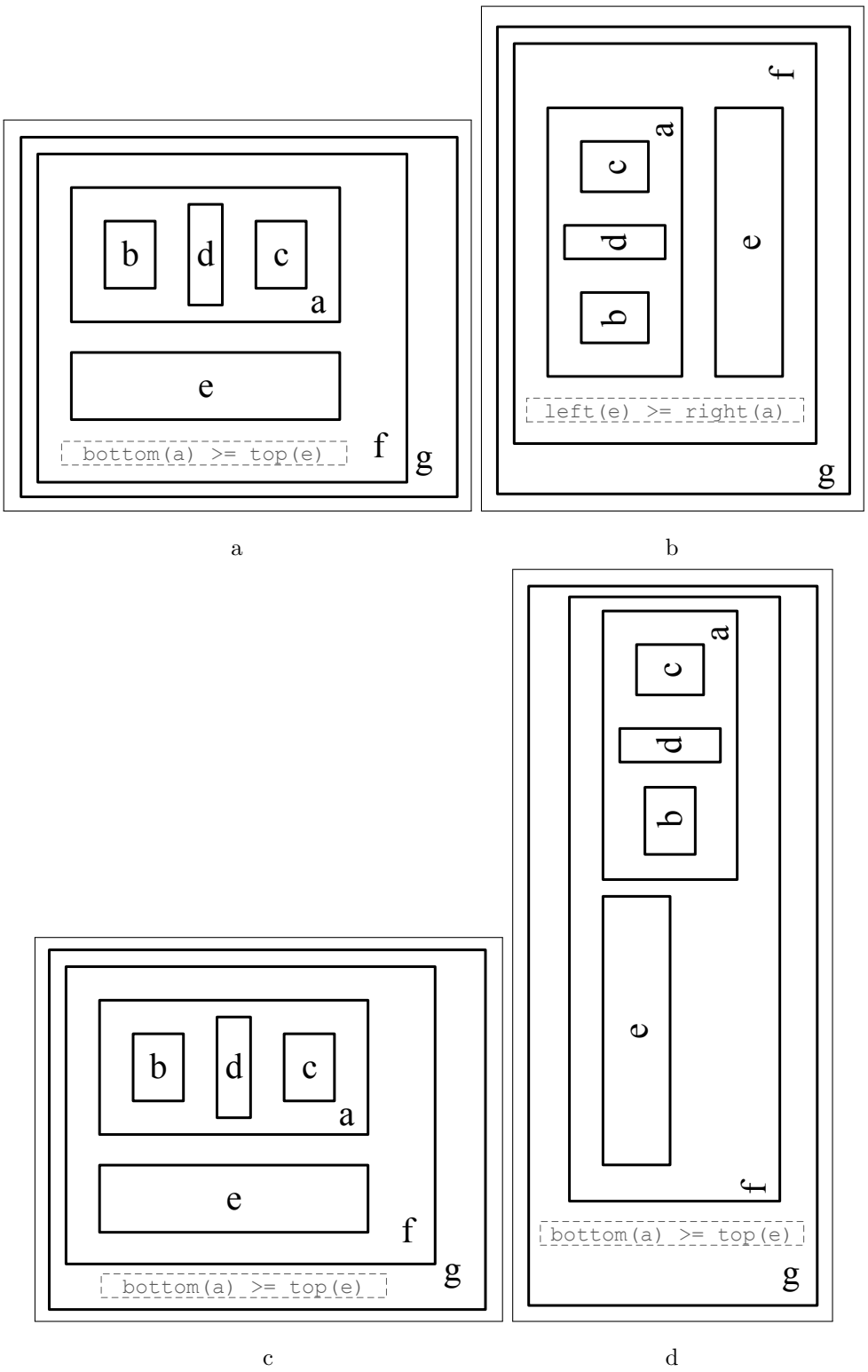
$$A' = V \cdot R \cdot T \cdot A \tag{2.7}$$

The functions to perform transformations are listed in Table 2.7.

Transformation stored in a rectangle is with respect to the substrate frame. For instance, assume that rectangle A is a child of rectangle B, and it is flipped vertically. Then, the new transformation of the rectangle A $T_A^{'}$ is going to be:

$$T_A^{'} = T_B \cdot V \cdot T_B^{-1} \cdot T_A \tag{2.8}$$

where the transformations T_A and T_B are the transformation of the rectangles A and B, before the flipping transformation, respectively. The transformation T_A is not the transformation with respect to its parent frame, but the transformation with respect to the substrate frame.



a b c d

Figure 2.18. Transformations on Constraints (a) Constraint Defined in Rectangle *f*, (b) Rectangle *f* Rotated, Constraint Transformed, (c) Constraint Defined in Rectangle *g*, (b) Rectangle *f* Rotated, Constraint Not Transformed.

The constraints are also transformed; they are assumed to be in the frame of the rectangle in which they are defined. Consider the cases in Figure 2.18. In Figure 2.18a, a constraint "`bottom(a) >= top(e)`" is defined in rectangle *f*, which forces rectangle *a* to be placed above rectangle *e*. If the rectangle *f* is rotated 90 degrees, counter clockwise, the constraint is also transformed and now, it forces rectangle *e* to be placed to the right of rectangle *a*. The constraint is transformed as "`left(e) >= right(a)`", shown in Figure 2.18b. However, if the constraint "`bottom(a) >= top(e)`" is defined in rectangle *g* as shown in Figure 2.18c, then the resulting placement is given in Figure 2.18d. Although, the rectangle *f* is rotated, the constraint is not affected which is defined in rectangle *g*. Note that a rectangle may only be referenced in itself or in its parents.

2.6. Code Generator

A template code in LDS is converted to a sequential program by the LDS compiler. A template in LDS may be instanced by running the compiled program. Note that, generating a sequential program from an LDS code is analogous to generating a machine code from a C program. Any program may be written by using directly machine code; however, writing a machine code is much harder and prone to more errors.

The LDS compiler is used to build an executable code from an LDS code. The compiler parses an LDS code and generates a Java program. Necessary code and data structures to handle the constraints and the linear program are added into the generated code. Each `lds` block defines a new constraint satisfaction problem and at the end of each block, the necessary code to solve the linear program is added. Note also that an LDS template may be compiled for different technologies.

2.6.1. Lexer and Parser

A lexer recognizes the identifiers, the keywords and the symbols, where as a parser recognizes the syntax. The parser and lexer code for LDS is coded in ANTLR4 [61]. Since LDS is an extension to Java [60], the parser and the lexer must recognize the

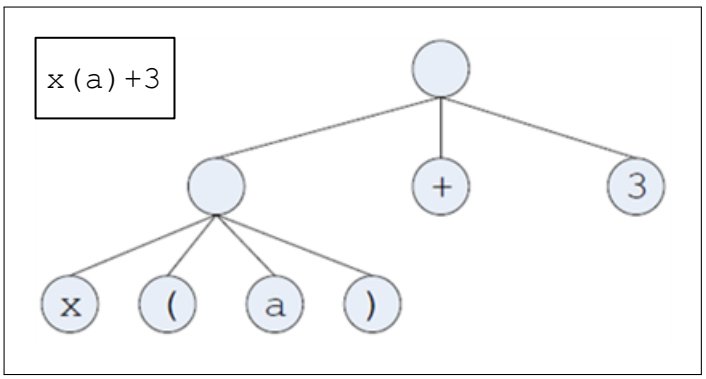


Figure 2.19. Abstract Syntax Tree for the Expression `x(a) + 3`.

Java language as well as LDS. The lexer recognizes Java keywords and also recognizes `layout`, `lds`, `constrain`, `minimize`, `maximize`, `objective`, `tech` and the `param` keywords. The parser is capable of parsing the LDS syntax in addition to the Java syntax.

The ANTLR4 is used to synthesize the lexer and the parser codes. The synthesized code is capable of constructing an abstract syntax tree (AST). A sample tree is given in Figure 2.19 where each part of the expression corresponds to a node in the tree. A part may be any meaningful character sequence: e.g. an identifier, a parenthesis, a numerical value, an operator.

The constructed AST is used to synthesize a Java program. The synthesis is done by expressing the tree in Java, and making use of the LDS library. As an example the AST in Figure 2.19 is converted into Java code as: "`lds.add(x(a),3)`". The functionality of the LDS library is presented in the next section.

2.6.2. Library

Code blocks calling the methods in the LDS library are automatically added by the LDS compiler. The following operations are supported by the LDS library:

- Addition ("+") and subtraction ("-") operations are supported for any combination of two of the following types: predefined numeric types of Java, numeric types of LDS, *Term* and *Rule* types.
- Multiplication ("*") and division ("/") are supported for any combination of two of the following types: numeric types of Java, numeric types of LDS and the *Rule* type.
- *Term* structures may be multiplied with or divided by a numeric of Java, numeric types of LDS and the *Rule* type.
- A numeric type of LDS may be explicitly casted to a numeric type of Java or to a *Term* type. Similarly, a *Term* type may be explicitly casted to a numeric type of LDS. Also casting from a numeric type of Java to a numeric type of LDS and to the *Term* type are supported.
- Overloaded operators "==", ">=", "<=", ">" and "<" may be used for numeric types, *Layer* type, *Net* type and the *Port* type.

Table 2.8 lists some of the operations and the corresponding methods defined in the LDS library. In addition to the listed ones, methods to manage constraints and objectives are also defined in the LDS library.

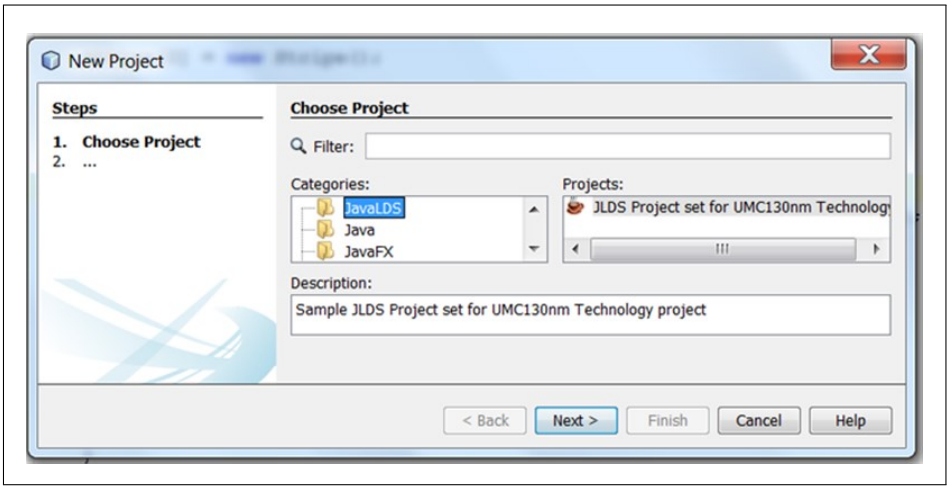
2.7. Integrated Development Environment

2.7.1. Project Manager

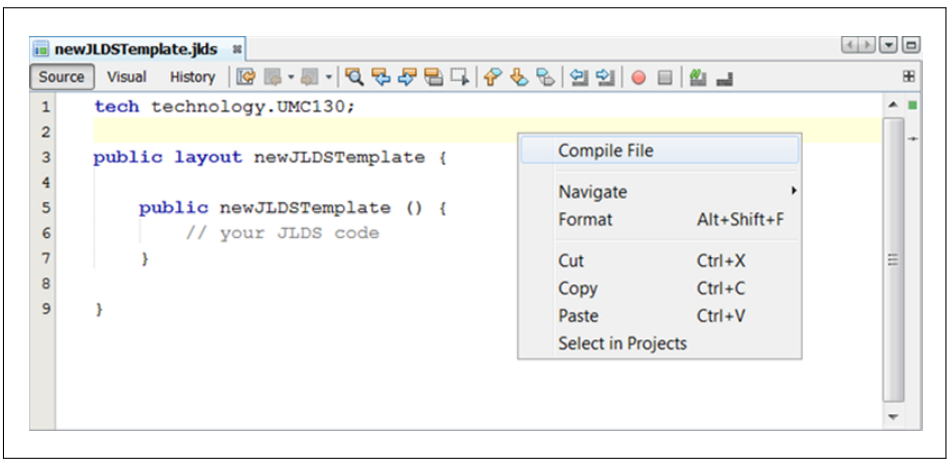
A new project type, *JLDS Project*, is added into the NetBeans [62] platform, which is designed for managing Java [60] projects and has extension capabilities via plug-ins. A screen shot of the new project window, where a new *JLDS UMC130nm* project is selected, is shown Figure 2.20a. When the project is generated, the LDS library, libraries for optimizers, and the technology library for the *UMC130nm* process are automatically added. Also a new file type, *JLDS file*, is added into the NetBeans platform. The *JLDS* editor, capable of keyword highlighting and syntax checking is registered with this file type. Thus, the *JLDS* editor is activated for any file with the extension *jlds*. If a new *JLDS* file is selected from the new file dialog, a wizard shows

Table 2.8. Methods Defined in the LDS Library.

Operation	Methods
a + b	`lds.add ( a , b )`
a - b	`lds.sub ( a , b )`
a * b	`lds.mult ( a , b )`
a / b	`lds.div ( a , b )`
a & b	`lds.and ( a , b )`
a | b	`lds.or ( a , b )`
a ^ b	`lds.xor ( a , b )`
a % b	`lds.mod ( a , b )`
a && b	`lds.and2 ( a , b )`
a || b	`lds.or2 ( a , b )`
a == b	`lds.eq ( a , b )`
a != b	`lds.neq ( a , b )`
a < b	`lds.lt ( a , b )`
a > b	`lds.gt ( a , b )`
a ≤ b	`lds.le ( a , b )`
a ≥ b	`lds.ge ( a , b )`
- a	`lds.neg ( a )`
~ a	`lds.inv ( a )`
! a	`lds.not ( a )`
(a) b	`lds.explicit ( a , b )`



a



b

Figure 2.20. The *JLDS* Project (a) Generating New *JLDS UMC130nm* Project, (b) Sample Code Automatically Added and New Menu Item added.

up. When the file name and location is specified, a startup code is automatically added as shown in Fgiure 2.20b.

A menu item, named *Compile File*, is added to the pop-up menu, and the menu pops up when the right mouse button is clicked, as shown in Figure 2.20b. When this item is selected, a Java source code is synthesized from the LDS source code. The generated source code is stored under the *generated code* folder and is visible by the other Java classes and LDS codes. To check the syntax of the LDS code in the editor, a Java code is synthesized and stored in the temporary memory, not in the file system. However, when the LDS file is compiled, a file is generated and stored in the file system.

2.7.2. Keyword Highlighting

The LDS language is integrated with Java [60] and an editor is designed to be used with the new language. The editor, makes use of a Lexer to highlight the keywords, which is synthesized by the ANTLR4 [61] tool. All Java and LDS keywords are made bold and colored in blue. Also, strings and comments are detected by the Lexer and these are highlighted in the editor. The editor inherits from the basic editor in the NetBeans Platform [62]. The LDS editor is a plug-in for this platform and it is installed by default, thus there is no need to install it in every startup. A screen shot is given in Figure 2.21a, where the keywords, a string and a comment are highlighted.

2.7.3. Syntax Checking

To ease code development, syntax errors are underlined in the code editor. First, the syntax is checked by the parser, synthesized by ANTLR4, and the errors are reported. However, this check is limited to syntax, and types of the variables or the return types of the methods are not checked. Then, the Java code is synthesized and the Java Compiler is called to check for errors. However, the reported errors are the ones in the generated Java file, not the errors in the original LDS code. A map is constructed, which maps the errors back to the original LDS code. Then, the editor underlines these errors. Note that all code synthesis is performed in the memory and the file system is not used. The syntax checker is also packed as a plug-in for the NetBeans platform and installed. The screen shot in Figure 2.21b, shows the errors in a sample LDS code. Descriptions for the errors are also added. If the mouse points to the underlined erroneous code, the error message is displayed. These error messages are copied from the diagnostic information generated by the Java Compiler.

```
1   package technology.UMC130.devices;
2
3   import java.util.Arrays;
4
5   tech technology.UMC130;
6
7   public layout DiffPair {
8
9       param {
10          int m = 5;
11          double width = 2e-6, length = 0.68e-6;
12          Net drain1 = new Net(), gate1 = new Net(), source = new Net()
13                  , gate2 = new Net(), drain2 = new Net(), bulk = new Net();
14          Transistor.Type type = Transistor.Type.NMOS;
15      }
16
17      public static Port ROW = Trouter.ROW;
18
19      public DiffPair (...) {
20          String demo = "demo";
21          // comment
```

a

```
34          for (int i = 1; i < param.ns; i++) {
35              stripe[i] = new Stripe();
36
37              constrain bottom(stripoe[i-1]) - top(stripe[i])
38                      >= tech.minspace(tech.POLY, tech.POLY, null, param.width);
39
40              constrain left(stripe[i-1]) == left(stripe[i]);
41
42              Rectangle metal = (new Rectangle(this)).setLayer(tech.ME1);
43
44              metal.x.eeqclose(stripe[i-1].getPort(i % 2));
45              constrain bottom(stripe[i-1].getPort(i%2)) == top(metal);
46              constrain top(stripe[i].getPort(i%2)) == bottom(metal);
47          }
```

b

Figure 2.21. The *JLDS* Editor (a) Keywords, Strings and Comments Highlighted, (b) Syntax Errors Reported.

3. TEMPLATE SYNTHESIS

This chapter presents the *Capture* tool, which synthesizes a template from an expert drawn layout. Also, in this chapter, algorithms are given to convert topological representation to LDS constraints. Moreover, device generator coded in LDS are presented.

3.1. Capturing a Template

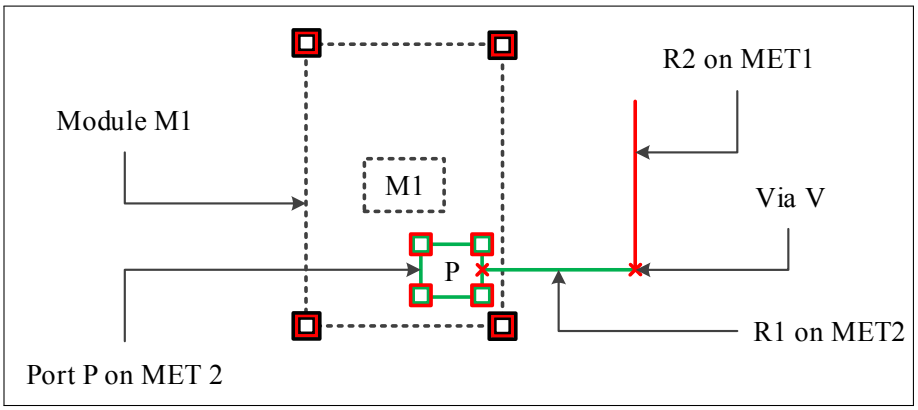
To implement a layout-aware synthesis loop as proposed in Chapter 1, either a pre-coded or synthesized template is required. Sometimes, it is convenient to benefit from the expert knowledge and excellent performance of existing layouts. An example may be to use a layout drawn for parasitic matching by an expert. In this case, the template must be either manually coded or extracted from the pre-drawn layout. Extracting a template from a pre-drawn layout is obviously going to be easier than manually generating a new code from scratch. The Capture tool is designed to assist the extraction of a template from a conventional pre-drawn layout.

3.1.1. Methodology

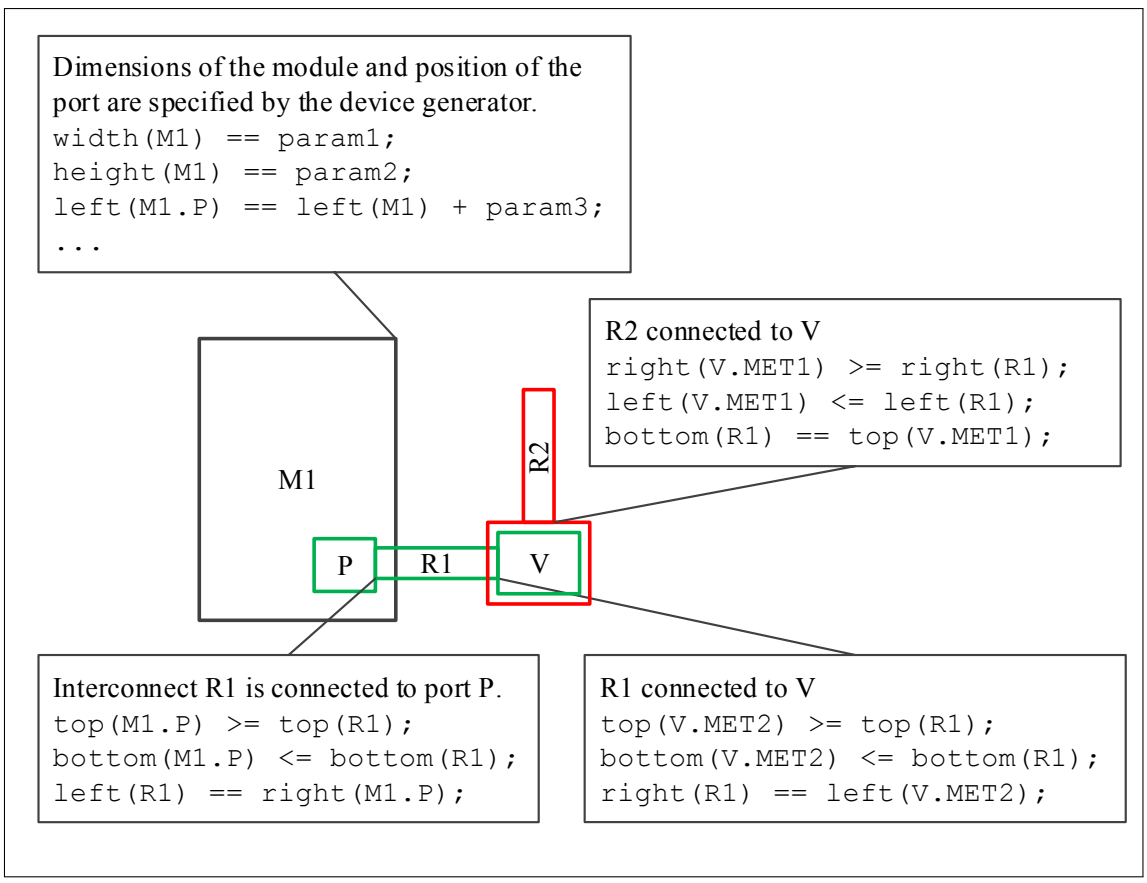
A template is constructed in two steps: component extraction and code generation.

(i) Component extraction is composed of three phases: module extraction, port extraction and connectivity extraction. A sample extraction is depicted in Figure 3.1a. Module extraction is carried out by extracting initial positions of the modules, their sizes, and names. After extracting the modules, the user specifies the ports of these modules. Finally, interconnect paths are defined and the user specifies a style for each path.

(ii) LDS code is generated using the extracted data. First, necessary code is generated to set all the rectangle positions to their initial values in the original layout.



a



b

Figure 3.1. Capture Extracts the Template in Two Steps (a) Module, Port, and Connectivity Extracted via a GUI, (b) Synthesized LDS Code.

Then, LDS constraints are added to constrain the positions of the ports in the extracted modules. Also, paths are converted to LDS by adding alignment and abutment constraints between interconnects. The routing widths are parametric and minimum width rules may be used. Some of the generated constraints are depicted in Figure 3.1b for the extracted data in Figure 3.1a. Required spacing for DRC is automatically added during template instancing.

3.1.2. Extracted Templates

The Capture tool has been tested with different analog circuits. Two examples are presented in Figure 3.2, namely: a fully differential amplifier circuit in Figure 3.2a and a voltage doubler circuit in Figure 3.2b. First, the data structures visualized in Figure 3.2c and in Figure 3.2d, respectively, are captured. Then, layout templates are generated from the extracted data structures and used to instance sample layouts. An instance of the template generated for the fully differential amplifier circuit is shown in Figure 3.2e. Similarly, a sample layout is instanced for the voltage doubler circuit and shown in Figure 3.2f.

The Capture tool generates an LDS template from an extracted data structure. This data structure may also be generated by a placement or a router tool and this shows the compatibility of LDS with other EDA tools.

3.2. Topological Representations

By modifying topological representations [26], new non-overlapping layouts may be found. Thus, they are commonly used with placement tools. However, topological representations may only represent a single layer. It is possible to extend these representations by using LDS, where different representations may be merged. Common representations and algorithms to convert these into LDS constraints are given, below.

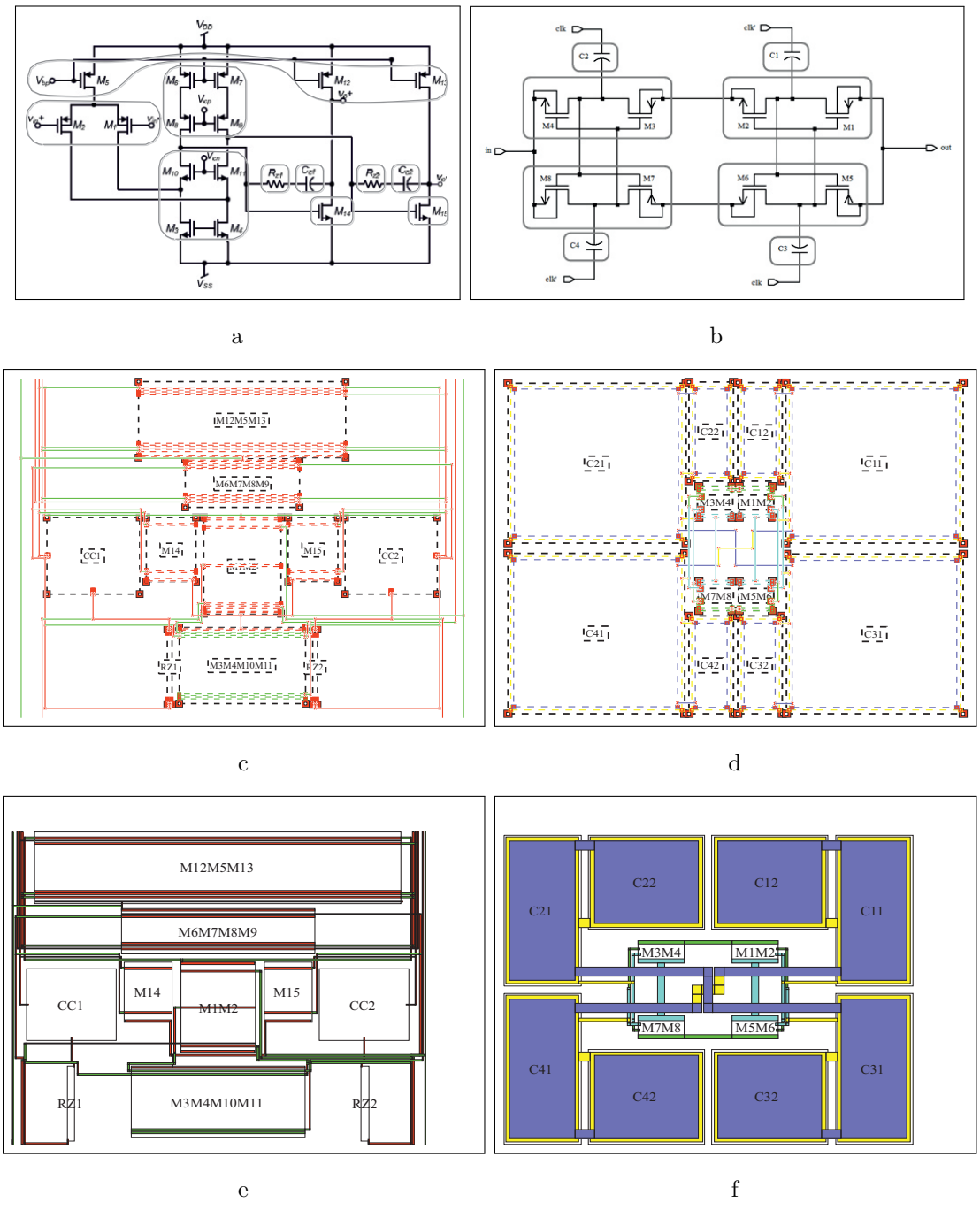


Figure 3.2. Capture Tool (a) Schematic of Amplifier (b) Schematic of Voltage Doubler, (c) Extracted Data for (a), and (d) Extracted Data for (b), (e) Instance of the Template for (c), and (f) Instance of the Template for (d).

3.2.1. Slicing Tree Representation

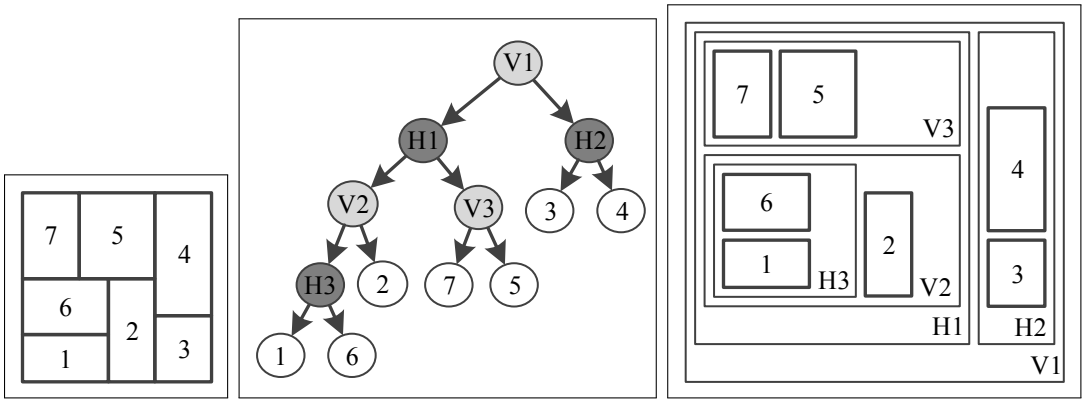
The Slicing Tree [27] is a well-known topological representation. It slices the layout continuously into horizontal and vertical pieces till a single rectangle is obtained. A slicing placement and the corresponding graph representation are given in Figure 3.3a and Figure 3.3b, respectively.

The algorithm in Figure 3.3d, written in LDS, generates necessary constraints for a slicing tree. Calling the `slicing2LDS` function with the tree in Figure 3.3b, this algorithm combines nodes into rectangles to obtain the structure shown in Figure 3.3c. For instance, the sixth rectangle must be placed above the first rectangle, a placement constraint is going to be added between these two, and they are going to be placed into a rectangle named *H3*. Then, the second rectangle is going to be placed to the right of the new rectangle *H3*. Similarly, rectangles and constraints are going to be added for each vertical and horizontal node in the slicing tree.

3.2.2. Sequence Pair Representation

The Sequence Pair (SP) [28] is composed of two name sequences. It may represent non-slicing layouts, which obviously cannot be represented by a Slicing Tree. A simple non-slicing layout is given in Figure 3.4a, which cannot be represented by the Slicing Tree representation. With SP, this layout is represented by positive and negative name sequences as $(\Gamma_+, \Gamma_-) = (ecadb, cbaed)$. If a rectangle x is above a rectangle y, the order of x is going to be greater than the order of y in Γ_+ and the order of x will be smaller than that of y in Γ_-. If the rectangle y is to the right of rectangle x, then the order of y is going to be greater than that of x in both sequences.

The algorithm in Figure 3.4c, written in LDS, generates constraints for the input sequence pair. The structure *element*, defined in this figure, represents a rectangle and its order in the positive and negative sequences. The function *sp2LDS* adds placement constraints indicated by the arrows in Figure 3.4b. This algorithm scans the positive and negative sequences and finds the closest rectangles above and to the right of a



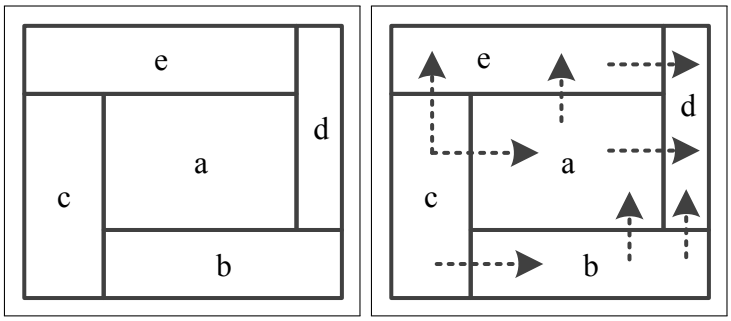
a b c

```
void slicing2LDS(Node node)
{
        // return if end of tree
        if (node.type != "H" &&| node.type != "V")
                return;
        // Go deeper in the tree, left child first
        slicing2LDS (node.leftChild);
        slicing2LDS (node.rightChild);
        // if H place above, if V place after
        if (node.type == "H")
                bottom(node.rightChild.boundary) >=
                        top(node.lefChild.boundary);
        else if (node.type == "V")
                left(node.rightChild.boundary) >=
                        right(node.leftChild.boundary);
        // place right and left children into parent
        placeinto(node.rightChild, node);
        placeinto(node.leftChild, node);
}

void placeinto(Node child, Node parent)
{
        bottom(parent.boundary) <= bottom(child.boundary);
        top(parent.boundary) >= top(child.boundary);
        left(parent.boundary) <= left(child.boundary);
        right(parent.boundary) >= right(child.boundary);
}
```

d

Figure 3.3. Slicing Tree to LDS (a) Placement, (b) Slicing Tree Representation for (a), (c) Rectangles Generated, (d) Algorithm to Generate LDS Constraints for a Slicing Tree Representation.



a b

```
// element holds the rectangle
// and its position in gamma_plus and gamma_minus
struct element{
        Rectangle rec;
        int pos, neg;
}

void sp2LDS(element[] pos, element[] neg) {
        // for each element e
        for (n = 1; n < pos.length+1; n++) {
                // find the closest elements below e
                nmax = pos.length+1;
                for (m = pos[n].neg-1; m >= 0; m--) {
                        if (neg[m].pos > n && neg[m].pos < nmax) {
                                nmax = neg[m].pos;
                                // add constraint
                                bottom(pos[n].rec) >=
                                        top(pos[nmax].rec);
                        }
                }
                // find the closest elements right to e
                nmax = pos.length+1;
                for (m = pos[n].neg+1; m <= neg.length; m++) {
                        if (neg[m].pos > n && neg[m].pos < nmax) {
                                nmax = neg[m].pos;
                                // add constraint
                                right(pos[n].rec) <=
                                        left(pos[nmax].rec);
                        }
                }
        }
}
```

c

Figure 3.4. Sequence Pair to LDS (a) Placement, (b) Extracted Constraints, (c) Algorithm to Generate LDS Constraints from a Sequence Pair Representation.

rectangle. For instance, constraints from the rectangle c to the rectangles a, b and e are generated. No constraint is added from c to d because there already exists a constraint to rectangle a, and a is closer than d.

3.2.3. B*-Tree Representation

B*-tree [29] is a binary tree, and nodes in the tree represent rectangles in the layout. A left child of a node, representing a rectangle, in the tree corresponds to a rectangle at its right side and a right child corresponds to a rectangle above it. A sample placement and the corresponding B*-tree are given in Figure 3.5.

The algorithm in Figure 3.5d adds two rectangles for each node, one for the left child and one for right child. The rectangle corresponding to the right child is placed above the rectangle represented by the parent node and the one corresponding to the left child is placed to the right of the rectangle. This process is continued recursively, till the bottom node is reached. The rectangles generated for the tree in Figure 3.5b are shown in Figure 3.5c. Finally, the dummy rectangles are removed and the layout is compacted, first in the vertical direction, then in the horizontal direction.

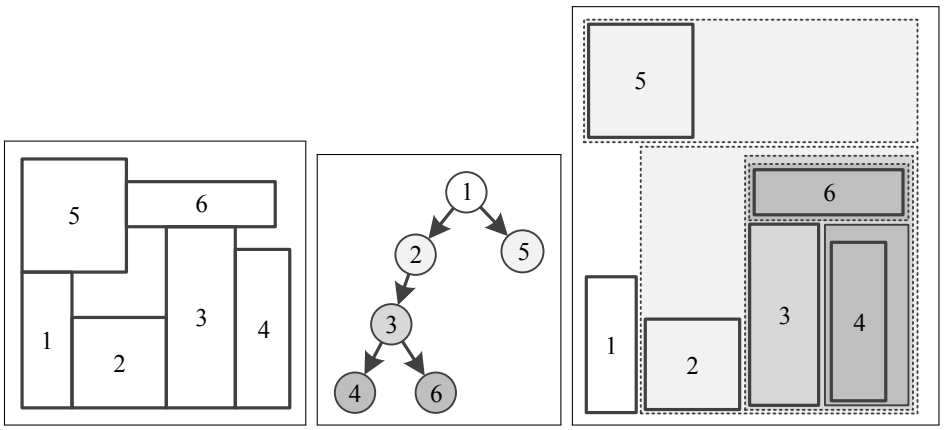
3.3. Device Generators

Device generators for the *UMC130nm* are coded in LDS.

3.3.1. Guard-ring Generator

A guard-ring is used to isolate a device from the surrounding noise sources. Two instances of the guard-ring are shown in Figure 3.6. A guard-ring has three input parameters:

- m : It specifies the number of vias and its default values is set to one.

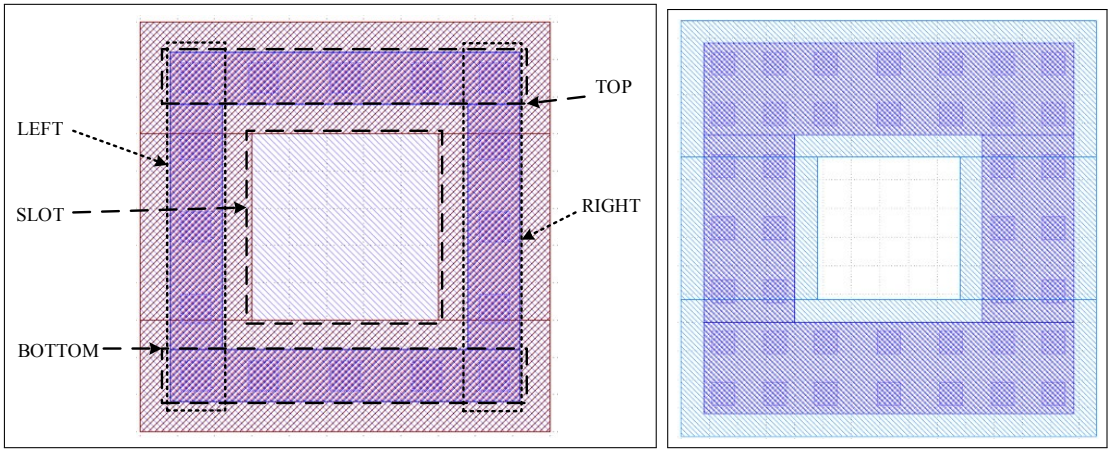


a b c

```
void bst2LDS (Node n, Rectangle rec){
        // Place n in rec
        placeinto(n.boundary,rec);
        Rectangle rec1 = null, rec2 = null;
        if (n.rightChild != null) {
                rec1 = new Rectangle();
                // Recursively go deeper
                bst2LDS(n.rightChild, rec1);
                // Place rec1 right to n
                left(rec1) >= right(n.rightChild.boundary);
                // Place rec1 into rec
                placeinto(rec1, rec);
        } if (n.leftChild != null) {
                rec2 = new Rectangle();
                // Recursively go deeper
                bst2LDS(n.rightChild, rec2);
                // Place rec2 above n
                bottom(rec2) >= top(n.leftChild.boundary);
                // Place rec2 into rec
                placeinto(rec2, rec);
                if (rec1 != null) {
                        bottom(rec2) >= top(rec1);
                }
        }
}
void placeinto(Rectangle child, Rectangle parent) {
        bottom(parent) <= bottom(child);
        top(parent) >= top(child);
        left(parent) <= left(child);
        right(parent) >= right(child);
}
```

d

Figure 3.5. B*-Tree to LDS (a) Placement, (b) B*-Tree (Horizontal) Representation for (a), (c) Rectangles Generated and Placed for Each Node, (d) Algorithm to Constrain a Placement for B*-Tree Representation.



a b

Figure 3.6. Guard-Ring (a) for *n-well* with $m = 1$, (b) for *p-substrate* with $m = 2$. Ports Shown in (a).

- *well* : The default value for this parameter is null, and if not set the guard-ring is a p-substrate guard-ring. If the parameter *well* is set the guard-ring is an n-well guard-ring.
- *net* : It specifies the connectivity information for the ports of the guard-ring. There are five types of ports for this device, namely: *RIGHT*, *TOP*, *LEFT*, *BOTTOM* and, *SLOT*. The ports *RIGHT*, *TOP*, *LEFT* and, *BOTTOM* are used to represent the connections and any routing path may be connected to these ports. The port *SLOT* specifies the internal region of the guard-ring. Devices such as transistors, capacitors or resistors may be constrained to be in this port, thus these devices will be protected by the guard-ring.

3.3.2. Transistor Generator

The transistor generator outputs a MOSFET given the type (NMOS or PMOS), width and, the length of the transistor. Instances of NMOS and PMOS transistors are shown in Figure 3.7. The parameters of the MOSFET generators are:

- W : It specifies the width of the transistor.
- L : It sets the lengths of the fingers.

- *net* : This parameter sets the nets for the source, gate and, the drain regions.
- *bulk* : It specifies the net for the bulk.
- *guard* : It accepts *true* or *false* values. If it is set to *true*, a guard-ring is surrounds the transistor.
- *type* : This parameters accepts *NMOS* or *PMOS* values and specifies the type of the MOSFET.
- *metalwidth* : This parameters is used to set the width of the source/drain regions.

Ports of a transistor have the types *SD*, *GATE*, *PLUS*, *DIFF* and, *GUARD*. The port types *SD* and *GATE* are used for the source/drain and gate regions, respectively. The port *PLUS* and *DIFF* represent the pplus/nplus regions and the diffusion region of the transistor. If the parameter *guard* is set to true, the port *GUARD* points the guard-ring.

3.3.3. Routed Transistor Generator

This device generator accepts a transistor and connectivity information as inputs and outputs a transistor where its fingers are connected in accordance to the input data. Examples to routed NMOS and PMOS devices are shown in Figure 3.8. Each vertical connection has the port type *ROW*, as indicated in Figure 3.8. This device generator accepts the following parameters:

- *m* : This parameter specifies the number of vias and its default values is set to one.
- *layer* : This is an array and specifies the layers of the *ROW*s.
- *above* : This parameters is used to input the nets of the *ROW*s above the transistor. The lower the index the closer the *ROW* to the transistor is.
- *below* : Similar to the *above* parameter, this parameters is used to input the nets of the *ROW*s below the transistor. Again, the lower the index the closer the *ROW* to the transistor is.
- *mosfet* : This parameter specifies the transistor to be routed.

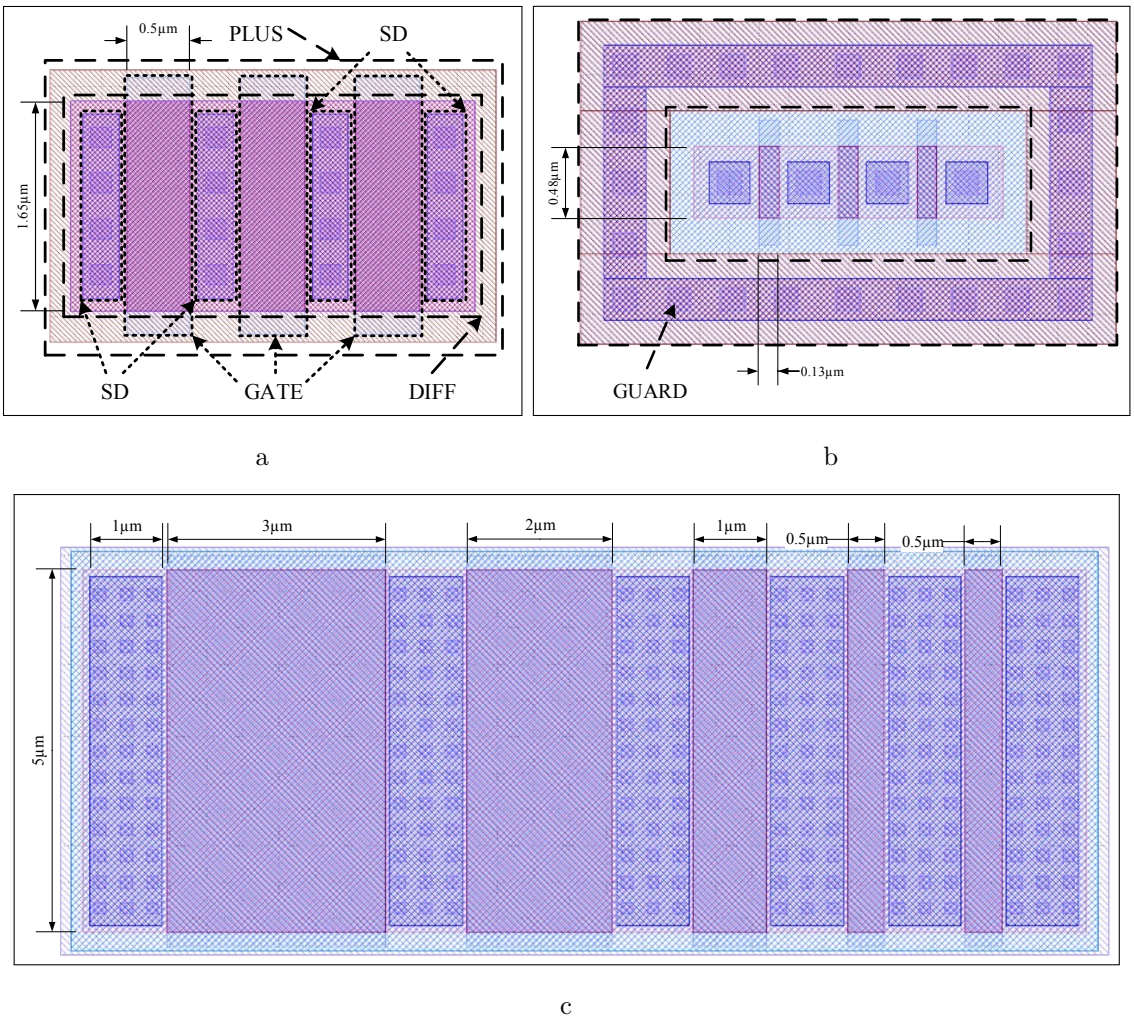
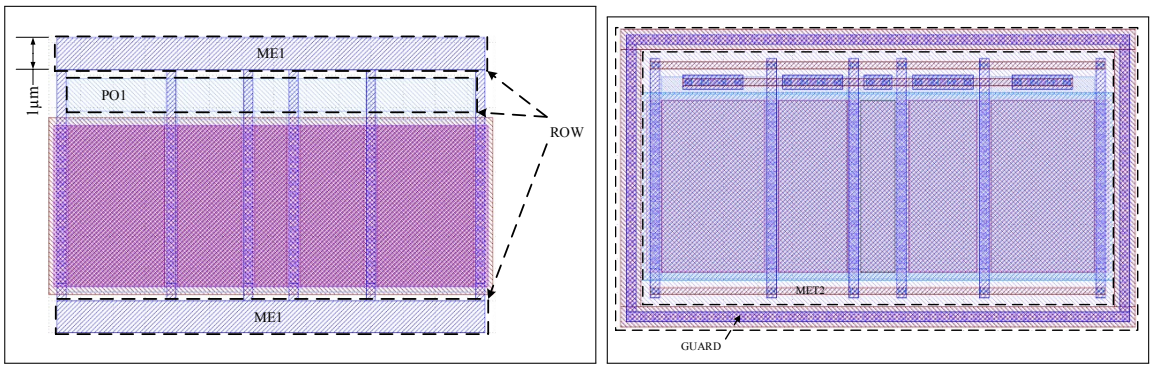


Figure 3.7. MOSFET (a) NMOS with $m = 3$, $L = 0.13\mu m$ and $W = 1.65\mu m$, (b) PMOS with Guard-Ring and $m = 3$, $L = 0.13\mu m$, $W = 0.48\mu m$, (c) PMOS with $L = \{3\mu m, 2\mu m, 1\mu m, 0.5\mu m, 0.5\mu m\}$, $W = 5\mu m$, $metalwidth = 1\mu m$.



a b

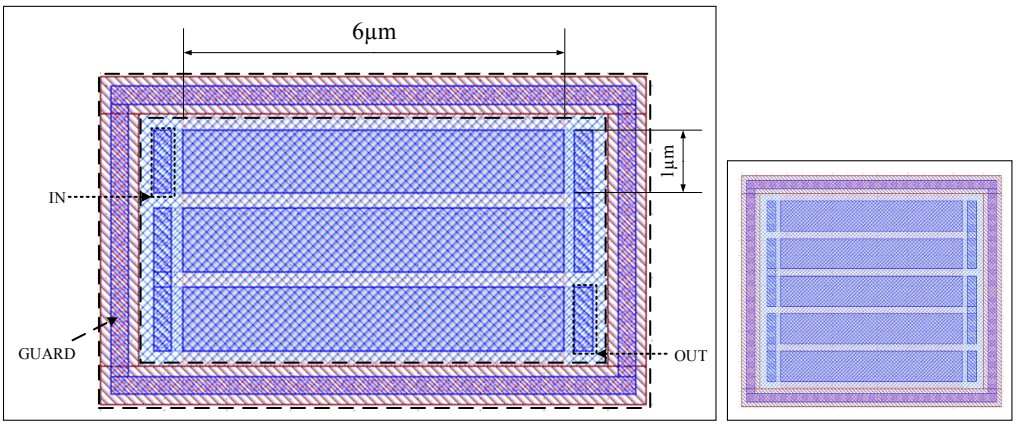
Figure 3.8. MOSFET with Source-Drain-Gate Connections (a) NMOS with $width = 1\mu m$ and $layer = \{tech.PO1, tech.MET1, tech.MET1\}$, (b) PMOS with Guard-Ring and All Rows in *MET2*.

- *width* : It specifies the widths of the *ROW*s, by default it is set to minimum design rule correct metal width.

3.3.4. Resistor

The resistor generator outputs a polysilicon resistor in an n-well, named as *NSPRES* in *UMC 130nm* technology. Sample instances of the resistor generator are shown in Figure 3.9. A resistor has the following input parameters:

- *width* : This parameter specifies the thickness of the polysilicon strips.
- *length* : It is used to input the lengths of the polysilicon strips. This parameters does not specify the complete length of the polysilicon, but the length of a single polysilicon strip.
- *ns* : This parameter is used to specify the number of polysilicon strips.
- *in* : It specifies the net of the input port of the resistor.
- *out* : This parameter specifies the net of the output port of the resistor.
- *bulk* : It specifies the net of the n-well.



a b

Figure 3.9. Resistor (a) $ns = 3$, $length = 6\mu m$ and $width = 1\mu m$ (b) $ns = 5$.

A resistor has *IN*, *OUT* and, *GUARD* typed ports. The ports *IN* and *OUT* correspond to the two electrical connections of the resistor. The *GUARD* port handles the guard-ring around the resistor and it must be biased by assigning a net to the **bulk** parameter.

3.3.5. Capacitor

This device generator outputs a metal-insulator-metal (MIM) capacitor in *UMC 130nm* technology. The capacitor plates are the aluminum pad layer and the last metal layer. An instance is shown in Figure 3.10. This device generator accepts the following parameters:

- *W* : This parameter specifies the width of the capacitor plate.
- *H* : It is used to input the height of the capacitor plate.
- *in* : It specifies the net of the input port of the capacitor.
- *out* : This parameter specifies the net of the output port of the capacitor.

The capacitor has two types of ports. The *IN* and the *OUT* typed ports points to the electrical ports of the capacitor as shown in Figure 3.10.

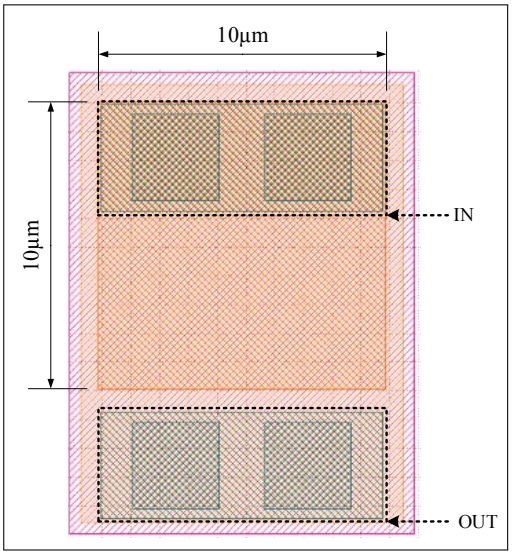


Figure 3.10. Capacitor with $W = 10\mu m$ and $H = 10\mu m$, Ports IN and OUT Shown.

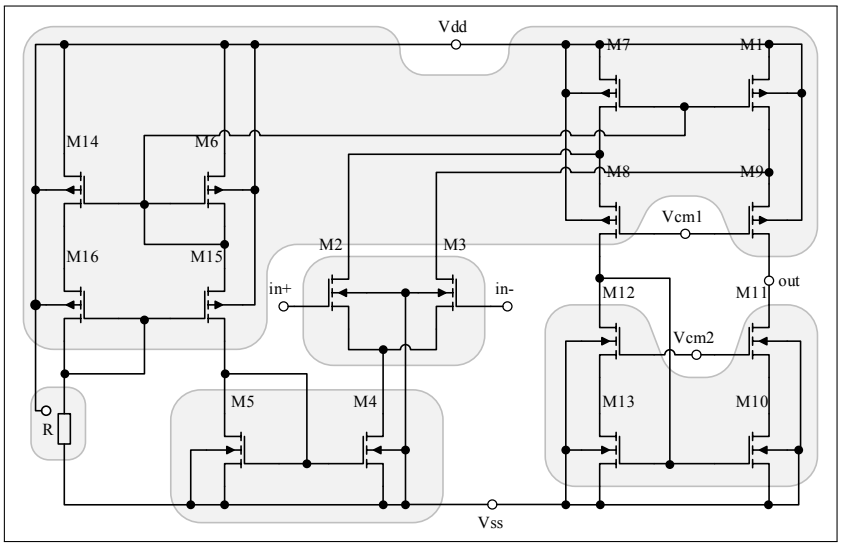


Figure 3.11. Schematic of the Folded Amplifier Circuit.

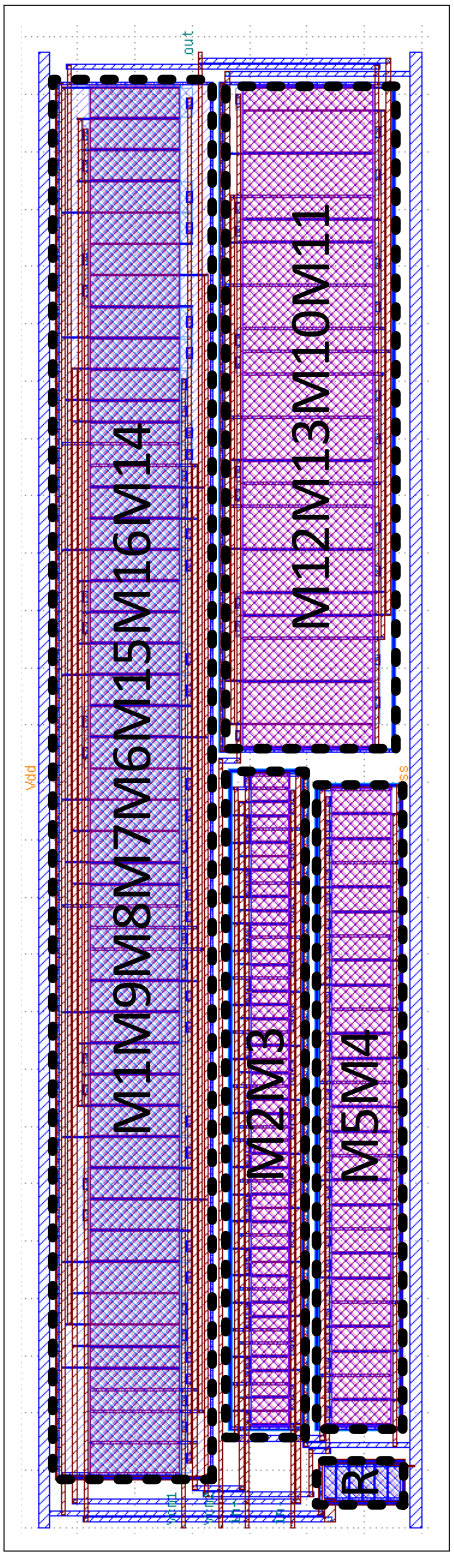


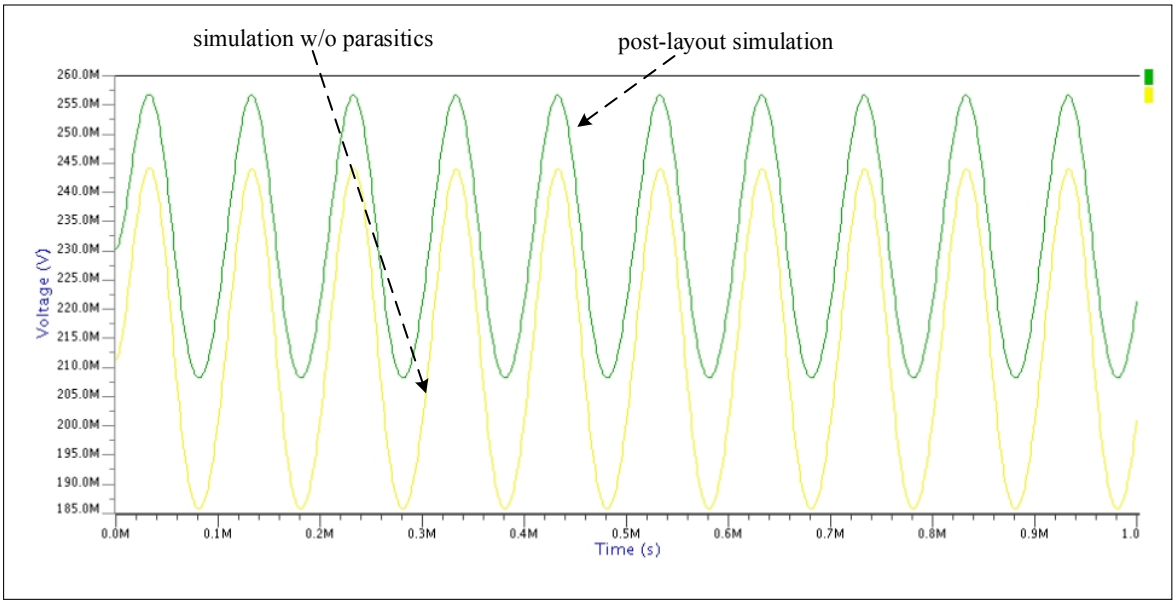
Figure 3.12. Instance of the Folded Amplifier Template.

3.3.6. Template for a Folded Amplifier

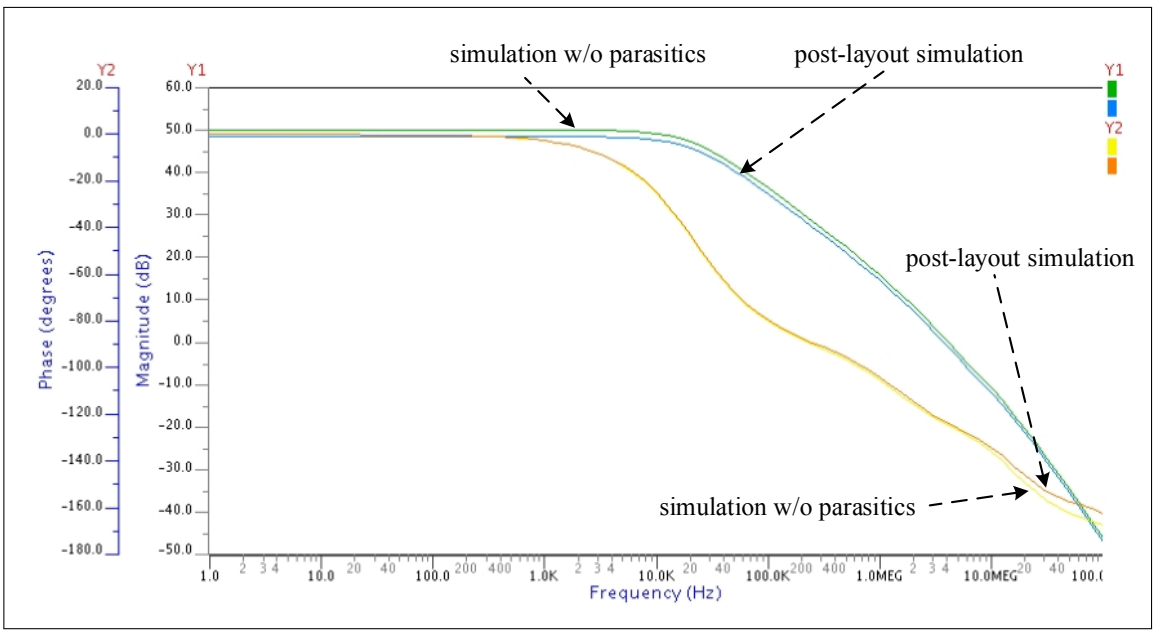
The routed transistor and the resistor generators are used to construct a template for the folded amplifier in Figure 3.11. The parameters of this template are listed:

- *WM1* : This parameter sets the widths of the M2 and M3 transistors.
- *WM2* : It sets the widths of the M1, M6, M7, M8, M9, M14, M15 and, M16 transistors.
- *WM3* : This one is used to set the widths of the M4 and M5 transistors.
- *WM4* : This parameter sets the widths of the M10, M11, M12 and, M13 transistors.
- *LM1* : Lengths of the M2 and M3 transistors are assigned via this parameter.
- *LM2* : It is used to set the lengths of the M1, M6, M7, M14, M15 and, M16 transistors.
- *LM3* : It sets the lengths of the M4 and M5 transistors.
- *LM4* : This parameter sets the lengths of the M10 and M13 transistors.
- *LM5* : This one sets the lengths of the M11 and M12 transistors.
- *LM6* : It sets the lengths of the M8 and M9 transistors.

An instance of the template is shown in Figure 3.3.6 where some of the transistors are folded and interdigitated. In Figure 3.3.6, the labels list interdigitated transistors. The instance of the folded amplifier, given in Figure 3.3.6, is extracted and simulated. Simulation results for the corresponding circuit and the post-layout simulation results are given in Figure 3.13.



a



b

Figure 3.13. Simulation Results (a) Transient Analysis, (b) AC Analysis.

4. AREA OPTIMIZATION

A key step of the physical synthesis process, and essential to determine layout area, is the layout floorplanning, i.e., the determination of properties and spatial relationships of a set of blocks. Lots of approaches have been reported in the three past decades for floorplanning of many (usually digital) cells in a chip [14, 27, 47, 48, 51, 55, 56].

One of the basic steps in building an analog layout is to find a good placement. Sometimes, it may be required to build the placement from scratch or it may also be required to reuse an existing placement. This may be the case if the layout is being synthesized from a template, where the template has been built by an expert or captured from an expert drawn layout, as described in Chapter 3. In this case, the placement problem reduces to variant selection, which is the process of choosing the appropriate implementation of each component for a given floorplan such that the total area is minimized. This problem is also called *area optimization on a fixed floorplan.*

An illustrative example is given in Figure 4.1. Assuming that the floorplan of Figure 4.1a is known to be good for some circuit in some desired performance range, the aim is to choose the correct variants for the given components and floorplan. In Figure 4.1b, the variants for the components A, B, C and D are listed, where the component B has two variants and the rest have single implementations. In other words, the layout of the component B is a *soft block* and the layouts of the components A, C and D are *fixed block*s. For the given floorplan, there exist two possible placements, given in Figure 4.1c and Figure 4.1d. The placement in Figure 4.1c has less area; therefore, it is the optimal solution of the problem. However, enumerating all solutions is not always possible in real situations. For instance, consider the case with 10 components where each component has 10 variants; there will be 10^{10} possible solutions which will take too long to enumerate. For some applications, such as layout-aware

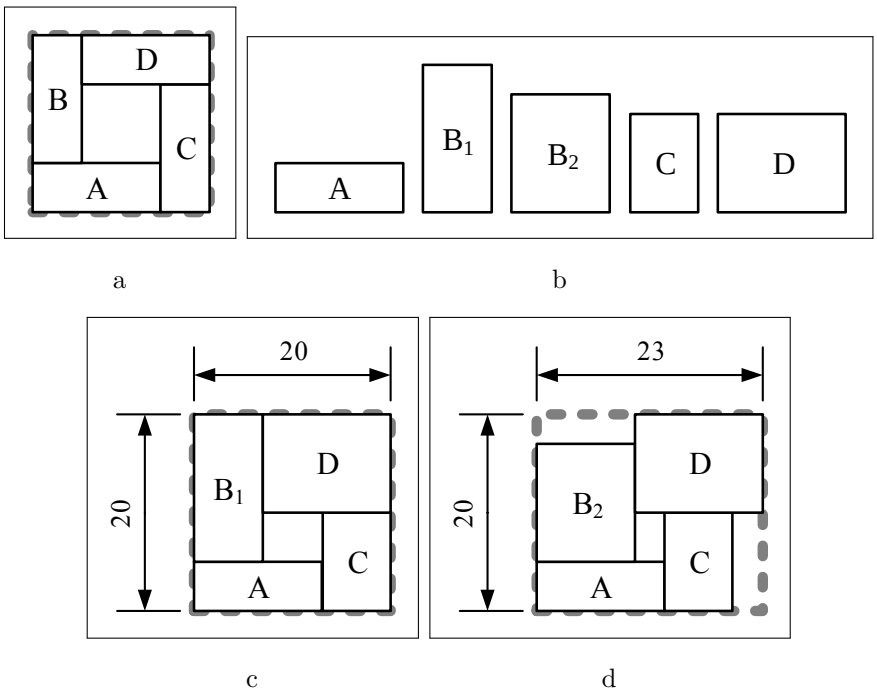


Figure 4.1. Variant Selection (a) Floorplan, (b) Variants for Components in (a), (c) Possible Placement where Variant B_1 is Selected, (d) Another Possibility where Variant B_2 is Selected.

optimization-based design loops [14], where a template is instanced many times, the run time is critical and iterations lasting more than a few seconds may be unacceptable.

Another issue is that analog circuits are very sensitive to layout parasitics whose effects on the circuit performance may be palliated through parasitic matching. Analog layout constraints are imposed on placement tools to minimize and match the parasitics. Even though the floorplan is fixed and the problem reduces to choosing the appropriate variants, analog constraints, such as symmetry, must still be satisfied.

4.1. Related Literature

The problem of variant selection can be solved on slicing floorplans[1] [45] using shape functions [46] which is a set composed of width w and height h pairs (w, h). The shape function of a given floorplan is obtained by hierarchically combining the

[1] A floorplan is called slicing if it can be obtained by repeated horizontal or vertical bisection of rectangular cells.

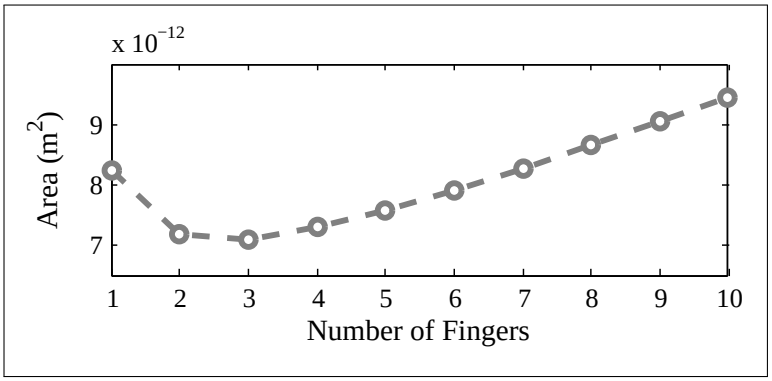


Figure 4.2. Area of a Transistor Changes as Number of Fingers m Changes. Each Circle Corresponds to a Different m Value for a Transistor in the UMC 0.13μm Technology with W=10μm and L=0.5μm.

shape functions of the lower levels in the hierarchy. This approach solves the problem optimally if all the variants are used during the construction of the shape function. As an example, if a capacitor whose width is allowed to change from a minimum value (5μm) to a maximum value (25μm), in steps of minimum resolution (10nm) is considered, all 2000 variants must be used in the construction of the shape function, which may be costly. Despite this fact, in most cases, this is a practical and fast approach. Unfortunately, it is not able to cope with non-slicing floorplans.

For non-slicing floorplans, a special case of the variant selection problem where the areas of the components are considered constant, irrespective of the selected variant, has been solved in [47–49] by using geometric programming. However, the variants of a component do not usually have the same area in realistic situations. For instance, the area of a transistor as the number of fingers changes is plotted in Figure 4.2. Moreover, this approach ignores the discrete nature of the geometrical parameters, such as the number of fingers, and solves the problem optimally for continuous parameters.

A related work [54] demonstrates that the minimum area floorplanning problem is not convex and its optimal solution cannot be obtained by solving its Lagrangian dual problem. However, this is only the case if the heights of the blocks are not constrained by their widths.

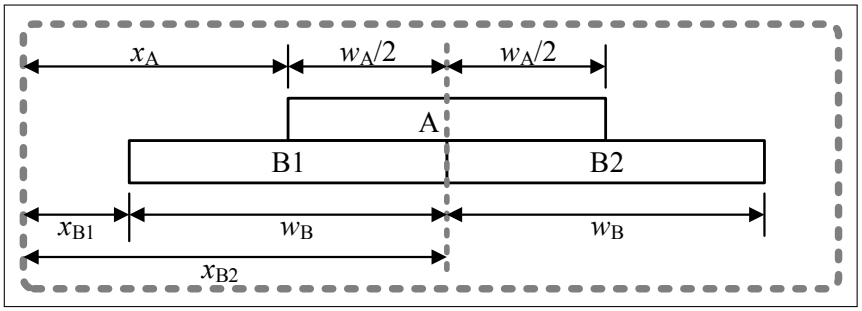


Figure 4.3. Components $B1$ and $B2$ are Symmetrically Placed with Respect to the Tick Dashed Line and the Center of the Component A Aligns with Symmetry Axis.

Analog designs are sensitive to parasitics where a remedy is parasitic matching which may be satisfied by applying symmetry constraints on a floorplan. Although the approach in [47–49] handles non-slicing floorplans, it is not supporting these analog layout constraints. As an example, a simple placement with symmetry and alignment constraints is given in Figure 4.3, where the dashed rectangle represents the boundary of the layout, B1 and B2 are symmetrically placed with respect to a symmetry axis, indicated by the vertical dashed line, and the center of A is aligned with this symmetry axis. Parameters x_A, x_{B1} and x_{B2} are the positions of the rectangles with respect to the boundary, w_A and w_B are the widths of the rectangles as indicated in Figure 4.3 and the mathematical formulation of the alignment and symmetry constraints is given by:

$$x_A + w_A/2 = x_{B2} \tag{4.1}$$

$$x_{B1} + w_B = x_{B2} \tag{4.2}$$

which can not be formulated in a geometric program. These equalities may only be added as inequalities into a geometric program as follows:

$$x_A + w_A/2 \leq x_{B2} \tag{4.3}$$

$$x_{B1} + w_B \leq x_{B2} \tag{4.4}$$

which do not guarantee satisfaction of the constraints.

Table 4.1. Comparison of the Available Methods with the Method Presented in This Chapter.

Method	Floorplan	Model	Constraints	Solution	Method
[46]	Slicing	Exact	Not Handled	Optimal	SF
[47–49]	General	Const. Area	Not Handled	Optimal	GP
[50]	General	Const. Area	Not Handled	Heuristic	SLP
[51]	General	Exact	Handled	Heuristic	ESF
[Presented Work]	General	Exact (Cont.)	Handled	Optimal	SCP

A function f(x) is called convex if it satisfies the condition:

$$\mathrm{f}(\theta \cdot x + (1-\theta) \cdot y) \leq \theta \cdot \mathrm{f}(x) + (1-\theta) \cdot \mathrm{f}(y) \tag{4.5}$$

where $\forall x, y \in \mathbf{dom} f$ and for all $\forall \theta \in \mathbf{R} : 0 \leq \theta \leq 1$. If the function f($x$) is continuous and twice differentiable,

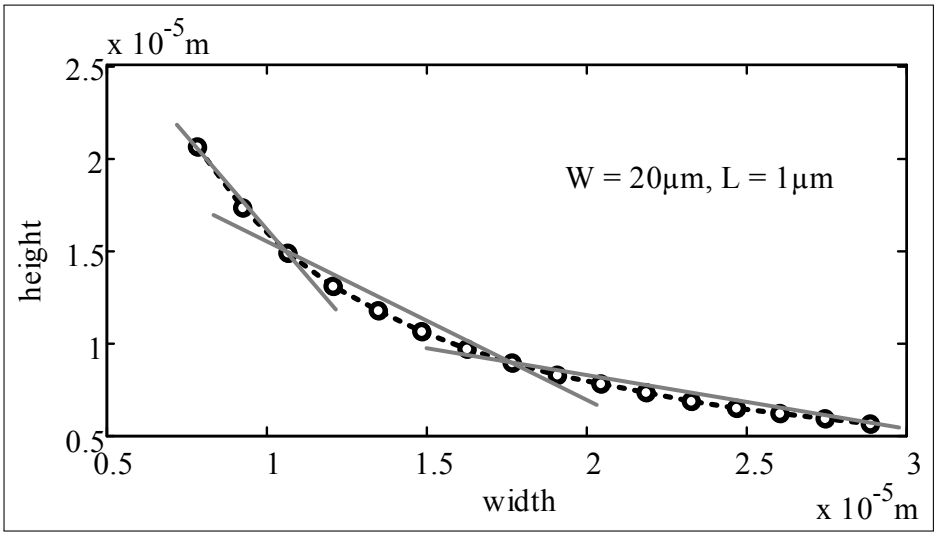
$$\frac{d^2\mathrm{f}(x)}{dx^2} \geq 0 \tag{4.6}$$

also indicates convexity. A local minimum of a convex function f(x) is in fact its global minimum [44]. Informally, a function is called convex if any line joining two points of the graph of the function lies above the graph.

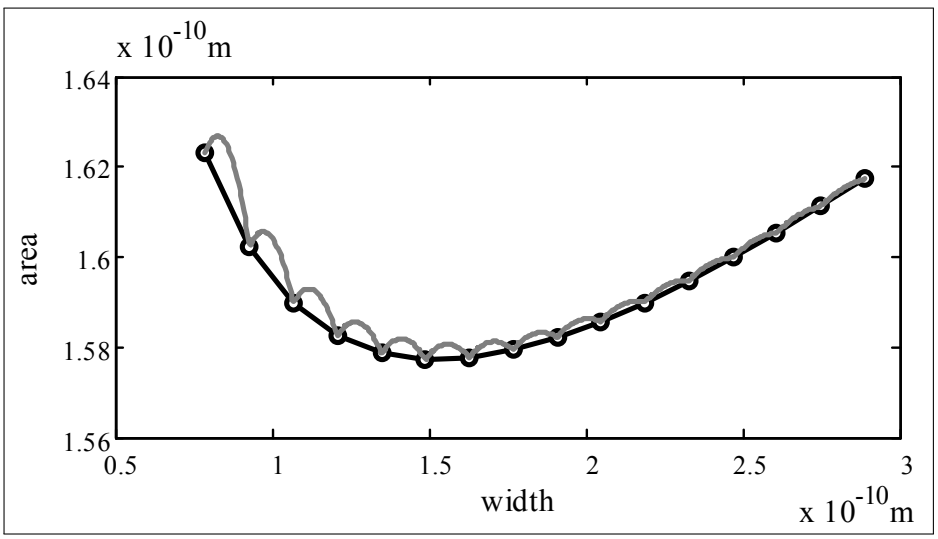
In [50] area optimization was performed by solving a sequence of linear programs iteratively. Areas of all components are also considered constant, which is not the present case, as indicated in Figure 4.2. This approach approximates the height h of a component with a set of lines l_i as follows:

$$\mathrm{h}(w) = \max(\mathrm{l}_1(w), \mathrm{l}_2(w), ..., \mathrm{l}_\mathrm{n}(w)) \tag{4.7}$$

where n is the number of lines. The approximation is applied to a real component, without constant area, in Figure 4.4a, where the height of a transistor is approximated



a



b

Figure 4.4. Linear Approximation (a) Height h vs. Width w, Number of Fingers m in $[5:20]$. Approximation for h with Three Lines Shown. (b) Area $\psi(w)$ of Transistor in (a) Plotted in Black, h Linearly Approximated with 15 Lines Plotted in Gray.

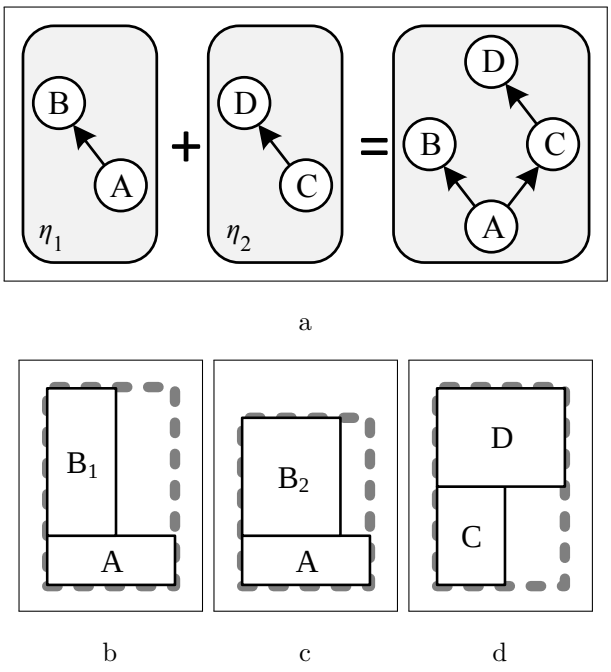


Figure 4.5. (a) Horizontal Addition of Two ESF where Placement of first ESF Represented by B*-Tree η_1 and Placement of Second One Represented by B*-Tree η_2. Possible Placements for η_1 Given in (b) and (c), and Placement for η_2 Given in (d).

with three lines. Due to the fact that the number of fingers m is an integer value the function is only defined at the data points, indicated by the circles. By increasing the number of lines, the height of this transistor may be exactly modeled at the data points. However, even for this case, if the area is plotted as a function of width or height, the plot has many local minima, as shown in Figure 4.4b. Thus, this formulation introduces many false local minima and reduces the probability of finding the global solution.

Area minimization with enhanced shape functions (ESF) for a non-slicing floorplan has been studied in [51]. This methodology does not specifically address variant selection but also tries to find a floorplan at the same time. In this paper, only the variant selection problem is addressed, thus only this aspect will be discussed. The ESF not only stores (w, h) pairs but also a B*-tree [29] representation η of the floorplan is attached to each pair. Then, by keeping the best solutions, constructed shape functions are combined hierarchically through horizontal/vertical enhanced shape addition [51], to construct the shape function of the complete layout. Although, the ESF methodology is also applicable to non-slicing floorplans, it is not a generalization of shape

functions, which are only applicable to slicing floorplans. As an example, the horizontal enhanced shape function addition combining two B*-trees is shown in Figure 4.5a. The variants are combined in an enhanced shape function by enumerating all possible solutions, but this may yield sub-optimal solutions. Let us assume the variants listed in Figure 4.1b are available for the horizontal enhanced shape addition in Figure 4.5a. During the construction of the ESF for the combination of A and B, all possible solutions are enumerated, which are shown in Figure 4.5b and Figure 4.5c. Due to the fact that the height of the combination given in Figure 4.5b is taller than the height of the one given in Figure 4.5c, the second one overwrites the first combination and the ESF contains only the solution shown in Figure 4.5c. The ESF for the combination of C and D contains only Figure 4.5d, which is the only possibility. Thus, the addition shown in Figure 4.5a uses the combinations given in Figure 4.5c and Figure 4.5d. The resulting layout after the addition is going to be the one in Figure 4.1d. However, a better solution may be obtained if the combinations in Figure 4.5b and Figure 4.5d were used. It would yield the layout in Figure 4.5c, which is narrower than the one given in Figure 4.1d. Although, this method yields suboptimal solutions, due to its hierarchical nature, its execution is very fast.

The approaches in [46, 47, 50, 51] and the methodology presented in this section are compared in Table 4.1. The approach proposed in this chapter is general, exactly models the areas of the components and handles symmetric placement constraints. Although the approach in [51] shares these properties, the presented methodology is shown to yield the globally optimal solution on a slicing floorplan when the components are modeled with the continuous model presented in Section 4.2. Although, some of the component parameters are allowed to have only discrete values, such as the number of fingers of a transistor, the results in Section 4.4.3 state that even when these parameters are rounded to the closest discrete value, the presented methodology still yields high quality solutions whereas the approach in [51] is a heuristic technique.

4.2. Convex Models for Devices

Some component parameters depend on the technology process and are fixed for a device generator, some are *input parameters* , e.g., width W and length L of a transistor, and the rest are *free parameters* which affect the geometry of the layout, e.g., the number of fingers of a resistor. If the input parameters are known for a component, its variants are generated by changing the free parameters. Analyzing the height-width relation of the variants of transistors, capacitors and resistors, the following model is developed:

$$h = \frac{k_1}{w - k_2} + k_3 \tag{4.8}$$

where the values of parameters k_i are determined by the technology and input parameters and they may be easily extracted by examining a few instances of a device generator. If the model in (4.8) is used for a component, it is observed that its area ψ is a convex function of its width w and height h as stated in the following theorem.

Theorem 4.1. *If the height h of a component can be modeled as a function of its width w by (4.8), and satisfies*

$$w \geq k_2 \tag{4.9}$$

$$h \geq k_3 \tag{4.10}$$

where the parameters k_1, k_2 and k_3 are non-negative real numbers, then, the area of this component

$$\psi = w \cdot h \tag{4.11}$$

is convex in w and h.

Proof. Rewriting the area $\psi(w)$ as a function of w:

$$\psi(w) = w \cdot h = w \cdot \left(\frac{k_1}{w - k_2} + k_3 \right) \tag{4.12}$$

and substituting $\psi(w)$ for f into (4.5), the following condition is obtained:

$$\psi(\theta \cdot x + (1 - \theta) \cdot y) \leq \theta \cdot \psi(x) + (1 - \theta) \cdot \psi(y) \tag{4.13}$$

where $\forall x, y \geq k_2$ and $\forall \theta \in \mathbf{R} : 0 \leq \theta \leq 1$. Evaluating $\psi(w)$ and simplifying the expression yields:

$$\frac{k_1 \cdot k_2 \cdot (x - y)^2 \cdot \theta \cdot (1 - \theta)}{(x - k_2) \cdot (y - k_2) \cdot (\theta \cdot x + (1 - \theta) \cdot y - k_2)} \geq 0 \tag{4.14}$$

which is always satisfied. Thus the area of the model is convex in w. Similar reasoning may be used to show that the area is also convex in h. □

The rest of this section shows that transistors, capacitors and resistors fit into the model stated in (4.8).

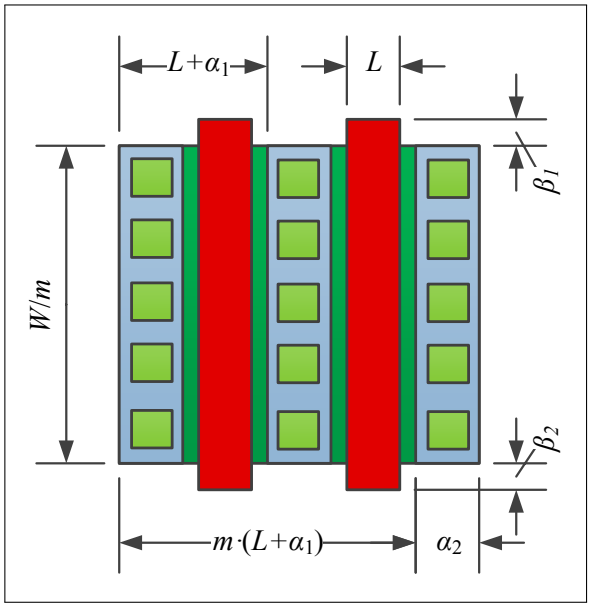
4.2.1. Transistor

A single transistor layout is sketched in Figure 4.6a. The width w of this cell is:

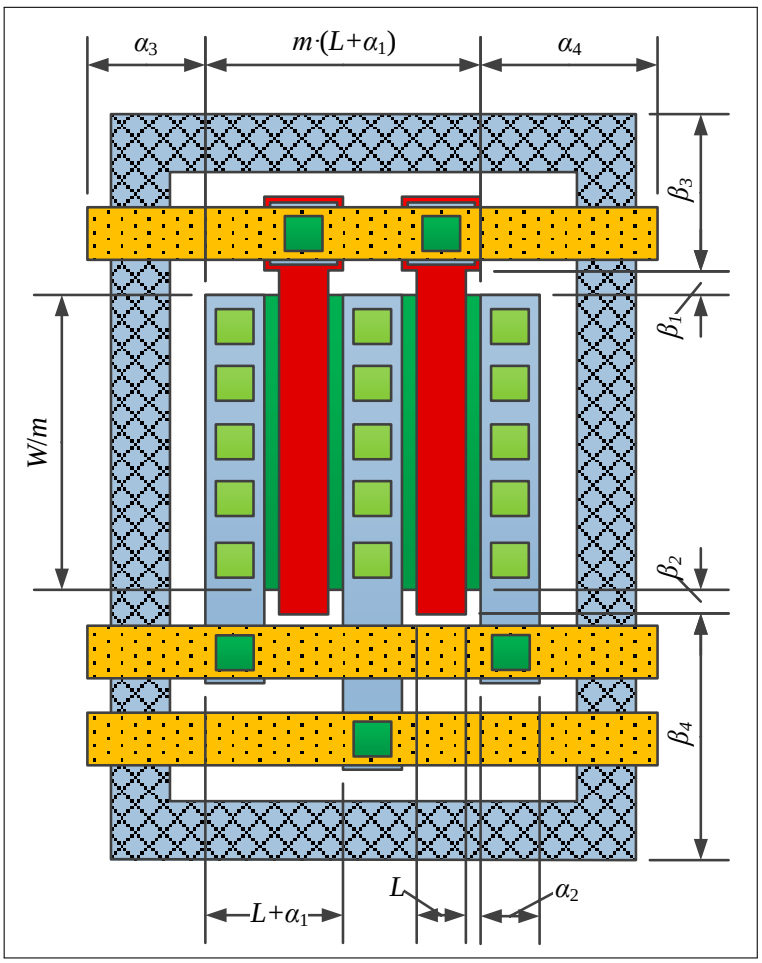
$$w = m \cdot (L + \alpha_1) + \alpha_2 \tag{4.15}$$

where m is the number of fingers, L is the channel length, α_1 and α_2 are technology-dependent constants. For this cell, the height h is:

$$h = \frac{W}{m} + \beta_1 + \beta_2 \tag{4.16}$$



a



b

Figure 4.6. Transistor Parameters Shown on Two Different Transistors (a) Simple Transistor, (b) Transistor with Guard-Ring and Routing.

where m is the number of fingers, W is the channel width, and β_1 and β_2 are constants. Finding the value of m in terms of the width w from (4.15) and substituting into (4.16), the height h is same as (4.8) and the value of k_1, k_2 and k_3 are:

$$k_1 = W \cdot (L + \alpha_1) \tag{4.17}$$

$$k_2 = \alpha_2 \tag{4.18}$$

$$k_3 = \beta_1 + \beta_2 \tag{4.19}$$

where the symbols α_i and β_i correspond to the physical dimensions in Figure 4.6a. Thus, the parameters k_i are all positive numbers and, from Figure 4.6a, the width w of the cell is wider than k_2 and the height h is taller than k_3. From Theorem 4.1, it is concluded that the area ψ of this cell is convex in w and h. The model in (4.8) is not limited to the simple transistor structure in Figure 4.6a. A transistor with routing and guard-ring is given in Figure 4.6b and the height h of this device may also be represented by (4.8), where the values of parameters k_i are:

$$k_1 = W \cdot (L + \alpha_1) \tag{4.20}$$

$$k_2 = \alpha_2 + \alpha_3 + \alpha_4 \tag{4.21}$$

$$k_3 = \beta_1 + \beta_2 + \beta_3 + \beta_4 \tag{4.22}$$

Thus, its area is also convex in its width w and h. It may be concluded that the formulation and the convexity is valid for all transistors independent of the orientation, the guard-ring, the existence of dummies, the routing.

4.2.2. Capacitor

In this section, diffusion, metal-insulator-metal, and metal-oxide-metal capacitor structures are analyzed.

A diffusion capacitor is depicted in Figure 4.7a. The capacitance of this cell is:

$$C = w_p \cdot h_p \cdot \gamma \tag{4.23}$$

where w_p and h_p are the width and height of the capacitor plate, respectively, and γ is the capacitance per unit area. The height h of the capacitor in Figure 4.7a is:

$$h = h_p + \beta_1 + \beta_2 \tag{4.24}$$

and its width w is:

$$w = w_p + \alpha_1 + \alpha_2 \tag{4.25}$$

where the parameters α_i and β_i are constants and are shown in Figure 4.7a. Then, the height h may be rewritten as:

$$h = \frac{\frac{C}{\gamma}}{w - (\alpha_1 + \alpha_2)} + (\beta_1 + \beta_2) \tag{4.26}$$

Equation (4.8) may be obtained by substituting the values of parameters k_i by

$$k_1 = C/\gamma \tag{4.27}$$

$$k_2 = \alpha_1 + \alpha_2 \tag{4.28}$$

$$k_3 = \beta_1 + \beta_2 \tag{4.29}$$

It can be verified in Figure 4.7a that

$$w \geq k_2 \tag{4.30}$$

$$h \geq k_3 \tag{4.31}$$

Theorem 4.1 concludes that the area of the model for a diffusion capacitance given in Figure 4.7a is convex in w and h.

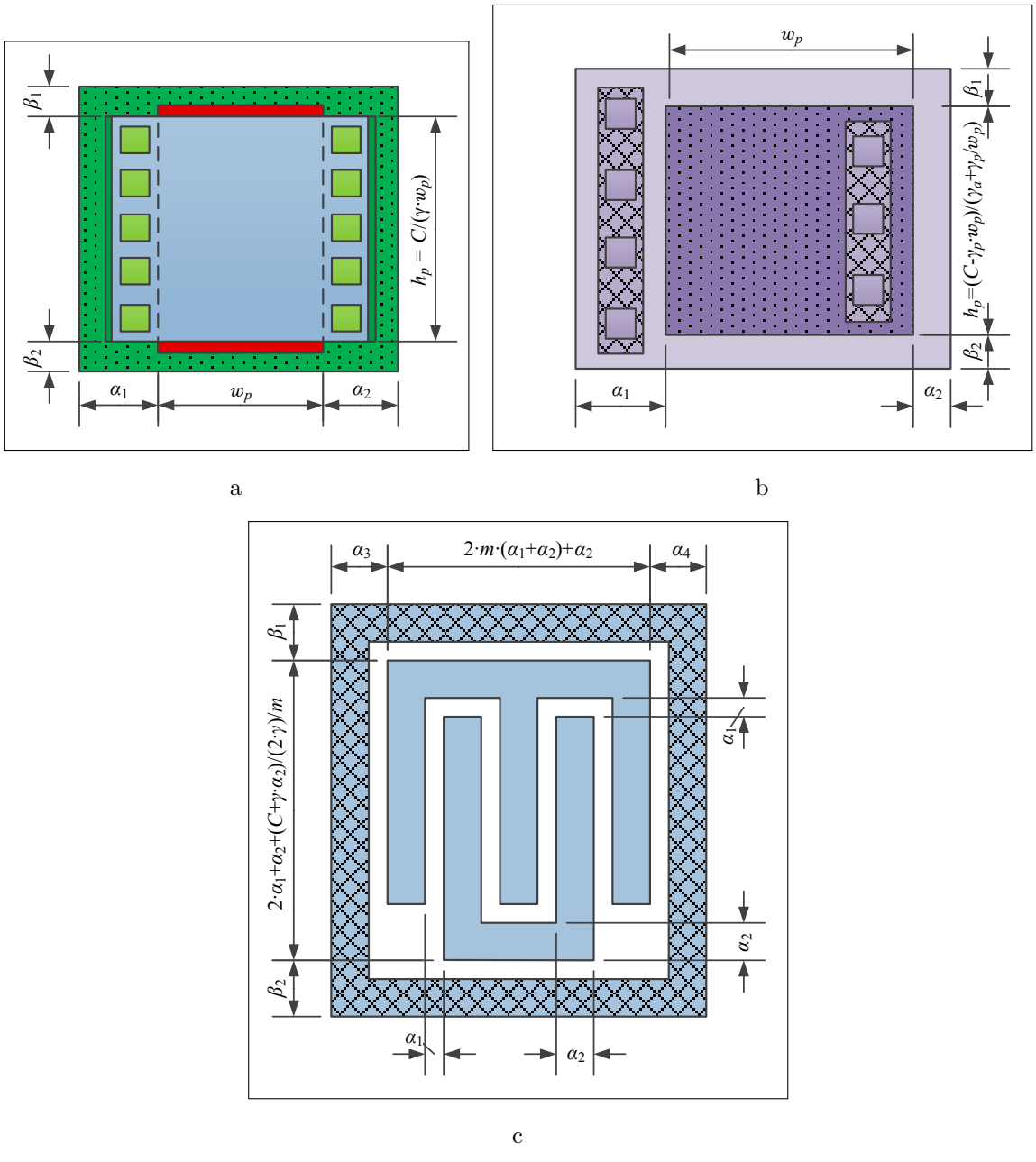


Figure 4.7. Parameters of Different Capacitor Shown (a) Diffusion Capacitor, (b) Metal-Insulator-Metal Capacitor, (c) Metal-Oxide-Metal Capacitor.

Figure 4.7b shows a metal-insulator-metal capacitor. The capacitance is calculated as follows:

$$C = w_p \cdot h_p \cdot \gamma_a + 2 \cdot (w_p + h_p) \cdot \gamma_p \tag{4.32}$$

where w_p, h_p, γ_a and γ_p are the width and height of the capacitor plate, capacitance per unit area, and fringing capacitance per unit length, respectively. The height h of the capacitor is:

$$h = h_p + \beta_1 + \beta_2 \tag{4.33}$$

and its width w is:

$$w = w_p + \alpha_1 + \alpha_2 \tag{4.34}$$

where the parameters α_i and β_i are constants. The height h of the model may be calculated in terms of the width w, as:

$$h = \frac{\frac{C}{\gamma_a} + \left(\frac{2 \cdot \gamma_p}{\gamma_a}\right)^2}{w - \left(\alpha_1 + \alpha_2 - \frac{2 \cdot \gamma_p}{\gamma_a}\right)} + \left(\beta_1 + \beta_2 - \frac{2 \cdot \gamma_p}{\gamma_a}\right) \tag{4.35}$$

Comparing (4.35) with (4.8), it is seen that:

$$k_1 = \frac{C}{\gamma_a} + \left(\frac{2 \cdot \gamma_p}{\gamma_a}\right)^2 \geq 0 \tag{4.36}$$

$$k_2 = \alpha_1 + \alpha_2 - \frac{2 \cdot \gamma_p}{\gamma_a} \tag{4.37}$$

$$k_3 = \beta_1 + \beta_2 - \frac{2 \cdot \gamma_p}{\gamma_a} \tag{4.38}$$

Technological parameters for 130nm, 90nm and 65nm CMOS processes have been checked and it is observed that

$$\alpha_1 + \alpha_2 \gg 2 \cdot \gamma_p/\gamma_a \tag{4.39}$$

$$\beta_1 + \beta_2 \gg 2 \cdot \gamma_p/\gamma_a \tag{4.40}$$

which states that the parameters k_2 and k_3 are positive. Thus, from Figure 4.7b it can be concluded that the conditions in Theorem 4.1 are satisfied. That is, the area of the model for a metal-insulator-metal capacitor is convex in w and h.

Figure 4.7c shows a metal-oxide-metal capacitor. The capacitance is:

$$C = \gamma_p \cdot ((2 \cdot m \cdot (h - \beta - 2 \cdot \alpha)) + (2 \cdot m - 1) \cdot \alpha) \tag{4.41}$$

where

$$\beta = \beta_1 + \beta_2 \tag{4.42}$$

$$\alpha = \alpha_1 + \alpha_2 \tag{4.43}$$

and m is the number of fingers, h is the height of the capacitor, γ_p is the capacitance per unit length and, the parameters α_i and β_i are constants. The width w of the capacitor is:

$$w = 2 \cdot m \cdot (\alpha_1 + \alpha_2) + \alpha_3 + \alpha_4 \tag{4.44}$$

Solving this equation for m and substituting the solution for m in (4.41), the height h is found to be:

$$h = \frac{\frac{\alpha_1+\alpha_2}{\gamma_p} \cdot (C + \gamma_p \cdot (\alpha_1 + \alpha_2))}{w - (\alpha_2 + \alpha_3 + \alpha_4)} + (\alpha_1 + \alpha_2 + \beta_1 + \beta_2) \tag{4.45}$$

This expression is equivalent to (4.8) when the values of the parameters k_i are selected as follows:

$$k_1 = \frac{\alpha_1 + \alpha_2}{\gamma_p} \cdot (C + \gamma_p \cdot (\alpha_1 + \alpha_2)) \tag{4.46}$$

$$k_2 = \alpha_2 + \alpha_3 + \alpha_4 \tag{4.47}$$

$$k_3 = \alpha_1 + \alpha_2 + \beta_1 + \beta_2 \tag{4.48}$$

The parameters k_1, k_2 and k_3 are non-negative numbers and it is seen from Figure 4.7c that

$$w \geq k_2 \tag{4.49}$$

$$h \geq k_3 \tag{4.50}$$

Thus, Theorem 4.1 concludes that the area of the cell given in Figure 4.7c is convex in w and h.

4.2.3. Resistor

A resistor layout is sketched in Figure 4.8. The height h of this cell is formulated as follows:

$$h = \frac{R \cdot (\alpha_1 + \alpha_2)}{w \cdot \gamma \cdot \alpha_1} + \beta_1 + \beta_2 \tag{4.51}$$

where the parameter γ is the sheet resistance. The parameters α_i and β_i are non-negative and their value depends on the device generator. This model may be rewritten as (4.8), if the values of the parameters k_1 and k_3 are chosen to be

$$k_1 = \frac{R \cdot (\alpha_1 + \alpha_2)}{\gamma \cdot \alpha_1} \tag{4.52}$$

$$k_3 = \beta_1 + \beta_2 \tag{4.53}$$

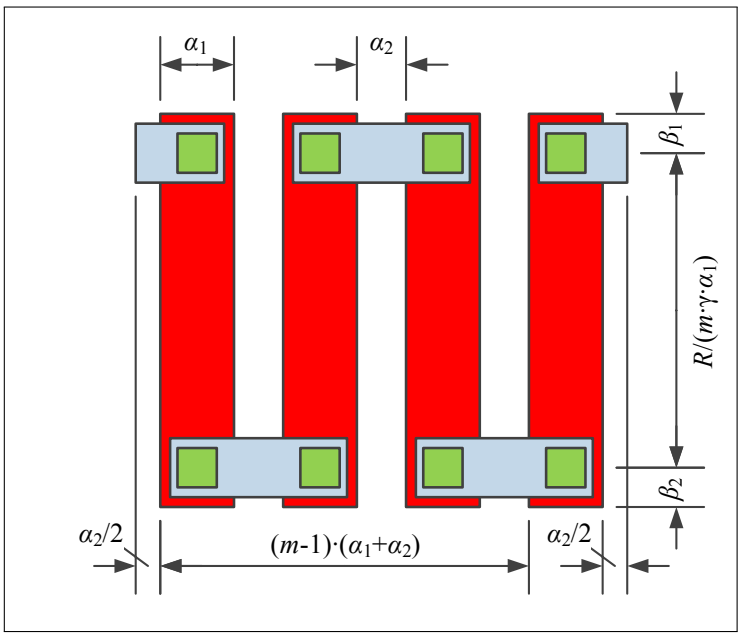


Figure 4.8. Parameters of a Resistor.

and the value of the parameter k_2 is chosen to be zero. It is obvious from Figure 4.8 that

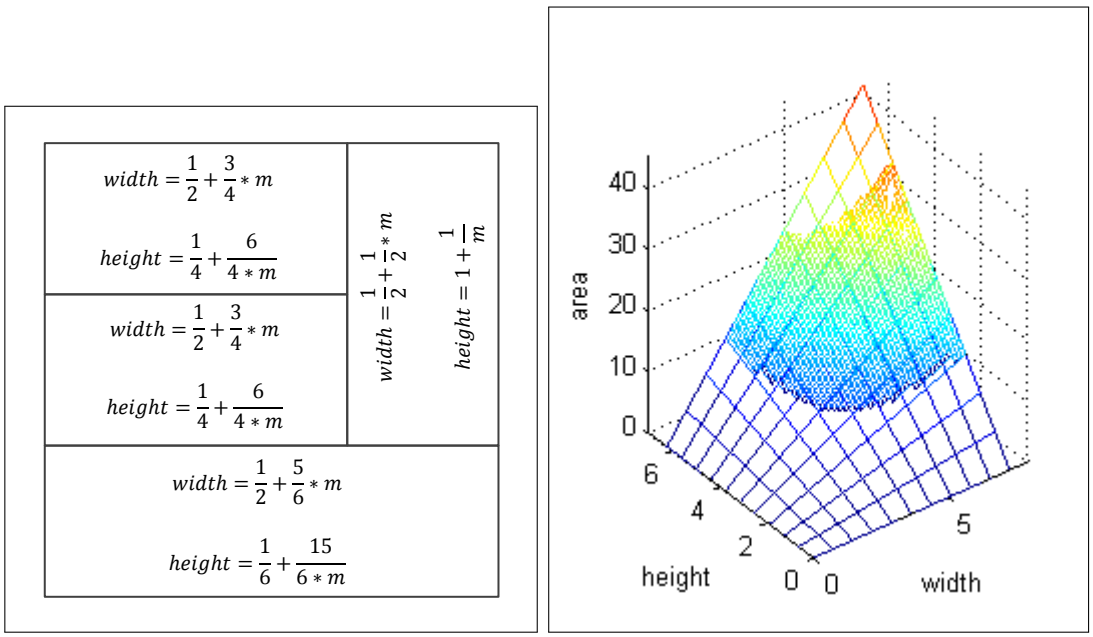
$$h \geq k_3 \tag{4.54}$$

By Theorem 4.1, the area of the given resistor cell is convex in its width w and height h.

4.3. Convexity of a Layout

Consider the following experiments:

(i) A slicing floorplan composed of four components is shown in Figure 4.9a. Geometrical models are assigned to these components and these are also given on Figure 4.9a. For this layout, the number of fingers m for all components are the free parameters. Keeping the floorplan fixed and sweeping all m's from 1 to 5, the area plot in Figure 4.9b is obtained. In this plot, the width/height (w/h) Pareto-Front may be observed to be a convex function of (w, h).



a b

Figure 4.9. (a) Placement and Functions for Dimensions, (b) Area Plot: Obtained by Varying All ms in (a) from 1 to 5, Pareto-Front (w/h) in (b) Observed to be Convex.

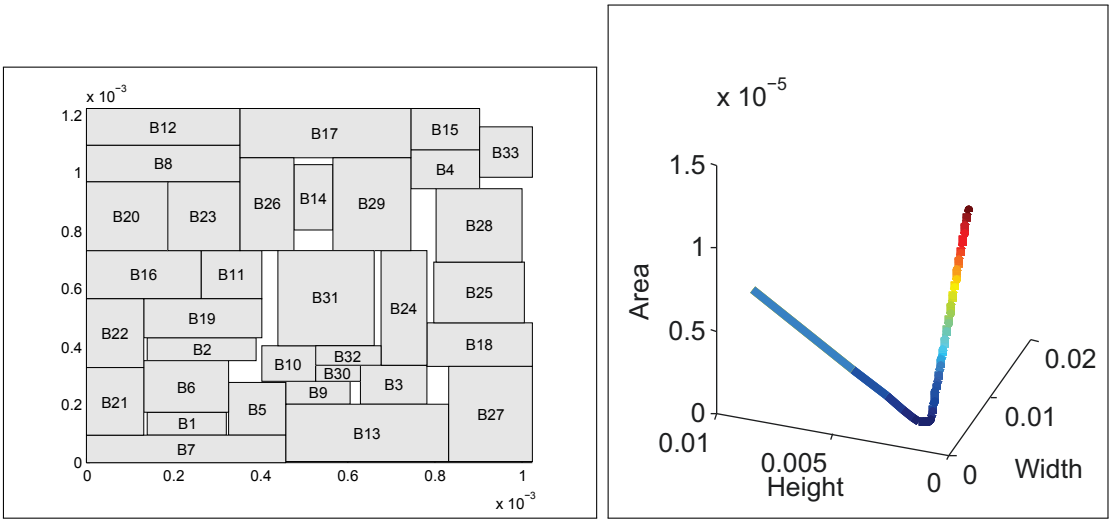
(ii) Hundreds of random placements of *ami33* circuit from the *MCNC* benchmarks are generated and the convexity is investigated. Component areas in the benchmark are kept constant and the aspect ratios are allowed to be free in the interval $[2/3, 3/2]$. Pareto-Fronts are plotted and all of them are observed to be convex. A placement and the corresponding Pareto-Front is shown in Figure 4.10.

The convexities in these experiments are not a coincidence and the following sections prove this fact.

4.3.1. Convexity of Slicing Floorplans

This sections deals with slicing floorplans. A floorplan is called slicing [27], if it is possible to recursively split it into rectangular slices by means of successive horizontal and vertical cuts. The resulting slices may be represented by a binary tree.

We define a layout as *compact* if for a given width (height), it is not possible to obtain a shorter (thinner) layout by changing the geometrical parameters. For



a b

Figure 4.10. (a) A Placement of *ami33* from *MCNC* Benchmark, (b) Pareto Frontier for the Placement in (a).

instance, the layout of Figure 4.11a is not compact for the defined width w, if there exists a shorter layout, as in Figure 4.11b. If the layout of Figure 4.11b is the shortest possible layout, we call it a compact layout for the given width w. The concept of compactness is visualized in Figure 4.11c, where the area of a layout and the area of a compact layout are represented by the filled region and the dark solid line, respectively.

In this section, the width and height of a combination is denoted by w and h, respectively. The symbol ψ is used to denote the area of a combination. The area of a horizontal combination, as in Figure 4.12, is denoted by ψ_{H} whereas the area of a vertical combination, as in Figure 4.13, is denoted by ψ_{V}. Depending on the context a second index is used to denote equal ($\psi_{\mathrm{H_E}}$ or $\psi_{\mathrm{V_E}}$), maximum ($\psi_{\mathrm{H_X}}$ or $\psi_{\mathrm{V_X}}$) and minimum ($\psi_{\mathrm{H_N}}$ or $\psi_{\mathrm{V_N}}$).

In this section we prove that the total area of a compact slicing floorplan is a convex function of its width (alternatively height). It is also shown that the convexity is satisfied, without applying any transformation and without any assumption on constant area. The proof makes only two assumptions on the component blocks:

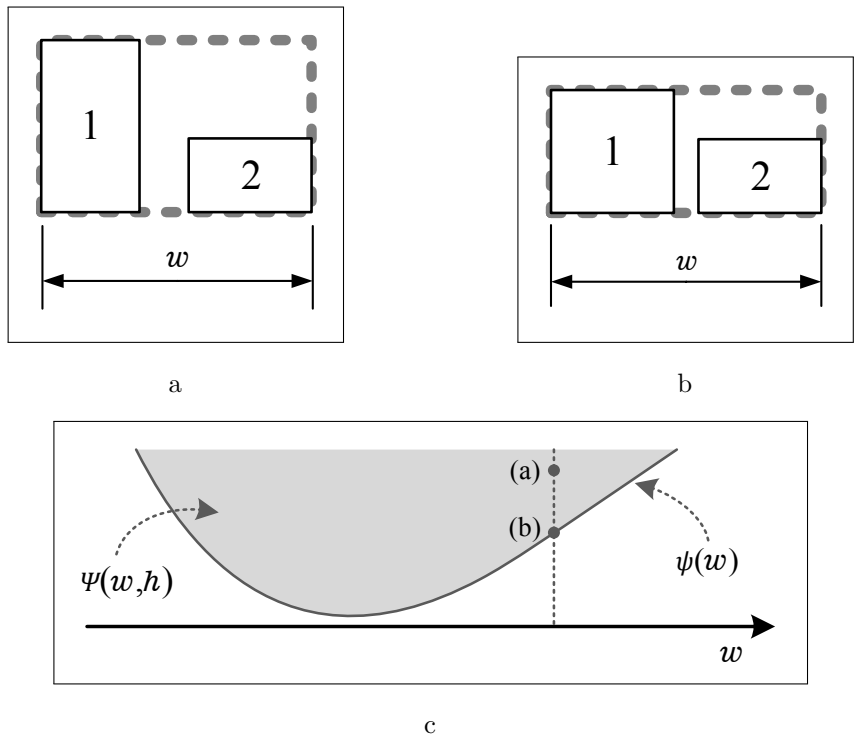


Figure 4.11. Layout (a) Not Compact, If Possible to Obtain Shorter Layout as in (b). Points Representing Areas of (a) and (b) Shown in (c). Filled Region in (c) Represents Area of Any Layout where Dark Solid Line is for a Compact Layout.

- The area of a block is a convex function of its width (height), which is the case for analog devices such as: resistors, capacitors and, transistors as shown in Section 4.2.
- Height h(w) (alt. width w(h)) is a non-increasing function of the width w (alt. height h). In other words, the wider the variant is, the shorter the height is. This is also the case for analog devices [55]. Formally stating:

$$\frac{d}{dw}[\mathrm{h}(w)] \leq 0 \tag{4.55}$$

From this point, the width w is accepted as the free parameter and the height h(w) is used as a function of the width. All the results may be obtained, without loss of generalization, if the height h is considered as the free parameter and the width w(h) is used as a function of the height.

Our proof demonstrates that the area of a compact layout with two blocks is convex and can be extended to any compact slicing floorplan. First, we will consider the horizontal (Figure 4.12) or vertical (Figure 4.13) combination of two blocks without dead space.

Theorem 4.2. *If the area of the i-th block in Figure 4.12, given as:*

$$\mathrm{h_i}(w_i) \cdot w_i \tag{4.56}$$

is a convex function of w_i *and the height of the block* $\mathrm{h_i}(w_i)$ *is a non-increasing function of* w_i*, then*

$$\frac{d^2}{dw_i^2}[\mathrm{h_i}(w_i)] \geq -\frac{2}{w_i} \cdot \frac{d}{dw_i}[\mathrm{h_i}(w_i)] \tag{4.57}$$

is satisfied and $\mathrm{h_i}(w_i)$ *is a convex function of* w_i*.*

Proof. Due to the fact that the area of the i-th block, given in (4.56), is convex in w_i, its second derivative is non-negative. Twice differentiating the expression in (4.56) and imposing that it is non-negative yields:

$$\frac{d^2}{dw_i^2}[\mathrm{h_i}(w_i)] \cdot w_i + 2 \cdot \frac{d}{dw_i}[\mathrm{h_i}(w_i)] \geq 0 \tag{4.58}$$

and rearranging the terms, equation (4.57) is obtained. As $\mathrm{h_i}(w_i)$ is a non-increasing function of w_i, its first derivative is a non-positive number:

$$\frac{d}{dw_i}[\mathrm{h_i}(w_i)] \leq 0 \tag{4.59}$$

Using the fact that the width of the i-th block w_i is a non-negative number and (4.59), the following expression is obtained from (4.57):

$$\frac{d^2}{dw_i^2}[\mathrm{h_i}(w_i)] \geq 0 \tag{4.60}$$

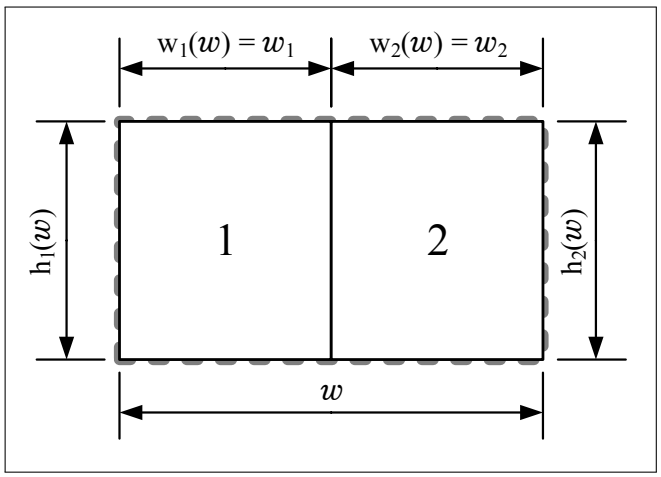


Figure 4.12. Horizontal Combination without Dead Space. Heights of the Blocks Equal, Configuration Denoted by $\mathrm{H_E}$ (horizontal-equal-height).

Therefore, the height of the i-th block $\mathrm{h_i}(w_i)$ is a convex function of its width w_i. □

Theorem 4.3. *Consider the horizontal combination, denoted by* $\mathrm{H_E}$*, given in Figure 4.12. If the area of each block in Figure 4.12 is a convex function of its width* w_i*, and its height* $\mathrm{h_i}(w_i)$ *is a non-increasing function of its width* w_i*, then the total area of the combination is a convex function of the width* w*.*

Proof. The area of the layout in Figure 4.12 may be written as:

$$\psi_{\mathrm{H_E}}(w) = \mathrm{h_1}(w_1(w)) \cdot w = \mathrm{h_2}(w_2(w)) \cdot w \tag{4.61}$$

generically:

$$\psi_{\mathrm{H_E}}(w) = \mathrm{h_i}(w_i(w)) \cdot w \tag{4.62}$$

The second derivative of (4.62) with respect to w is defined as Δ:

$$\begin{aligned} \psi''_{\mathrm{H_E}}(w) &= \Delta & (4.63) \\ &= \frac{d^2}{dw_i^2}[\mathrm{h_i}(w_i)] \cdot (w_i')^2 \cdot w \\ &\quad + \frac{d}{dw_i}[\mathrm{h_i}(w_i)] \cdot (w_i'' \cdot w + w_i') \\ &\quad + \frac{d}{dw_i}[\mathrm{h_i}(w_i)] \cdot w_i' & (4.64) \end{aligned}$$

where the prime symbol ($'$) indicates the first derivative with respect to w. Replacing the term

$$\frac{d^2}{dw_i^2}[\mathrm{h_i}(w_i)]$$

with a smaller quantity given by (4.57):

$$-\frac{2}{w_i} \cdot \frac{d}{dw_i}[\mathrm{h_i}(w_i)]$$

δ_i is obtained, that must necessarily be smaller than Δ:

$$\Delta \geq \delta_i = -\frac{d}{dw_i}[\mathrm{h_i}(w_i)] \cdot \zeta_i \tag{4.65}$$

where ζ_i is defined as:

$$\zeta_i = \left(2 \cdot w_i' \cdot \left(\frac{w_i' \cdot w}{w_i} - 1\right) - w_i'' \cdot w\right) \tag{4.66}$$

Summing ζ_1 and ζ_2 and using:

$$\begin{aligned} w_1 + w_2 &= w & (4.67) \\ w_1' + w_2' &= 1 & (4.68) \\ w_1'' + w_2'' &= 0 & (4.69) \end{aligned}$$

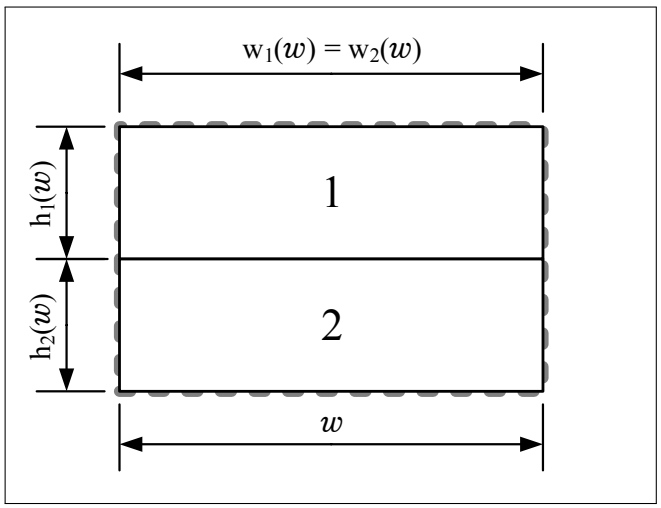


Figure 4.13. Vertical Combination without Dead Space. Widths of Blocks Equal, Configuration Denoted by V_E (vertical-equal-width).

the following expression is obtained:

$$\zeta_1 + \zeta_2 = 2 \cdot \frac{(w_1' \cdot w_2 - w_1 \cdot w_2')^2}{w_1 \cdot w_2} \geq 0 \tag{4.70}$$

This expression must be non-negative because w_1 and w_2 are non-negative. Thus, at least one of the functions, ζ_1 or ζ_2, must be non-negative. It is also known from (4.59) that the first derivative of the height of the i-th block with respect to its width is non-positive. Using this fact in (4.65), it is concluded that either δ_1 or δ_2 must be non-negative. Again from (4.65), Δ is non-negative which is defined in (4.63) as the second derivative of the area ψ_{H_E}.

In conclusion, ψ''_{H_E} is always positive, thus the total area is a convex function of the width w.

□

Theorem 4.4. *Consider the vertical combination, denoted by* V_E*, given in Figure 4.13. If the area of each block is a convex function of its width, then the total area of the layout is convex.*

Proof. The total area of the layout may be formulated as:

$$\psi_{\mathrm{V_E}}(w) = \mathrm{h}_1(w) \cdot w + \mathrm{h}_2(w) \cdot w \tag{4.71}$$

It is known that the area of each block in Figure 4.13 is a convex function of its width. But it is also known that the width of each block is the width of the layout. Thus, the products in (4.71) are convex in w. It is also known that a positive weighted sum of convex functions is also a convex function. Thus, the expression in (4.71) is a convex function of w. □

When the i-th block in Figure 4.12 reaches its dimensional limits:

$$w_{i_{min}} \leq w_i \leq w_{i_{max}} \tag{4.72}$$

some dead space is going to emerge. The possible cases are shown in Figure 4.14, where the layouts are considered to be compact for a given width w.

Theorem 4.5. *For the layouts given in Figure 4.14, the total area is a convex function of the width w.*

Proof. For the cases in Figure 4.14a and in Figure 4.14b, the height of the layout reaches its minimum value $h_{1_{min}}$ and does not decrease if w increases. Formally, the area $\psi_{\mathrm{H_N}}$ is:

$$\psi_{\mathrm{H_N}}(w) = \psi_{\mathrm{H_{N1}}}(w) = \psi_{\mathrm{H_{N2}}}(w) = h_{1_{min}} \cdot w \tag{4.73}$$

This expression is a linear function of w, thus the area is a convex function of w. For the case in Figure 4.14c, the area is :

$$\bar{\psi}_{\mathrm{H_{X1}}}(w_2) = \mathrm{h}_2(w_2) \cdot w_{1_{min}} + \mathrm{h}_2(w_2) \cdot w_2 \tag{4.74}$$

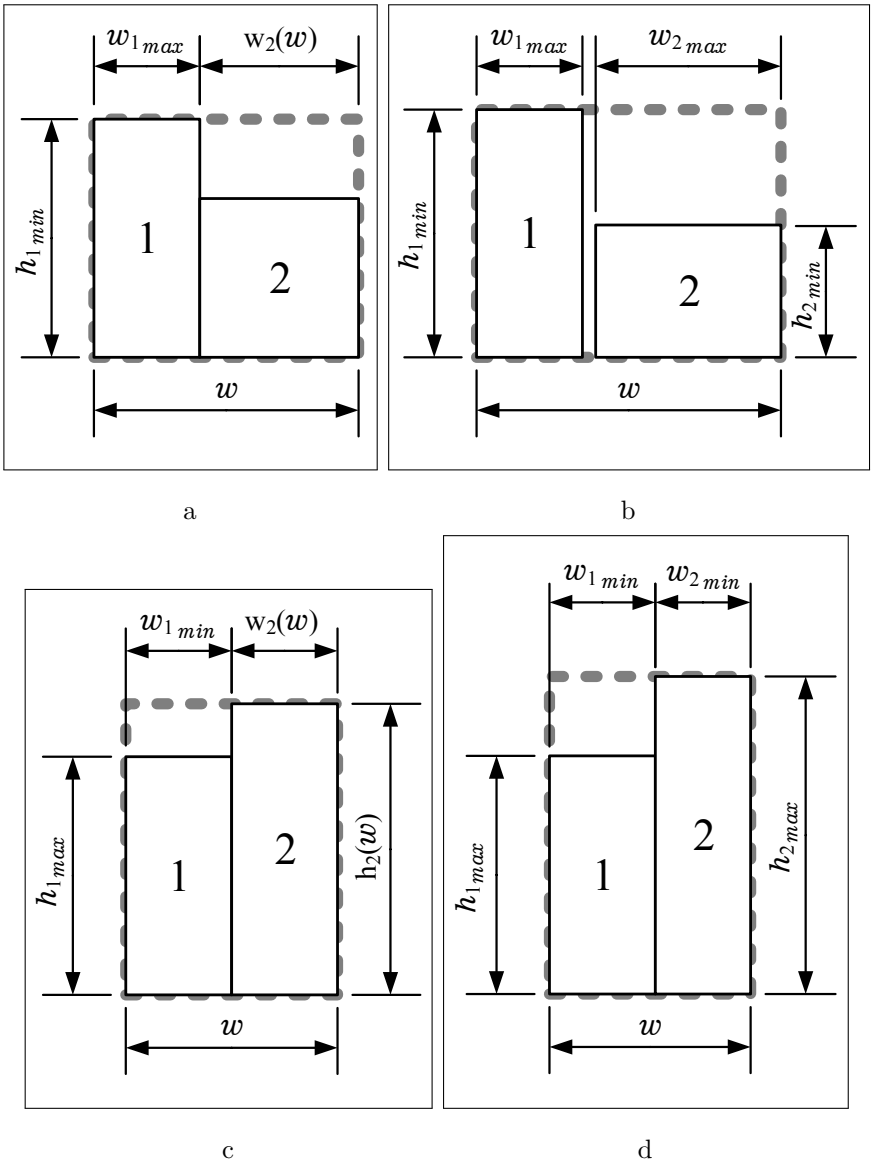


Figure 4.14. Horizontal Combination with Dead Space (a) Block Reaches Minimum Height (H_{N1}), (b) Both Blocks Reach Minimum Height (H_{N2}), (c) One Block Reaches Maximum Height (H_{X1}), (d) Both Blocks Reach Maximum Height (H_{X2}).

where $\bar{\psi}_{\mathrm{H_{X1}}}$ is a function of w_2. Noting that the term $\mathrm{h}_2(w_2)$ is known to be convex from Theorem 4.2 and $w_{1_{min}}$ is a positive constant, the first and the second products in the sum are convex functions of w_2. Thus, $\bar{\psi}_{\mathrm{H_{X1}}}$ is a convex function of w_2. The area of the combination given in Figure 4.14c:

$$\psi_{\mathrm{H_{X1}}}(w) = \bar{\psi}_{\mathrm{H_{X1}}}(w - w_{1_{min}}) \tag{4.75}$$

is a shifted version of $\bar{\psi}_{\mathrm{H_{X1}}}$, and if one is convex, the other one is also convex. Finally, the case in Figure 4.14d is the limit case: w is not allowed to have values smaller than $w_{1_{min}} + w_{2_{min}}$, and the area of the combination is a constant:

$$\psi_{\mathrm{H_{X2}}} = \psi_{\mathrm{H_{X1}}}(w_{1_{min}} + w_{2_{min}}) \tag{4.76}$$

□

Vertical combination, other than the one in Figure 4.13, are generalized in Figure 4.15, where each one is compact for a given w.

Theorem 4.6. *For the layouts given in Figure 4.15, the total area is a convex function of the width w.*

Proof. In Figure 4.15a, the narrowest possible case is given. In this case there is no dead space and convexity is already stated in Theorem 4.4:

$$\psi_{\mathrm{V_{N1}}} = \psi_{\mathrm{V_E}}(w_{1_{min}}) \tag{4.77}$$

For the case of Figure 4.15b, the total area is:

$$\psi_{\mathrm{V_{X1}}}(w) = h_{1_{min}} \cdot w + \mathrm{h}_2(w) \cdot w \tag{4.78}$$

where the expression is a sum of two convex functions of w. The first product in the sum is a linear function of w and the second term is convex by the initial assumptions

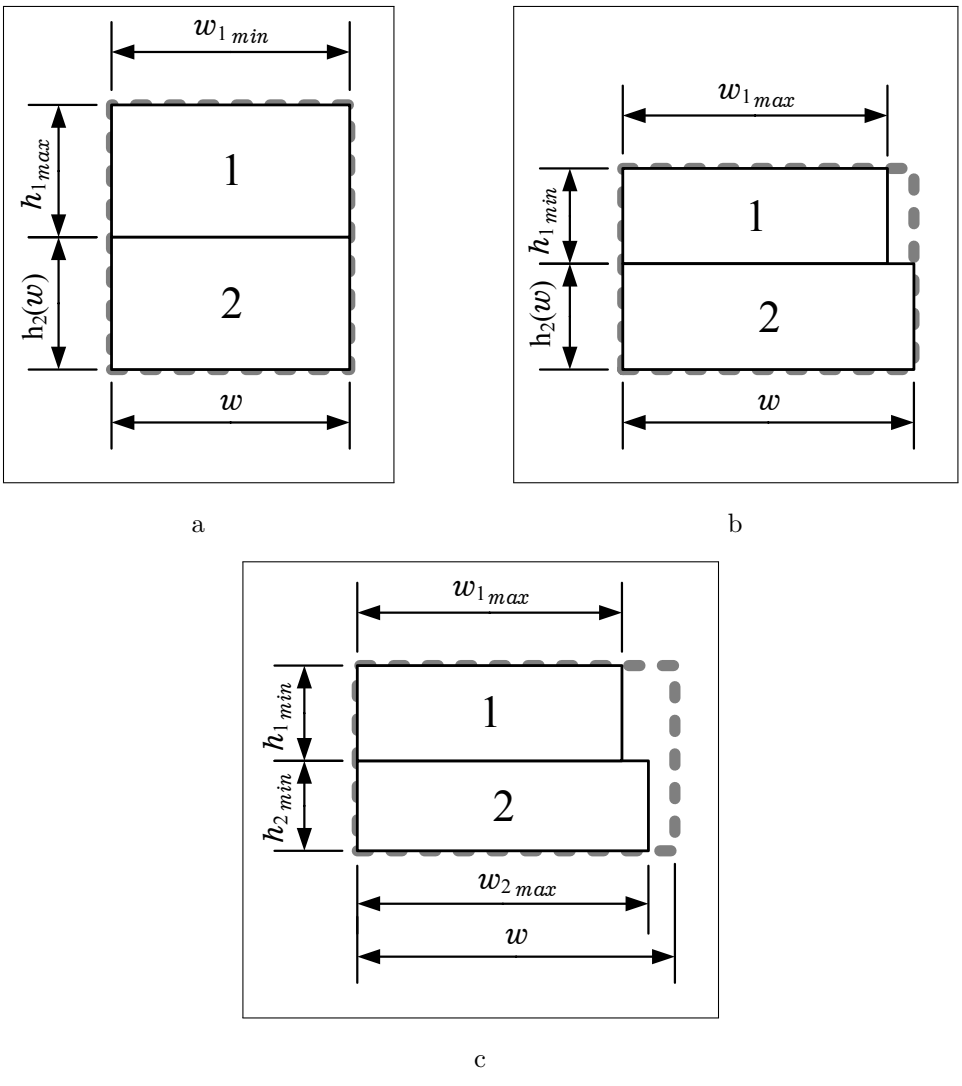
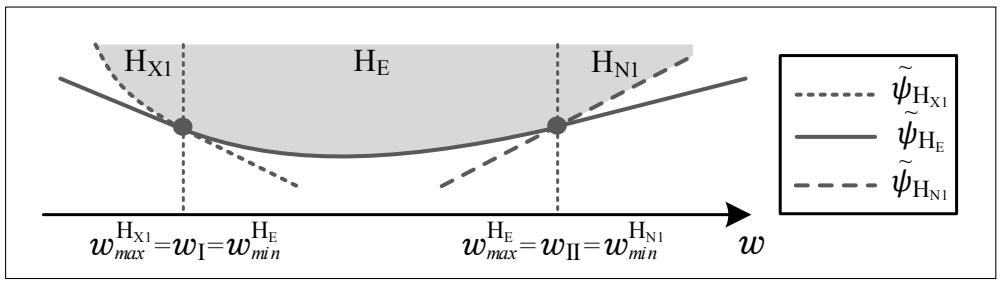
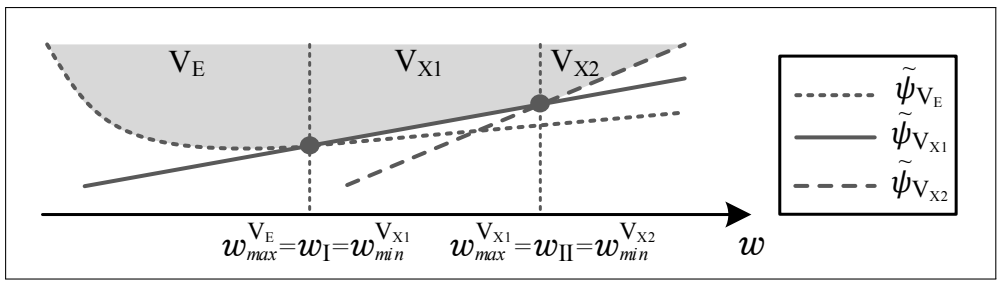


Figure 4.15. Vertical Combination (a) Block Reaches Minimum Width (V_{N1}), (b) Block Reaches Maximum Width (V_{X1}), (c) Both Blocks Reach Maximum Width (V_{X2}).



a



b

Figure 4.16. Area as a function of w Obtained by Taking Maximum of Extended Functions (a) Horizontal Combination, Area ψ_{H} is the Maximum of $\tilde{\psi}_{\mathrm{H_{X1}}}$, $\tilde{\psi}_{\mathrm{H_E}}$, $\tilde{\psi}_{\mathrm{H_{N1}}}$. (b) Vertical Combination, Area ψ_{V} is the Maximum of $\tilde{\psi}_{\mathrm{V_E}}$, $\tilde{\psi}_{\mathrm{V_{X1}}}$, $\tilde{\psi}_{\mathrm{V_{X2}}}$.

in this section. Considering the case of Figure 4.15c, the area may be formulated as:

$$\psi_{\mathrm{V_{X2}}}(w) = (h_{1_{min}} + h_{2_{min}}) \cdot w \tag{4.79}$$

where this expression is linear, thus, a convex function of w. □

Up to this point convexity for different cases are analyzed. The following theorem proves the convexity when all the cases in Figure 4.12, Figure 4.13, Figure 4.14 and Figure 4.15 are considered together.

Theorem 4.7. *The area of the horizontal combination may be formulated as:*

$$\psi_{\mathrm{H}}(w) = \max(\tilde{\psi}_{\mathrm{H_{X1}}}, \tilde{\psi}_{\mathrm{H_E}}, \tilde{\psi}_{\mathrm{H_{N1}}}) \tag{4.80}$$

and, similarly, the area of the vertical combination may be formulated as:

$$\psi_{\mathrm{V}}(w) = \max(\tilde{\psi}_{\mathrm{V_E}}, \tilde{\psi}_{\mathrm{V_{X1}}}, \tilde{\psi}_{\mathrm{V_{X2}}}) \tag{4.81}$$

where $\tilde{\psi}$ is a function of the width w and is obtained through extending $\psi(w)$, which is defined in the range $[w_{min}, w_{max}]$, linearly in the ranges $(0, w_{min})$ and (w_{max}, ∞) such that for each of the points w_{min} and w_{max} the first derivative of $\tilde{\psi}(w)$ both from left and right are same.

The area of both the horizontal combination $\psi_{\mathrm{H}}(w)$ and the vertical combination $\psi_{\mathrm{V}}(w)$ are convex functions of the width w.

Proof. If $\psi(w)$ is constructed by linearly extending the convex function $\tilde{\psi}(w)$ as described in the theorem, then the function $\psi(w)$ is also going to be convex. And if the area of the horizontal and vertical combination may be formulated as given in (4.80) and (4.81), respectively, both the area of the horizontal and the vertical compositions are convex in w, as the maximum of several convex functions is also convex. The following discussion shows that the area of horizontal and vertical combinations may be represented by the formulations given in (4.80) and (4.81), respectively.

Considering the illustration given in Figure 4.16a and the definitions given in (4.55), (4.61), (4.73) and (4.75) the following conclusions may be derived:

$$\left.\frac{d}{dw}\psi_{\mathrm{H_{X1}}}\right|_{w=w_{\mathrm{I}}} \leq \left.\frac{d}{dw}\psi_{\mathrm{H_E}}\right|_{w=w_{\mathrm{I}}} \tag{4.82}$$

$$\left.\frac{d}{dw}\psi_{\mathrm{H_E}}\right|_{w=w_{\mathrm{II}}} \leq \left.\frac{d}{dw}\psi_{\mathrm{H_{N1}}}\right|_{w=w_{\mathrm{II}}} \tag{4.83}$$

Similarly, considering the illustration given in Figure 4.16b and the definitions given in (4.55), (4.71), (4.78) and (4.79) the following conclusions may be derived:

$$\left.\frac{d}{dw}\psi_{\mathrm{V_E}}\right|_{w=w_{\mathrm{I}}} \leq \left.\frac{d}{dw}\psi_{\mathrm{V_{X1}}}\right|_{w=w_{\mathrm{I}}} \tag{4.84}$$

$$\left.\frac{d}{dw}\psi_{\mathrm{V_{X1}}}\right|_{w=w_{\mathrm{II}}} \leq \left.\frac{d}{dw}\psi_{\mathrm{V_{X2}}}\right|_{w=w_{\mathrm{II}}} \tag{4.85}$$

Thus, the *max* function in (4.80) and (4.81) selects the correct functions for each region as indicated in Figure 4.16b. This proves that the horizontal and the vertical combinations may be formulated with (4.80) and (4.81), respectively. □

The convexity of horizontal and vertical combinations have been proven. The following theorem generalizes the proof to any slicing floorplan, composed of horizontal and vertical combinations.

Theorem 4.8. *The area of a compact slicing floorplan is a convex function of its width* w.

Proof. As defined at the beginning of this section, a compact layout has the shortest possible height for a given width w. Thus, the height $\mathrm{h}(w)$ of any layout combination, is a non-increasing function of the width w, as stated in (4.55). In Theorem 4.7, the area of horizontal and vertical combinations, $\psi_{\mathrm{H_E}}$ and $\psi_{\mathrm{V_E}}$, are proven to be convex in w. Thus, any combination satisfies the conditions listed in this section. It is concluded that, a horizontal or vertical combination of combinations also satisfies the mentioned conditions. Due to the hierarchy of slicing floorplans, area of any compact slicing floorplan is convex in w. □

Because of the fact that dimensions of symmetric blocks depend on each other, this case is not covered in Theorem 4.8. Symmetric placement and proximity constraints are covered in the following theorem.

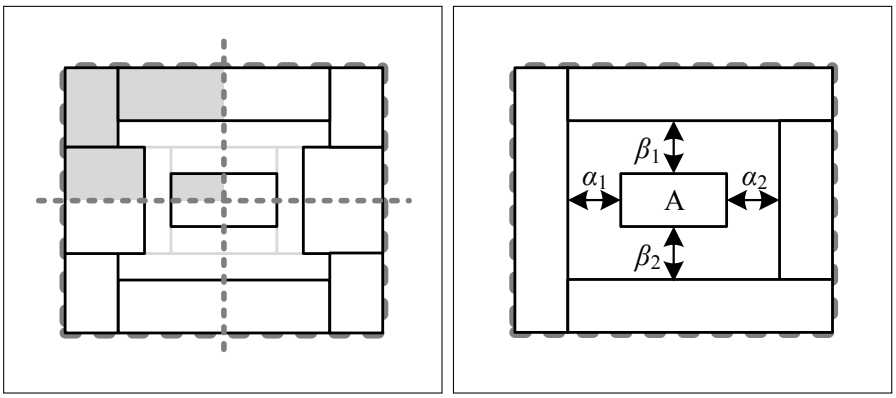


Figure 4.17. (a) Combination Composed of Symmetric Blocks, (b) Proximity Constraints Between the Block A and its neighbors Shown.

Theorem 4.9. *The area of a compact slicing floorplan with symmetry constraint, in a combination composed of symmetric blocks, and with proximity constraint, between a block and its neighbors, is a convex function of its width w.*

Proof. A symmetric slicing floorplan is shown in Figure 4.17a, where the symmetry axes are indicated by the dashed lines. If a horizontal or (and) vertical axis passes through a block, it cuts the area of this block into two (four) equal sub-blocks. The area of each sub-block, as a function of its width, may be obtained by scaling the area of the original block, convex by the assumption listed in this section, with a positive constant. Thus, the area of the sub-block is convex in its width and satisfies the listed conditions. As a result, the symmetry axes partition the layout into four quarters and each is composed of blocks and sub-blocks satisfying the conditions in this section. In Figure 4.17a, the blocks and sub-blocks in the upper-left quarter are indicated with gray regions. Theorem 4.8 states that the area of a quarter is convex in its width and its height is a non-increasing function of its width. The function for the total area may be obtained by scaling the function for the area of a quarter. Thus, the total area of the layout, dashed rectangle in Figure 4.17a, is convex in its width. If the width (height) of a quarter increases, the width (height) of the layout also increases. It may be concluded that the height of the layout is a non-increasing function of its width as

stated in (4.55). In conclusion, a combination composed of symmetric blocks may be considered as a super block in a compact layout and Theorem 4.8 states the convexity.

A layout with proximity constraints is shown in Figure 4.17, where the block A is constrained. Let us, replace the block A with a larger block $\tilde{\mathrm{A}}$ which satisfies the following:

$$w = w_A + \alpha \tag{4.86}$$

$$\mathrm{h}(w) = \mathrm{h_A}(w_A) + \beta \tag{4.87}$$

where α and β are defined as:

$$\alpha = \alpha_1 + \alpha_2 \tag{4.88}$$

$$\beta = \beta_1 + \beta_2 \tag{4.89}$$

The area of the larger block is:

$$\psi(w) = (w - \alpha) \cdot \mathrm{h_A}(w - \alpha) + (w - \alpha) \cdot \beta + \alpha \cdot \mathrm{h_A}(w - \alpha) + \alpha \cdot \beta \tag{4.90}$$

The first product in the sum resembles (4.75) and is convex in w. The second product in (4.90) is a linear term and it also convex in w. The third product is the scaled and shifted version of $\mathrm{h_A}(w_A)$, proven to be convex in Theorem 4.2, thus the third product is also convex. As a result, area of the larger block $\tilde{\mathrm{A}}$ is convex in w. If the width (height) of the block A increases, the width (height) of the block $\tilde{\mathrm{A}}$ also increases. Thus, the block $\tilde{\mathrm{A}}$ satisfies the conditions listed and Theorem 4.8 states the convexity of the layout where the block A is replaced with the block $\tilde{A}$. This concludes that the area of the layout is convex if there exists proximity constraints between a block and its neighbors. □

All the previous results also apply if the height h is chosen as the free parameter rather than the width w.

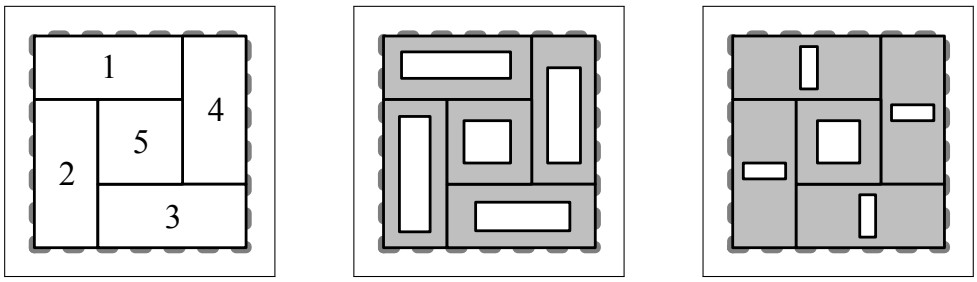


Figure 4.18. (a) Basic Non-Slicing Floorplan, (b) A Realistic Case, (c) An Unrealistic Case. In (b) and (c), Gray Area Represents Overhead, The White Area Represent Flexible Regions.

4.3.2. Convexity of General Floorplans

Area of each analog component is exactly modeled with continuous convex functions and the convexity of the complete floorplan is analyzed. In this section, convexity of the layout area is demonstrated for the typical example of non-slicing floorplan shown in Figure 4.18a. An analogous derivation can be performed for other non-slicing floorplans.

Theorem 4.10. *For all practical cases, the area of the non-slicing floorplan in Figure 4.18a is convex in its width w, if each component may be modeled by (4.8).*

Proof. According to Theorem 4.1, if each component can be modeled by equation (4.8), its area ψ_i

$$\psi_i = w_i \cdot h_i \tag{4.91}$$

is convex in its width w_i. Therefore, according to (4.6):

$$\frac{d^2\psi_i}{dw_i^2} = 2 \cdot \frac{dh_i}{dw_i} + w_i \cdot \frac{d^2h_i}{dw_i^2} \geq 0 \tag{4.92}$$

Thus, the following needs to be satisfied

$$\frac{d^2 h_i}{dw_i^2} \geq -\frac{2}{w_i} \cdot \frac{dh_i}{dw_i} \tag{4.93}$$

Mathematical relations between the component sizes of the floorplan in Figure 4.18a are:

$$h = h_1 + h_2 \tag{4.94}$$
$$h = h_3 + h_4 \tag{4.95}$$
$$h = h_1 + h_3 + h_5 \tag{4.96}$$
$$w = w_1 + w_4 \tag{4.97}$$
$$w = w_2 + w_3 \tag{4.98}$$
$$w = w_2 + w_4 + w_5 \tag{4.99}$$

where w and h are the width and the height of the complete layout, represented by the dashed rectangle in Figure 4.18a, respectively. Differentiating these equalities with respect to w and combining them, the following expression is obtained:

$$A \cdot x' = b \tag{4.100}$$

where the prime symbol stands for differentiation with respect to w, the matrix A is defined as:

$$A = \begin{bmatrix} \frac{dh_1}{dw_1} & \frac{dh_2}{dw_2} & 0 & 0 & 0 & -1 \\ 0 & 0 & \frac{dh_3}{dw_3} & \frac{dh_4}{dw_4} & 0 & -1 \\ \frac{dh_1}{dw_1} & 0 & \frac{dh_3}{dw_3} & 0 & \frac{dh_5}{dw_5} & -1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix} \tag{4.101}$$

vector x is:

$$x = \begin{bmatrix} w_1 & w_2 & w_3 & w_4 & w_5 & h \end{bmatrix}^T \tag{4.102}$$

and vector b is:

$$b = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}^T \tag{4.103}$$

Differentiating (4.100) with respect to w, the following is obtained:

$$A \cdot x'' + \frac{dA}{dw} \cdot x' = 0 \tag{4.104}$$

If the chain rule is used as follows:

$$\frac{d}{dw}\left[\frac{dh_i}{dw_i}\right] = w_i' \cdot \frac{d^2 h_i}{dw_i^2} \tag{4.105}$$

(4.104) may be rewritten as:

$$A \cdot x'' = -B \cdot \mathbf{diag}(x_1', x_2', \cdots, x_n') \cdot x' \tag{4.106}$$

where n is the size of the vector x and the matrix B is defined as:

$$B = \begin{bmatrix} \frac{d^2 h_1}{dw_1^2} & \frac{d^2 h_2}{dw_2^2} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{d^2 h_3}{dw_3^2} & \frac{d^2 h_4}{dw_4^2} & 0 & 0 \\ \frac{d^2 h_1}{dw_1^2} & 0 & \frac{d^2 h_3}{dw_3^2} & 0 & \frac{d^2 h_5}{dw_5^2} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{4.107}$$

Solving (4.100) for x' and substituting the solution in (4.106) and then solving (4.106) for h'', last entry in x'', the following is obtained:

$$h'' = \sum_i (w_i')^3 \cdot \frac{d^2 h_i}{dw_i^2} \tag{4.108}$$

Using (4.93) in (4.108) yields:

$$h'' \geq -\frac{2}{w} \cdot h' \cdot \sum_i (w_i')^2 \cdot \frac{w}{w_i} \cdot \frac{h_i'}{h'} \tag{4.109}$$

Using (4.97), (4.98) and (4.99), following may be shown to be valid:

$$(w_1')^2 \cdot \frac{w}{w_1} + (w_4')^2 \cdot \frac{w}{w_4} = 1 + \Delta_{1,4} \tag{4.110}$$

$$(w_2')^2 \cdot \frac{w}{w_2} + (w_3')^2 \cdot \frac{w}{w_3} = 1 + \Delta_{2,3} \tag{4.111}$$

$$(w_2')^2 \cdot \frac{w}{w_2} + (w_4')^2 \cdot \frac{w}{w_4} + (w_5')^2 \cdot \frac{w}{w_5} = 1 + \Delta_{2,4} + \Delta_{2,5} + \Delta_{4,5} \tag{4.112}$$

where

$$\Delta_{i,j} = \frac{(w_i \cdot w_j' - w_j \cdot w_i')^2}{w_i \cdot w_j} \tag{4.113}$$

Observing Figure 4.18a, the following equalities are obvious:

$$h_2 = h_3 + h_5 \tag{4.114}$$

$$h_4 = h_1 + h_5 \tag{4.115}$$

Using (4.96) and (4.110)-(4.115), it is possible to write (4.109) as follows:

$$h'' \geq -\frac{2}{w} \cdot h' \left(\frac{h'_1}{h'} + \frac{h'_3}{h'} + \frac{h'_5}{h'} + \delta \right) \tag{4.116}$$

where δ is defined as:

$$\delta = \frac{h'_1}{h'} \cdot \Delta_{1,4} + \frac{h'_3}{h'} \cdot \Delta_{2,3} + \frac{h'_5}{h'} \cdot (\Delta_{2,4} + \Delta_{2,5} + \Delta_{4,5}) \tag{4.117}$$

This theorem assumes that the height of the components may be model by equation (4.8), then, under the conditions stated in Theorem 4.1, the derivative of the height of the ith component h_i with respect to its width w_i is non-positive. Let us assume that w'_i is always non-negative. The expression h'_i may written as:

$$h'_i = w'_i \cdot \frac{dh_i}{dw_i} \leq 0 \tag{4.118}$$

Solving the system of equations in (4.100) for x', the derivative of the height h' is found to be non-positive which means h is a non-increasing function of w. Formally stating:

$$h' \leq 0 \tag{4.119}$$

Thus, the ratio of h'_i to h' is positive and the δ (4.117) is non-negative. Then, using (4.96) to rewrite (4.116), the following is obtained:

$$h'' \geq -\frac{2}{w} \cdot h'(1 + \delta) \geq -\frac{2}{w} \cdot h' \tag{4.120}$$

By comparing (4.93) with (4.120), it follows that the area ψ of the floorplan is convex in its width w. We still have to demonstrate that w'_i is non-negative. Solving the system in (4.100), the derivatives w'_1, w'_2, w'_3 and w'_4 turn out to be always positive. However, the sign of w'_5 is found to have the same sign as the following expression:

$$\frac{dh_2}{dw_2} \cdot \frac{dh_4}{dw_4} - \frac{dh_1}{dw_1} \cdot \frac{dh_3}{dw_3} \tag{4.121}$$

The proof of this theorem reduces to showing that (4.121) is positive. If the area of the components were constant, the area of the i-th component would be:

$$\psi_i = w_i \cdot h_i = a_i \tag{4.122}$$

where a_i is constant. Then h_i' is simply:

$$\frac{dh_i}{dw_i} = -\frac{a_i}{w_i^2} = -\frac{h_i}{w_i} \tag{4.123}$$

and (4.121) yields:

$$\frac{1}{r_2} \cdot \frac{1}{r_4} - \frac{1}{r_1} \cdot \frac{1}{r_3} \tag{4.124}$$

where r_i is the aspect ratio of the ith component

$$r_i = \frac{w_i}{h_i} \tag{4.125}$$

From Figure 4.18a the following inequalities are obvious:

$$w_1 \geq w_2, w_3 \geq w_4, h_2 \geq h_3, h_4 \geq h_1 \tag{4.126}$$

using these in (4.124), the sign of (4.124) is found to be always positive, which in turn indicates that the sign of (4.121) is positive. Thus, the total area is a convex function of the width w. However, if the areas of the components ψ_i are not constants, but the relation in (4.8) holds for them, after differentiating (4.8), (4.121) may be shown to be:

$$\frac{1}{\bar{r}_2} \cdot \frac{1}{\bar{r}_4} - \frac{1}{\bar{r}_1} \cdot \frac{1}{\bar{r}_3} \tag{4.127}$$

where $\bar{r}_i$ is:

$$\bar{r}_i = \frac{w_i - k_{i2}}{h_i - k_{i3}} \tag{4.128}$$

This expression resembles (4.124), where constants k_{i2} and k_{i3} are subtracted from the width w_i and height h_i, respectively. A realistic case and an unrealistic one are shown in Figure 4.18b and Figure 4.18c, respectively. The gray regions in these figures represent the overhead related to the constants k_i and these are due to technology rules, routing, dummies, etc. It is safe to conclude that equation (4.127) is positive in realistic cases, which concludes that the floorplan given in Figure 4.18a is convex in w. □

Considering h as the free parameter, the same proof concludes that the area of the floorplan, given in Figure 4.18a, is convex in h. Also, the numerical results given in Section 4.4.3 confirm that the total area, the solution space, is convex in w and h.

4.4. Optimization Technique

This section presents a methodology for optimizing the area of a general fixed floorplan for an analog circuit. The methodology also satisfies placement constraints such as symmetry. A sequential convex optimization method is presented to optimize the area. The computational cost and the resulting area are compared with the results of other approaches. The proposed methodology offers high quality solutions with a low run time cost and is a good candidate to be used in a layout-aware design loop. Generally speaking, the area ψ

$$\psi = w \cdot h \tag{4.129}$$

is not a convex function of the width w or the height h. Available convex programming solver allows only convex functions as the objectives, thus, the function (4.129) can not be formulated in the convex solver. Note that, in general, the area function in (4.129) is not convex; however, the area of a floorplan composed of the proposed components in Section 4.2 is convex in w and h. In other words, the height-width relations of the components constrain the problem such that the resulting solution space becomes convex. Although, the function in (4.129) can not be formulated in the convex solver,

we overcome the restriction by solving a sequence of convex problems. The objective function ψ is approximated and iteratively updated. The following theorem proves that the approximated objective function $\tilde{\psi}$ and the exact objective function ψ behave similarly around the optimal point.

Theorem 4.11. *If the optimal aspect ratio r_{opt} is known to be close to the estimated aspect ratio $\tilde{r}$, the objective function ψ may be approximated by the affine function*

$$\tilde{\psi} = \lambda \cdot \left(\frac{w}{\tilde{r}} + h \right) \tag{4.130}$$

where w and h are the width and height of the layout, respectively and $\lambda \in \mathbf{R}$ is a constant and may be chosen as 1.

Proof. The gradient for the approximate objective function $\tilde{\psi}$ is:

$$\nabla\tilde{\psi} = \left(\frac{\partial\tilde{\psi}}{\partial w}, \frac{\partial\tilde{\psi}}{\partial h} \right) = \left(\frac{\lambda}{\tilde{r}}, \lambda \right) \tag{4.131}$$

and the gradient for the exact objective function ψ, which is the floorplan area, is

$$\nabla\psi = \left(\frac{\partial\psi}{\partial w}, \frac{\partial\psi}{\partial h} \right) = (h, w) \cong (h, \tilde{r} \cdot h) \tag{4.132}$$

Although the magnitudes of the two gradients in (4.131) and (4.132) are different, their directions are close to each other, if the optimal aspect ratio is close to $\tilde{r}$. Therefore, minimization of (4.130) will also minimize the area. □

For instance, if r_{opt} is known to be close to unity, then (4.130) can be written as

$$\tilde{\psi} = 2 \cdot (h + w) \tag{4.133}$$

which is simply the perimeter. In this case, minimizing the perimeter will also minimize the area.

4.4.1. Methodology

Unfortunately, we do not know the optimal aspect ratio a priori, so we can not directly use the objective function in (4.130). Therefore, we applied an iterative approach. First, the aspect ratio r in (4.130) is set to an initial guess r_0. If r_0 is not specified, it is initialized to unity ($r_0 = 1$) and the area is optimized. Resulting dimensions (w_1 and h_1) are used to calculate the aspect ratio ($r_1 = w_1/h_1$) for the next iteration until r converges. The objective function may be formulated as:

$$\tilde{\psi}_n = \frac{w}{r_{n-1}} + h \tag{4.134}$$

where r_{n-1} is

$$r_{n-1} = \frac{w_{n-1}}{h_{n-1}} \tag{4.135}$$

For the bottom component in Figure 4.9a, changes in the objective function, according to (4.134), are plotted in Figure 4.19. For this component, optimal area is obtained when $m = 3$ and this may be also be observed from Figure 4.19, where the minimum of the objective function shifts to 3.

In the final implementation, the objective function in (4.134) is slightly modified by using the aspect ratio r_{n-2} to improve the convergence of the method:

$$\tilde{\psi}_n = \frac{w}{r_{n-2} + k * (r_{n-1} - r_{n-2})} + h \tag{4.136}$$

where $k \in \mathbf{R}$ and $1 \leq k$. The objective function in (4.136) boils down to (4.134) for $k = 1$. Parameter k is empirically chosen to be $k = 1.3$. Our algorithm stops when changes in the aspect ratio Δr is small, more formally:

$$\Delta r = r_n - r_{n-1} \leq \epsilon \cdot r_n \tag{4.137}$$

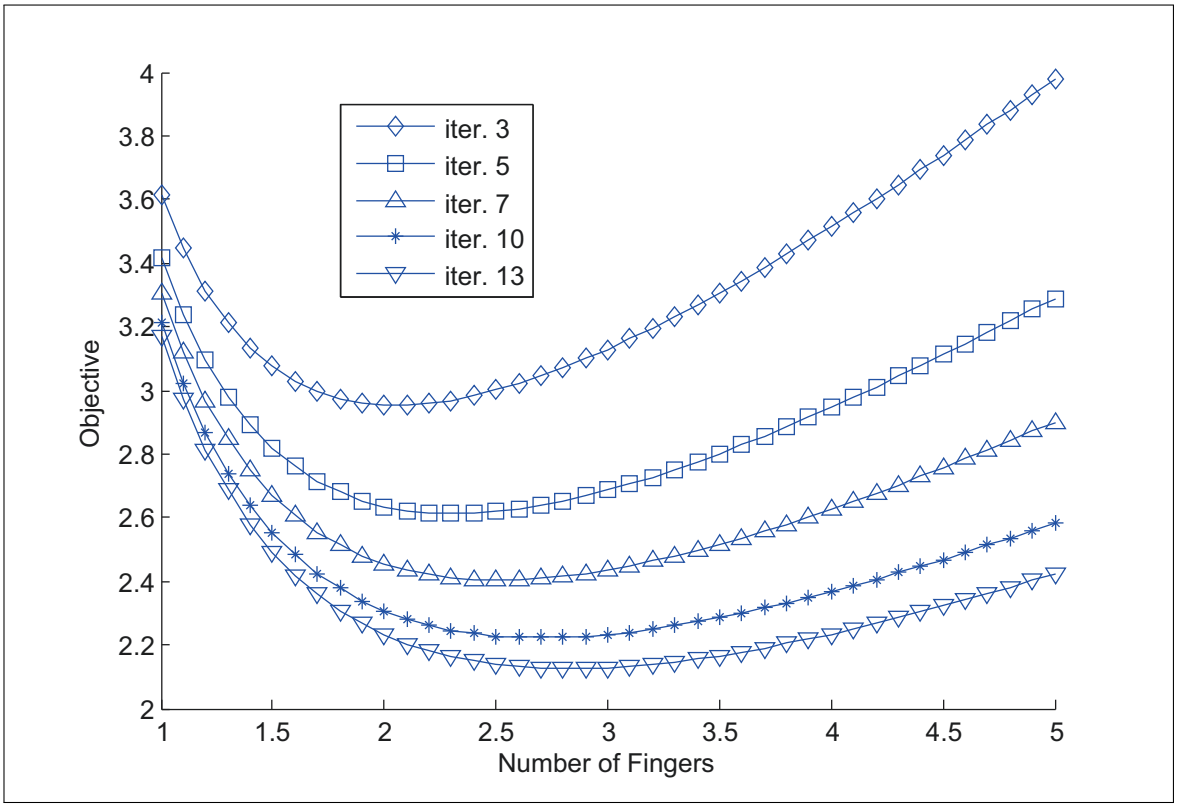


Figure 4.19. Change in Objective Function.

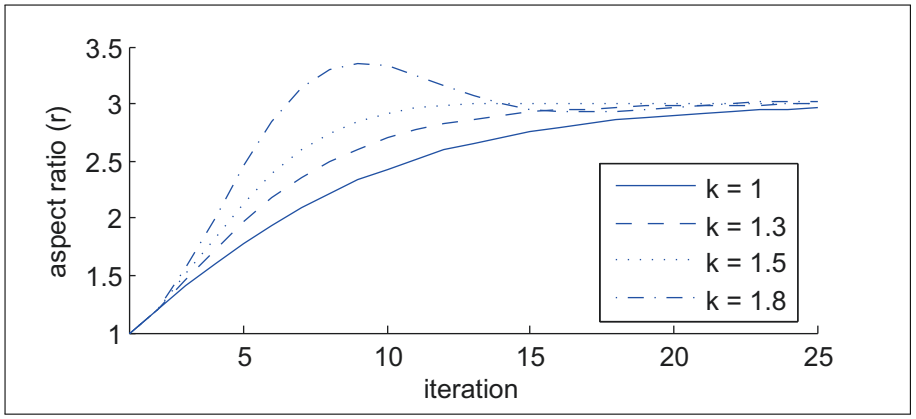
and ϵ is chosen as 0.01.

The proposed optimization methodology supports only continuous parameters; however, some of these are integers, such as the number of fingers m for the transistors. We round these parameters to the closest integer values such that none of the constraints are violated, thus, the solutions are still feasible. After rounding integer variables, their values are fixed and the problem is optimized one more time by using only the free continuous variables to further improve the solution.

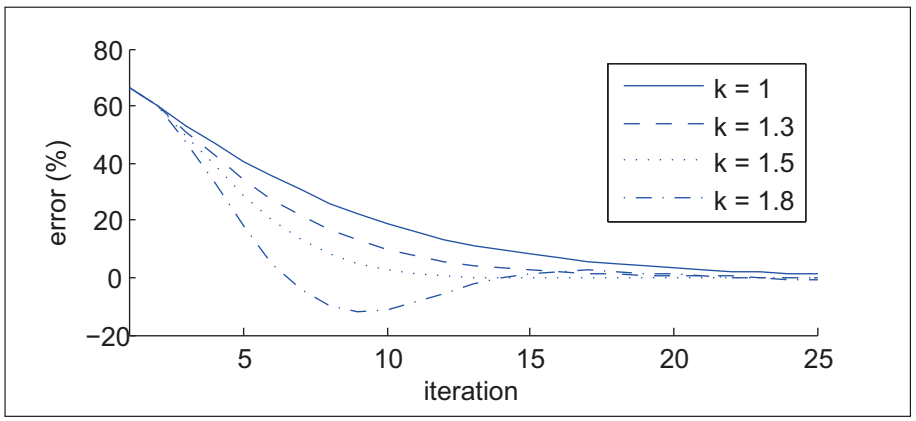
The procedure can be summarized as follows:

(i) For the first iteration, the specified affine function is used as the objective function. If not specified, the perimeter is used as the initial objective function.

(ii) Iteratively, solutions for width and height are used to improve approximation of the area with the objective function.



a



b

Figure 4.20. (a) Convergence of r to $r_{opt} = 3$, (b) Percentage Error.

(iii) When the solution converges, iterative process stops.

(iv) Integer parameters are rounded to nearest feasible integer values and the problem is optimized one more time by using the continuous parameters.

4.4.2. Convergence

Convergence of the method is tested for several examples. The number of iterations mainly depends on the optimal aspect ratio r_{opt}; the closer r_{opt} to unity, the fewer the iterations. For the bottom component in Figure 4.9a, $r_{opt} = 3$, convergence of the method and percentage error are plotted in Figure 4.20 for different k values in (4.136). By experience, $k = 1.3$ is a good choice for k. Convergence is observed to be fast initially; however, slows down when the error is small.

Table 4.2. Comparison of LDS with Other Area Minimization Methods.

	Area & Time	op-amp	ami-33	ami49	ami49
# Flexible Comp.	-	7	7	7	8
SCP (Continuous)	Area (m^2)	4.52e-9	1.26e-6	3.87e-5	3.83e-5
	Time (s)	0.010	0.021	0.039	0.042
SCP + Rounding	Area (m^2)	4.56e-9	1.27e-6	3.87e-5	3.84e-5
	Time (s)	0.013	0.031	0.057	0.061
Sim. Annealing	Area (m^2)	4.99e-9	1.45e-6	5.51e-5	5.01e-5
	Time (s)	0.087	0.034	0.043	0.037
ESF Addition	Area (m^2)	4.88e-9	1.89e-6	6.05e-5	6.87e-5
	Time (s)	0.008	0.004	0.006	0.006
Hier. Enumeration	Area (m^2)	4.56e-9	1.26e-6	3.87e-5	3.84e-5
	Time (s)	2.135	1347	1226	18019
Enumeration	Area (m^2)	4.56e-9	1.26e-6	3.87e-5	3.84e-5
	Time (s)	106.9	852.1	1092	19530

4.4.3. Results

The methodology is tested here on four examples:

(i) Two-stage op-amp designed in UMC 0.13μm technology with 7 soft blocks
(ii) The ami-33 layouts from the MCNC benchmark suite with 7 soft and 26 fixed blocks
(iii) The ami-49 layouts from the MCNC benchmark suite with 7 soft and 42 fixed blocks
(iv) The ami-49 layouts with 8 soft and 41 fixed blocks

The area of the op-amp is optimized with the components being modeled by the convex models introduced in Section 4.2. For the ami-33 and ami-49 layouts, component models are not available; therefore, we assigned realistic models for these components. In each example, there exist 7 or 8 soft components and the sizes of the rest are fixed.

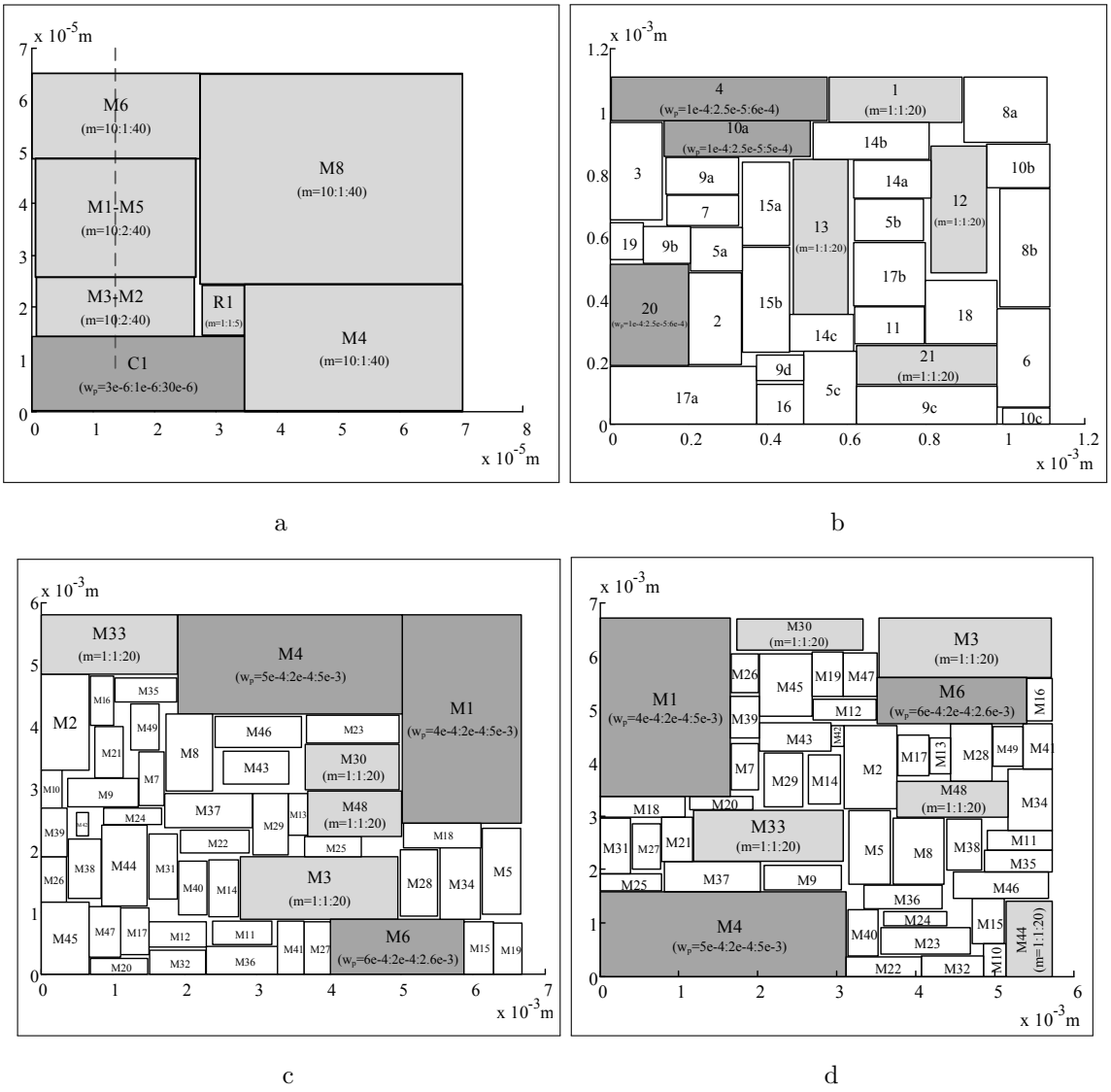


Figure 4.21. Sample Layouts (a) Placement Op-Amp , (b) Placement for ami-33 from MCNC Bench., (c) and (d) Placements for ami-49 from MCNC Bench. Flexible Modules are Filled: Transistors and Resistors in Light Gray, Capacitors in Dark Gray.

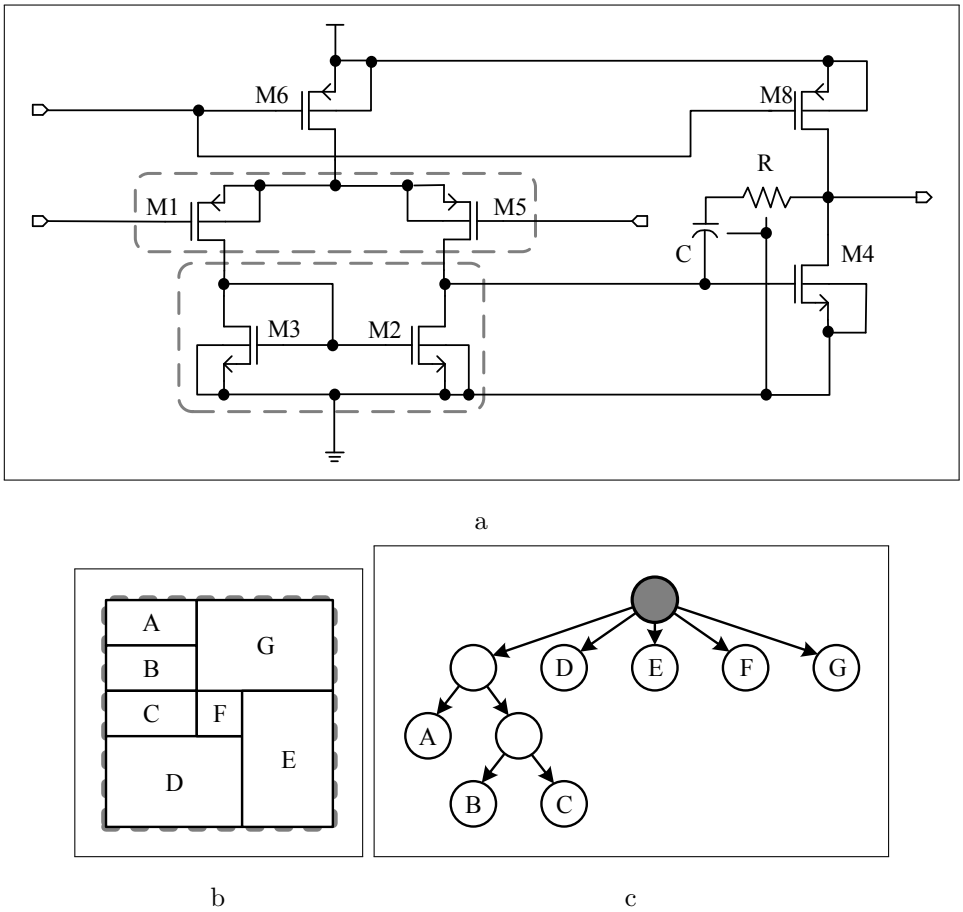


Figure 4.22. Op-Amp Circuit (a) Schematic, (b) Floorplan (c) Hierarchy.

Sample layouts are given in Figure 4.21 in which the soft transistors and resistors are filled with light gray and the soft capacitors are filled with dark gray. The labels on these soft components indicate the free parameters and their range, e.g. the label of M1-M5 component in Figure 4.21a is

$$(m = 10 : 2 : 40) \tag{4.138}$$

which indicates that the number of fingers m is defined in the range $[10, 40]$ and may have one of the values in the set $\{10, 12, 14, ..., 38, 40\}$.

The proposed methodology targets to minimize the area of a layout where the floorplan is known. For the op-amp circuit in Figure 4.22a, the floorplan in Figure 4.22b is used and the corresponding hierarchy is given in Figure 4.22c, where the top gray-filled node represents the layout boundary, which is the dashed rectangle

in Figure 4.22b. The floorplans for ami-33 and ami-49 have been randomly synthesized by a simulated annealing algorithm; thus, the hierarchies in these depend on the synthesized floorplans and due to randomness, there is no hierarchy between most components.

The performance of our methodology and the quality of the solution are compared with the results of other approaches in Table 4.2. The first row lists the results for the proposed sequential convex optimization (SCP) approach, where all the parameters are considered as continuous variables, although the number of fingers m must have integer values. The second row is obtained after rounding the number of fingers to feasible integer values (SCP + Rounding). The results are compared with those of the following methodologies:

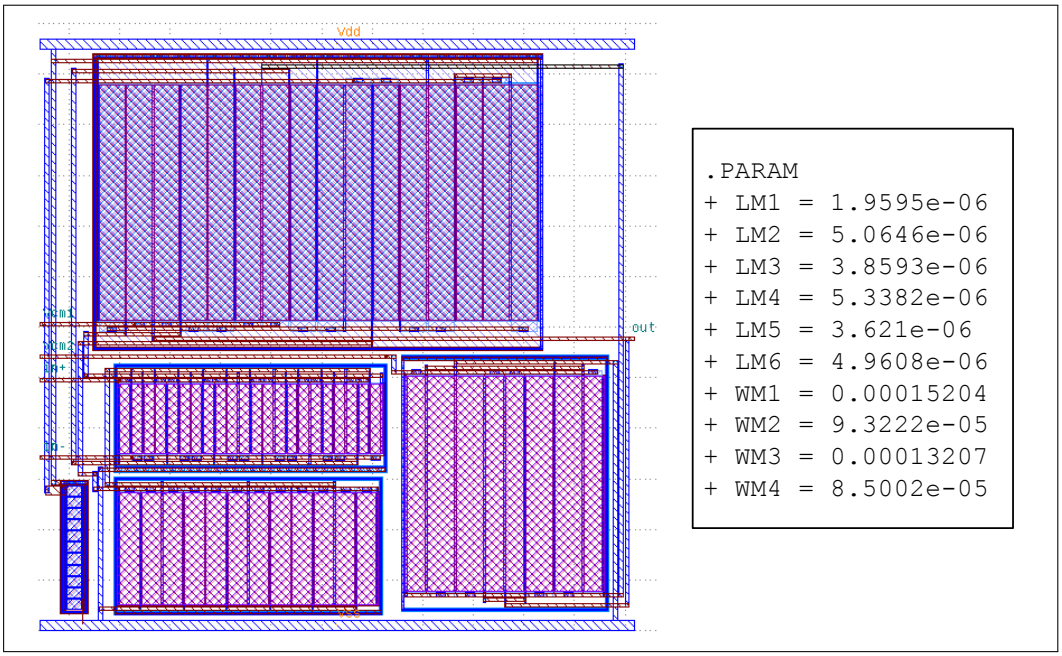
(i) A general probabilistic optimization method, simulated annealing (SA), where the optimization parameters are tuned by trial and error.
(ii) The enhanced shape function (ESF) addition method described in [51] where the solution with the minimum area is chosen from the resulting ESF.
(iii) A hierarchical enumeration methodology: given a floorplan, shape functions of the slicing parts are combined using the methodology in [46] and the shape functions for the non-slicing part are combined by enumeration. From the resulting shape function, the solution with the minimum area is chosen.
(iv) A simple enumeration. All possible solutions are tried in a loop and the one with the minimum area is kept.

The area resulting from the simulated annealing and the ESF addition approaches are relatively large compared to the proposed methodology, whereas the run time of the ESF addition methodology is the smallest. If hierarchy exists, the hierarchical enumeration methodology performs faster than the simple enumeration. However, its run time performance is worse when there is no hierarchy between most of the components, due to the overhead of shape function construction. The SCP methodology finds the same or better solutions than these enumeration methodologies because of the fact that some parameters are used as continuous variables. When the parameters,

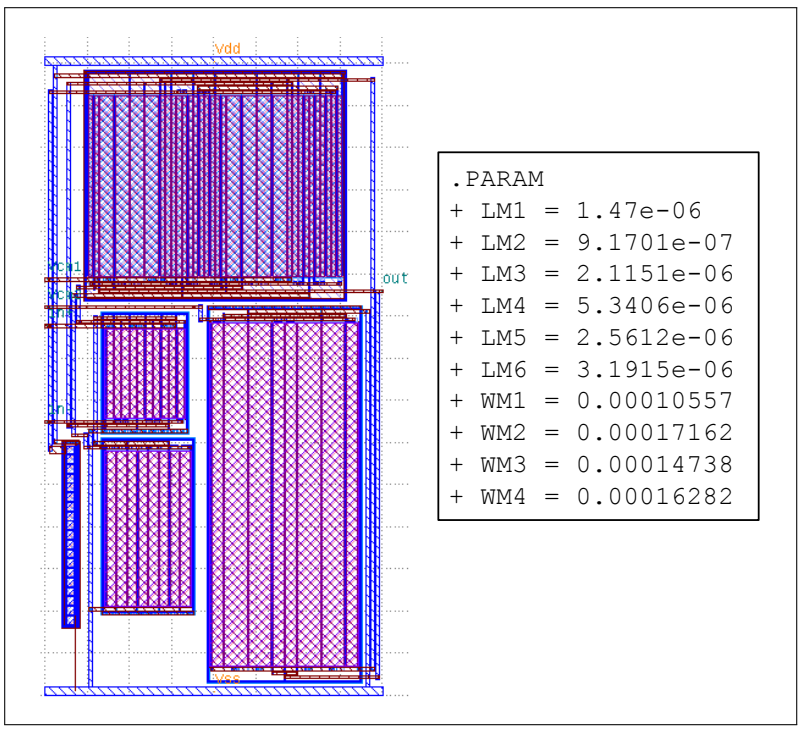
constrained to have integer values, are rounded, the solutions are very close to enumeration techniques and the run time is considerably smaller. All experiments were coded in LDS [43] and were executed on an Intel i7-3610QM processor with 6GByte RAM.

4.5. Folded-Amplifier Layout

The methodology presented in this chapter is applied to a folded cascode amplifier. The template for this amplifier was introduced in Chapter 3.3.6 and the schematic was shown in Figure 3.11. The geometrical parameter, number of fingers, of the resistor is allowed to have an integer value in the range $[1, 20]$. The number of fingers for the interdigitized transistors M1, M9, M8, M7, M6, M15, M16 and, M14 is restricted to have even integer value in the range $2, 20$. The transistor pairs M2 and M3, M4 and M5 have number of fingers as the free geometrical parameters and they take integer values in the range $[2 : 40]$. The number of fingers for the interdigitized transistors M12, M13, M10 and, M11 is limited to take integer values in the range $[2 : 40]$. Sample instances with minimum area are shown in Figure 4.23.



a



b

Figure 4.23. Folded Amplifier with Minimum Area (a) Instance I, (b) Instance II. Template Presented in Chapter 3.3.6 and Values of Parameter Given in (a) and (b).

5. CONCLUSION

This thesis presents the Layout Description Script (LDS), which is a domain specific language (DSL) intended to code layout templates to be used for layout-aware circuit synthesis. LDS supports both sequential and constraint programming and is suitable for both manual coding and automatic code generation. LDS is compared with previous approaches related to layout description.

Code samples are given for alignment, abutment, symmetry, and similar constraints. A methodology to solve LDS constraints is presented. Also a hierarchical constraint satisfaction methodology is given and its computational cost is compared with two non-hierarchical methodologies.

Data structures in LDS are explained and implementation of the LDS compiler is discussed. An integrated design environment to code LDS templates is implemented and screen shots are shown.

Capture tool has been implemented. This tool is designed to extract a template from an expert-drawn layout. Capture converts a data structure extracted through a guided user interface (GUI) into an LDS template. This tool highlights the compatibility of LDS with electronic design automation (EDA).

Due to its support for constraint programming, it is possible to constrain topological representations and even combine them. Algorithms to convert topological representations to LDS constraints and explanatory examples are given.

Convex functions have a single minimum and their optimal solution may be found very efficiently. Geometric layout models for transistors, capacitors and resistors are developed and the areas of these components are shown to be convex functions in their width and height.

It is proven that the area of a compact slicing floorplan is a convex function of its width (height), where the area of each component is formulated as continuous convex functions. Only two assumptions are made on these functions, which are readily satisfied by analog components, such as transistors, capacitors and resistors. Possible cases are considered, with and without dead space.

Also the convexity of a non-slicing floorplan area is demonstrated. A sequential convex programming methodology is presented that is capable of finding the optimum dimensions of the layout components under integer relaxation. During the optimization, analog constraints, e.g., symmetry, are also satisfied. The proposed methodology is tested on three layouts. The computation times and the area of the resulting layouts are compared with alternative approaches. The quality of the solutions provided by the proposed approach is much better than alternative approaches, e.g. simulated annealing, ESF, with comparable computation times. The quality is comparable to enumeration approaches but orders of magnitude faster.

Summing up, a language for template coding is designed and its properties are highlighted on realistic examples. The language may be extended to non-orthogonal shapes. For widespread acceptance of the language, compatible EDA tools should be implemented. The contributions of this thesis may be listed as follows:

- A template language, LDS, is presented and compared with other layout representations in Chapter 2.
- Necessary developer tools, compiler and editor, for LDS are implemented and explained in Chapter 2.
- A tool, Capture, is implemented to synthesize templates from available layouts and presented in Chapter 3.
- Algorithms are developed to convert topological representation [27–29] to LDS code and explained in Chapter 3.
- Device generators for the UMC 0.13μm technology are coded in LDS and presented in Chapter 3.

- Dimensions of analog devices are modeled and the area of these devices are shown to be convex in Chapter 4.
- Under realistic assumptions, area minimization problem on a fixed floorplan is shown to be a convex problem in Chapter 4.
- A new methodology is developed to minimize the area of a layout. The methodology is compared with other approaches in Chapter 4.

REFERENCES

1. Rutenbar, R., "Analog Layout Synthesis: What's missing?", *International Symposium on Physical Design*, San Francisco, CA., 2010.

2. Rijmenants, J., J.B. Litsios, T.R. Schwarz, and M.G.R. Degrauwe, "ILAC: An Automated Layout Tool for Analog CMOS Circuits", *IEEE Journal of Solid State Circuits*, Vol. 24, No. 2, pp. 417-425, 1989.

3. Cohn, J., D. Garrod, R. Rutenbar, and L.R. Carley, *Analog Device-Level Layout Generation*, Kluwer Academic, Boston, MA., 1994.

4. Lampaert, K., G. Gielen, and W.M. Sansen, "A Performance-Driven Placement Tool for Analog Integrated Circuits", *IEEE Journal of Solid-State Circuits*, Vol. 30, No. 7, pp. 773-780, 1995.

5. Malavasi, E., E. Charbon, E. Felt, and A. Sangiovanni-Vincentelli, "Automation of IC Layout with Analog Constraints", *IEEE Transactions on Computer-Aided Design*, Vol. 15, No. 8, pp. 923-942, 1996.

6. Tang, H., H. Zhang, and A. Doboli, "Layout-Aware Analog System Synthesis based on Symbolic Layout Description and Combined Block Parameter Exploration, Placement and Global Routing", *Proceedings of IEEE Computer Society Annual Symposium*, pp. 266-271, 2003.

7. Habal, H., and H. Graeb, "Constraint-Based Layout-Driven Sizing of Analog Circuits", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 30, No. 8, pp. 1089-1102, 2011.

8. Martin, R., N. Lourenço, and N. Horta, "LAYGEN-II: Automatic Analog IC Layout Generator Based on Template Approach", *Proceedings of Genetic and Evolutionary Computation Conference*, pp. 1127-1134, 2012.

9. Agarwal, A., H. Sampath, V. Yelamanchili, and R. Vemuri, "Fast and Accurate Parasitic Capacitance Models for Layout-Aware Synthesis of Analog Circuits", *Proceedings of Design Automation Conference*, pp. 145-150, 2004.

10. Onedera, H., H. Kanbara, and K. Tamaru, "Operational Amplifier Compilation with Performance Optimization", *IEEE Journal of Solid-State Circuits*, Vol. 25, No. 2, pp. 466-473, 1990.

11. Dessouky, M., and M. Louerat, "A Layout Approach for Electrical and Physical Design Integration of High-Performance Analog Circuits", *Proceedings of the IEEE International Symposium on Quality Electronic Design*, pp. 291-298, 2000.

12. Vancorenland, P., G. Van der Plas, M. Steyaert, G. Gielen, and W. Sansen, "A Layout-Aware Synthesis Methodology for RF Circuits", *Proceedings of International Conference on Computer Aided Design*, pp. 358-362, 2001.

13. Ranjan, M., W. Verhaegen, A. Agarwal, H. Sampath, R. Vemuri, and G. Gielen, "Fast, Layout-Inclusive Analog Circuit Synthesis using Pre-Compiled Parasitic-Aware Symbolic Performance Models", *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 604-609, 2004.

14. Castro-Lopez, R., O. Guerra, E. Roca, and F.V. Fernandez, "An Integrated Layout Synthesis Approach for Analog ICs", *IEEE Transactions on Computer-Aided Design*, Vol. 27, No. 7, pp. 1179-1189, 2008.

15. Rutenbar, R. A., and J. Cohn, "Layout Tools for Analog ICs and Mixed-Signal SoCs: A Survey", *Proceedings of the International Symposium on Physical Design*, pp. 76-83, San Diego, CA, 2000.

16. Gielen, G., and R.A. Rutenbar, "Computer-Aided Design of Analog and Mixed-Signal Integrated Circuits", *Proceedings of the IEEE*, Vol. 88, No. 12, pp. 1825-1852, 2000.

17. Meng-Chen, W., "Placement with Alignment and Performance Constraints using the B*-tree Representation", *Proceedings of the IEEE International Conference on Computer Design*, pp. 568-571, 2004.

18. Zhang, L., and Y. Jiang, "Global-Routing Driven Placement Strategy in Analog VLSI Physical Designs", *Proceedings of the IEEE International Symposium on Circuit and Systems*, Vol. 2, pp. 1239-1242, 2005.

19. Fu, N., S. Nakatake, and M. Mineshima, "Multi-SP: A Representation with United Rectangles for Analog Placement and Routing", *Proceedings of the IEEE Computer Society Annual Symposium*, Karlsruhe, 2006.

20. Tam, Y-C., E. Young, and C. Chu, "Analog Placement with Symmetry and Other Placement Constraints", *Proceedings of the ACM/IEEE International Conference on Computer Aided Design*, pp. 349-354, 2006.

21. Zhang, L., R. Raut, Y. Jiang, and U. Kleine, "Two-Stage Placement for VLSI Analog Layout Design", *Proceedings of the IEEE Circuits, Devices and Systems*, Vol. 153, No. 3, pp. 274-280, 2006.

22. Lourengo, N., M. Vianello, J. Guilherme, and N. Horta, "LAYGEN - Automatic Layout Generation of Analog ICs from Hierarchical Template Descriptions", *Research in Microelectronics and Electronics*, pp. 213-216, 2006.

23. Lin, P-H., and S-C. Lin, "Analog Placement based on Hierarchical Module Clustering", *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 50-55, 2008.

24. Yilmaz, E., and G. Dundar, "Analog Layout Generator for CMOS Circuits", *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, Vol. 28, No. 1, pp. 32-45, 2009.

25. Chen, H-Y., and Y-W. Chang, "Routing for Manufacturability and Reliability",

IEEE Circuits and Systems Magazine, Vol. 9, No. 3, pp. 20-31, 2009.

26. Graeb, H.E., *Analog Layout Synthesis: A Survey of Topological Approaches*, Springer, London, 2011.

27. Otten, R.H., "Automatic Floorplan Design", *Proceedings of Design Automation Conference*, pp. 261-267, 1982.

28. Murata, H., K. Fujiyoshi, S. Nakatake, and Y. Kajitani, "VLSI Module Placement based on Rectangle-Packing by the Sequence-Pair", *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, Vol. 15, No. 12, pp. 1518-1524, 1996.

29. Chang, Y.C., Y.W. Chang, G.M. Wu, and S.W. Wu, "B*-Trees: A New Representation for Non-Slicing Floorplans", *Proceedings of Design Automation Conference*, pp. 458-463, 2000.

30. Rubin, S.M., *Computer Aids for VLSI Design - Appendix B*, 1994, *http://www.rulabinsky.com/cavd/text/chapb.html*, [Accessed January 2013].

31. Rubin, S.M., 1994, *Computer Aids for VLSI Design - Appendix C*, 1994, *http://www.rulabinsky.com/cavd/text/chapc.html*, [Accessed January 2013].

32. Wood, G., and H-F. S. Law, "SKILL - An Interactive Procedural Design Environment", *Proceedings of Custom Integrated Circuits Conference*, pp. 544547, 1986.

33. Pilgrim, M., *Dive into Python*, 2004, *http://www.diveintopython.net/*, [Accessed January 2013].

34. Badaoui, R.F., H. Sampath, A. Agarwal, and R. Vemuri, "A High Level Language for Pre-Layout Extraction in Parasite-Aware Analog Circuit Synthesis", *Proceedings of Great Lakes symposium on VLSI*, pp. 271-276, 2004.

35. Wolf, M., U. Kleine, and J. Schulze, "New Description Language and Graphical

User Interface for Module Generation in Analog Layouts" *Proceedings of Circuits and Systems Symposium*, pp. 290-293, 1998.

36. Owen, B.R., R. Duncan, S. Jantzi, C. Ouslis, S. Rezania, and K. Martin, "BALLISTIC: An Analog Layout Language", *Proceedings of the IEEE Custom Integrated Circuits Conference*, pp. 41-44, 1995.

37. Croes, K., H.J. De Man, and P. Six, "CAMELEON: a Process-Tolerant Symbolic Layout System", *IEEE Journal of Solid-State Circuits*, Vol. 23, No. 3, pp. 705-713, 1988.

38. Cory, W.E., "Layla: A VLSI Layout Language", *Proceedings of the Design Automation Conference*, pp. 245-251, 1985.

39. Oster, G.M., and A.J. Kusalik, "Icola - Incremental Constraint-Based Graphics for Visualization", *Constraints; An International Journal*, Vol. 3, No. 1, pp. 33-59, 1998.

40. Roach, J.A., "The Rectangle Placement Language", *Proceedings of the Design Automation Conference*, pp. 405-411, 1984.

41. Lipton, R.J., S.C. North, J. Valdes, G. Vijayan, and R. Sedgewick, "ALI: A Procedural Language to Describe VLSI Layouts", *Proceedings of Design Automation Conference*, pp. 467-474, 1982.

42. Young, E.F.Y., C.C.N. Chu, and M.L. Ho, "Placement Constraints in Floorplan Design", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 12, No. 7, pp. 735-745, 2004.

43. Unutulmaz, A., G. Dundar and F.V. Fernandez, "Template Coding with LDS and Applications of LDS in EDA", *Analog Integrated Circuits and Signal Processing*, Vol. 78, pp. 137-151, 2014.

44. Boyd, S., and L. Vandenberghe, *Convex Optimization*, Cambridge University Press, New York, 2009.

45. R. Otten, "Automatic Floorplan Design", *Proceedings of the Design Automation Conference*, pp. 261-267, 1983.

46. R. Otten, "Efficient Floorplan Optimization", *Proceedings of the International Conference on Computer Design*, pp. 499-502, 1983.

47. Moh, T-S., T-S. Chang, and S.L. Hakimi, "Globally Optimal Floorplanning for a Layout Problem", *IEEE Transactions on Circuit and Systems I*, Vol. 43, pp. 713-720, 1996.

48. Rosenberg, E., "Optimal Module Sizing in VLSI Floorplanning by Nonlinear Programming", *Methods and Models of Operations Research*, Vol. 33, No. 2, pp. 131-143, 1989.

49. Chen, T., and M.K.H. Fan, "On Convex Formulation of the Floorplan Area Minimization Problem", *Proceedings the ACM International Symposium on Physical Design*, New York, NY, USA, pp. 124-128, 1998.

50. Chen, P., and Kuh E.S., "Floorplan Sizing by Linear Programming Approximation", *Proceedings of ACM/IEEE Design Automation Conference*, pp. 468-471, 2000.

51. Strasser, M., M. Eick, M., H. Grab, U. Schlichtmann, and F.M. Johannes, "Deterministic Analog Circuit Placement using Hierarchically Bounded Enumeration and Enhanced Shape Functions", *Proceedings of the International Conference on Computer-Aided Design*, pp. 306-313, 2008.

52. Qiang, M., X. Linfu, T. Yiu-Cheong, and E. Young, "Simultaneous Handling of Symmetry, Common Centroid and General Placement Constraints", *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, Vol. 30, No. 1, pp. 85-95, 2011.

53. Tsao H., P. Chou, S. Huang, Y. Chang, M. Lin, D. Chen, and D. Liu, "A Corner Stitching Compliant B*-tree Representation and its Applications to Analog Placement", *Proceedings IEEE/ACM International Conference on Computer Aided Design*, pp. 507-511, 2011.

54. Lin, C., H. Zhou, and C. Chu, "A Revisit to Floorplan Optimization by Lagrangian Relaxation", *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, New York, NY, USA, pp. 164-171, 2006.

55. Unutulmaz, A., G. Dundar, and F.V. Fernandez, "Area Optimization on Fixed Analog Floorplans using Convex Area Functions", *Proceedings of the Design Automation and Test in Europe Conference*, Grenoble, pp. 1843-1848, 2013.

56. Wang, T., and D.F. Wong, "Optimal Floorplan Area Optimization", *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, Vol. 11, No. 8, pp. 992-1002, Aug. 1992.

57. Okuda, R., T. Sato, H. Onodera, and K. Tamariu, "An Efficient Algortihm for Compaction Problem with Symmetry Constraints", *Proceedings of the IEEE International Conference on Computer Aided Design*, Santa Clara, CA, USA, pp. 148-151, 1989.

58. Mosek, *Optimizer API for Java*, 2015, https://mosek.com/resources/doc, [Accessed January 2015].

59. Applegate, D., et al., *QSopt*, 2006, http://www.math.uwaterloo.ca/bico/qsopt, [Accessed January 2015].

60. Oracle, *Java*, 2015, https://www.java.com/, [Accessed January 2015].

61. Parr, T., *ANTLR 4 Documentation*, 2014, http://www.antlr.org/, [Accessed January 2015].

62. NetBeans Community, *NetBeans Platform (RCP) and Module Development*, 2015, https://netbeans.org/, [Accessed January 2015].

63. Nakatake, S., K. Fujiyoshi, H. Murata, and Y. Kajitani, "Module Placement on BSG Structure and IC Layout Application", *Proceedings of the International Conference on Computer Aided Design*, pp. 484-491, 1996.

64. Guo, P. N., C. K. Cheng, and T. Yoshimura, "An O-tree Representation of Non-Slicing Floorplans and its Application", *Proceedings of the Design Automation Conference*, pp. 268-273, 1999.

65. Jai-Ming, L., and C. Yao-Wen, "TCG: a Transitive Closure Graph based Representation for Non-Slicing Floorplans", *Proceedings of the Design Automation Conference*, pp. 764-769, 2001.

66. Hong, X., G. Huang, Y. Cai, J. Gu, S. Dong, C.K. Cheng, and J. Gu, "Corner Block List: An Effective and Efficient Topological Representation of Non-Slicing floorplan", *Proceedings of the International Conference on Computer Aided Design*, pp. 8-12, 2000.

67. Sakanushi, K., and Y. Kajitani, "The Quarter-State Sequence (Q-Sequence) to Represent the Floorplan and Applications to Layout Optimization", *Proceedings of the IEEE Asia Pacific Conference on Circuits and Systems*, pp. 829-832, 2000.