TECHNIQUES FOR IC SYMBOLIC LAYOUT AND COMPACTION

by

Jeffrey Lyn Burns

Memorandum No. UCB/ERL M90/103

13 November 1990

TECHNIQUES FOR IC SYMBOLIC LAYOUT AND COMPACTION

by

Jeffrey Lyn Burns

Memorandum No. UCB/ERL M90/103

13 November 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TECHNIQUES FOR IC SYMBOLIC LAYOUT AND COMPACTION

by

Jeffrey Lyn Burns

Memorandum No. UCB/ERL M90/103

13 November 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Techniques for IC Symbolic Layout and Compaction

by

Jeffrey Lyn Burns

University of California
Berkeley, California

Department of Electrical Engineering
and Computer Sciences

Abstract

The task of creating the physical layout data for an integrated circuit is difficult, due to the large number (10^5 - 10^7) of geometries that must be created, and due to the large number of rules that govern the sizes, spacings, and overlaps of the geometries. As a result, various methods for automatic layout generation have been proposed; symbolic layout with layout compaction comprise one such method. A symbolic layout is a less-detailed abstraction of the final, physical layout. A layout compactor is an optimization program that maps a symbolic layout into a detailed physical layout, such that the layout rules are satisfied and the layout area is minimized.

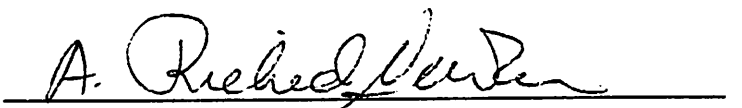
Three contributions to symbolic layout and layout compaction are presented. Symbolic-layout models have been investigated, and a new model is proposed which is technology and hierarchy-level independent. Most symbolic-layout models are specific to a technology and to a hierarchy level. Results are presented to demonstrate the ability of the proposed model to effectively capture a wider variety of layouts than other symbolic-layout models, including layouts from different technologies, hierarchy levels, and design styles.

A new constraint-generation algorithm is reported and shown to effectively complement the proposed symbolic-layout model. The constraint-generation module of a layout compactor analyzes the symbolic layout to construct the mathematical model of the layout that is subsequently used to drive the area optimization. A constraint generator must execute efficiently, yet it must also capture all the relevant detail of the symbolic layout. Results are presented which indicate that the proposed generator and model compare well with the more restrictive, standard methods, even though the proposed methods are more general.

A new, four-variable constraint, which significantly expands the domain of problems amenable to layout compaction, is proposed as is an efficient algorithm for solving mixed two and four-variable constraint problems. The types of constraints that can be satisfied by a layout compactor determine, in part, the set of designs to which the compactor

can be applied. The two-variable constraints used by most compactors are inadequate for some layouts, such as analog designs and pitch-matching hierarchical designs, where more complicated relationships between layout components must be satisfied. It is shown that the addition of the proposed four-variable constraint-type can solve problems of this nature. The addition of four-variable constraints significantly affects the addressed formulation of the compaction problem, transforming it from class P to class NP. As a result, a heuristic method for solving the mixed two and four-variable constraint problem has been developed. The results presented herein show that the proposed methods efficiently accommodate compaction problems that cannot be modeled or solved via the conventional two-variable model.

The presented models and algorithms have been implemented in a new layout-compaction program called SPARCS (Spacing Program with ARbitrary ConstraintS). Results obtained using SPARCS are included to demonstrate the utility of the proposed symbolic layout and layout compaction techniques. The SPARCS program is shown to be significantly more general than other compactors while achieving similar levels of runtime and area efficiency.



Professor A. Richard Newton
Thesis Committee Chairman

Acknowledgements

First, I thank Professor Richard Newton, my research advisor, for suggesting this project and for his guidance and support throughout its course.

I am grateful to Professor Alice Agogino and Professor Alberto Sangiovanni-Vincentelli for reading this dissertation, and for serving on my committee.

I especially thank Ron Gyurcsik for carefully reading the first draft of this dissertation. I also thank Brian Lee and Peter Moore for reading parts of it.

I thank Elise Mills for helping me file this dissertation, by tracking down people, obtaining signatures, and being the keeper of my title page.

Numerous discussions with several of my colleagues at Berkeley contributed to this project. I thank Tom Laidig for discussions on layout modeling. I thank Peter Moore for helping me formulate some of the concepts used in terminal merging. I thank Fabio Romeo for discussions on active constraints. I thank Rick Spickelmier for assisting in the development of the RPC interface to SPARCS, through discussions and by writing some of the code. I also thank the "local programming consultants" David Harrison, Tom Laidig, Peter Moore, Tom Quarles, and Rick Spickelmier for answering many questions.

A number of people contributed to the development of SPARCS by using it and suggesting enhancements. In this regard I thank Andrea Casotto, Chuck Kring, Bill Lin, and Han Koh in particular. Conversations on compaction with Mark Bales, David Boyer, Min-Yu Hsueh, Abdul Malik, Dr. Mori of Toshiba, Richard Newton, Alberto Sangiovanni-Vincentelli, Carlo Sequin, Hyunchul Shin, and Wayne Wolf over the years were very helpful.

The financial support of DARPA, the Digital Equipment Corporation, the State of California MICRO program, Raytheon Company, and Rockwell International is gratefully acknowledged.

I was fortunate to be among a great group of people during my time at Berkeley. I thank my friends Srinivas Devadas, Ron Gyurcsik, David Harrison, Mark Hofmann, Ken Keller, Chuck Kring, Tom Laidig, Brian Lee, Frank Ma, Karti Mayaram, Peter Moore, Tom Quarles, Fabio Romeo, Rick Rudell, and Rick Spickelmier for helping to make my stay here fun and worthwhile. Also, I thank Professors Richard Newton, Donald Pederson, and Alberto Sangiovanni-Vincentelli for creating an enjoyable and stimulating environment to work in.

I thank Ron and Peggy Gyurcsik, Mark Hofmann, Dan Ostapko, Fabio Romeo,

Albert Ruehli, and Louise Trevillyan for their encouragement over the last few months of this project.

I thank my parents, Julie and Mickey, for their consistent love and support during all my activities, not just this one.

Finally, I thank my wife Darlene for her love, encouragement, and optimism during this project and especially during the writing of this dissertation.

Contents

Chapter 1

Introduction

Integrated circuits (ICs) have steadily grown in complexity since their invention, and at present, very-large-scale integration (VLSI) has resulted in circuits ("chips") with more than one million transistors. Any system of such scale is not amenable to manual design. As a result a great deal of research has been conducted over the past 25 years in computer-aided design (CAD) methods for ICs [64].

An integrated circuit is a planar structure that is fabricated by a sequence of processing steps; each step modifies selected areas on the surface of the semiconductor wafer. A **layout** is the geometric data that is used in this process. The layout geometries define the areas that are modified at each step. The set of geometries used in a particular step is called a **mask**. Since several shapes are needed to define a basic electrical component, like a transistor, the layout for a one million transistor circuit may consist of several million geometries. The mask geometries must obey a large set of rules to insure that the circuit will function. The rules specify, e.g., sizes of geometries, spacings between geometries, and the overlap of one geometry by another. The requirement that the layout satisfy this large set of intricate rules exacerbates the problem of creating a layout.

Symbolic layout with layout compaction has developed as one means of automating the layout-creation phase of IC design. In this methodology the layout is described in an abstract, high-level fashion. The layout is then automatically transformed into its detailed geometric representation such that its area is minimized and such that the layout rules are satisfied.

The results of new research in symbolic layout and layout compaction are described in this dissertation. In particular, three contributions are presented. The existing symbolic-

layout models are typically hierarchy-level dependent, and technology parameterized, not technology independent. A new symbolic-layout model has been developed which is both technology and hierarchy-level independent; it is thus able to represent a much wider variety of IC designs. A new algorithm for constraint-graph generation is proposed. This algorithm exploits the ability of the layout model to capture a wide variety of designs, with runtime and area-efficiency results that are competitive with existing algorithms that operate on less general layout models. The two-variable constraints used in layout compactors are inadequate in some cases, such as analog designs with symmetric element configurations, because relationships must be enforced between more than two elements. For such problems, a new, four-variable constraint is proposed, as are algorithms for solving mixed two and four-variable constraint systems. It is shown that these new constraints and algorithms can effectively solve compaction problems, e.g., symmetric elements and hierarchy preservation, that cannot be solved using existing methods. These three contributions are summarized more fully in Section 1.5 of this Chapter.

1.1 Mask-Level Layout

The layouts for the first ICs were created by cutting openings in an opaque material, which was then reduced photographically to create the correctly-sized masks. Computer graphics was subsequently applied to the problem of entering mask data. A graphics terminal was used to capture geometries digitized by the user via a mouse or some other pointing device. This method of entering layout data manually is still used today in many circumstances. Manual creation of mask-level layout is a difficult problem because of the volume of data and because of the number of and complexity of the layout rules.

The need to manually enter mask-level layouts is gradually being eliminated; program-generated layouts are now common. A large investment is required to develop and maintain a program that generates high-quality layout at the mask level. Part of the development cost is due to the need to incorporate detailed knowledge of the layout rules into the program. A large part of the maintenance cost is due to the fact that fabrication processes constantly evolve, hence the layout rules constantly change. A program that generates mask layout must be changed to accommodate the changing rules. The cost of tracking fabrication-technology changes will continue to increase as more layout is program-generated, if layout generators are required to produce mask-level layout.

Integrated circuits are costly to manufacture; thus an IC design is carefully verified as correct before it is produced. A mask-level layout is difficult to verify because such a description does not directly correspond to descriptions of the design at other levels of abstraction. A mask layout must therefore be reverse-engineered to map it into a higher-level description, such as a net list, as part of the verification process.

These problems could be reduced if layouts were described at a higher level of abstraction, and if automatic procedures were available to transform that higher-level data into mask-level data effectively. A sufficiently general and flexible symbolic layout and compaction system, as described in the next section, has the potential to provide such a capability.

1.2 Symbolic Layout and Compaction

Symbolic layout was first proposed as a means of speeding up manual layout, by allowing the user to produce layouts via an abstract, shorthand representation for IC mask data [6.25]. The shorthand is "symbolic" in that symbols are used to denote the basic circuit components (e.g., transistors, contacts) and wiring. The symbols are less detailed than their mask equivalents. A symbolic layout is produced by placing symbols on a coarse grid, as opposed to entering detailed mask geometries on a fine grid. The most common example of a symbolic layout is the "stick diagram" style of representation, as shown in Figure 1.1, that evolved from the style used in the STICKS system [83]. Other symbolic-layout styles are presented later.

The coordinates in a symbolic layout that specify the positions and sizes of the elements are "pseudo-coordinates", in that they are dimensionless and they do not express the coordinates of a mask layout. The pseudo-coordinates *do* express the topology of the layout, however. This topology is essentially preserved in the creation of the mask layout from the symbolic.

A symbolic layout must be processed to create a corresponding mask-level layout. In a typical system a symbolic layout undergoes two transformations. The first is called **symbolic translation**: the symbolic elements are replaced with the mask geometries they represent, such that the layout rules pertaining to the construction of the elements are satisfied.

In the second transformation the positions of the layout elements in the plane are

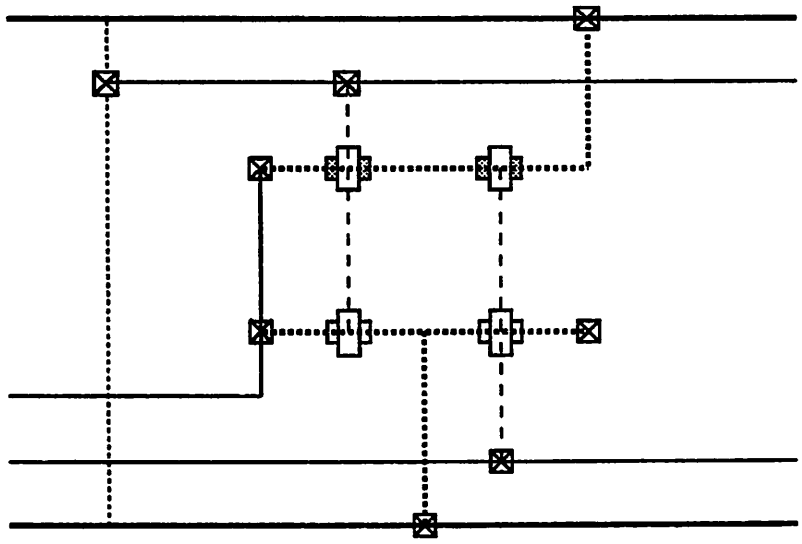


Figure 1.1: Stick-diagram style symbolic layout, as in [31].

modified by translation, such that the layout rules between elements are satisfied and the layout area is minimized. This transformation is called **layout compaction** [4].[1] Layout compaction is an optimization problem, wherein the layout rules are constraints and area minimization is the *primary* objective. Some compactors also optimize a *secondary* objective, which is typically wire-length minimization.[2]

Figure 1.2 shows the circuit from Figure 1.1 after compaction by *SPARCS*, the compactor developed in this project.[3] The various compaction methods that have been proposed are described in Chapter 2.

1.2.1 Advantages of Symbolic Layout with Compaction

Ideally, symbolic layout coupled with compaction has a number of advantages over mask-level layout. A symbolic-layout methodology reduces the amount of data that must be entered to describe a layout. The layout-creation process is thus easier, regardless of whether the data is entered manually or generated by a program. Since the design rules are

[1] Many authors use the term "compaction" to refer to the combination of symbolic translation and layout compaction.

[2] A few symbolic-layout systems do not employ layout compaction: in such systems the coarse grid is defined such that the spacing rules between elements are satisfied for the set of allowed elements. Layouts created in this manner are generally not as area-efficient as those created via other methods, hence nearly all systems provide a compaction capability.

[3] The SPARCS program is described Chapter 8.

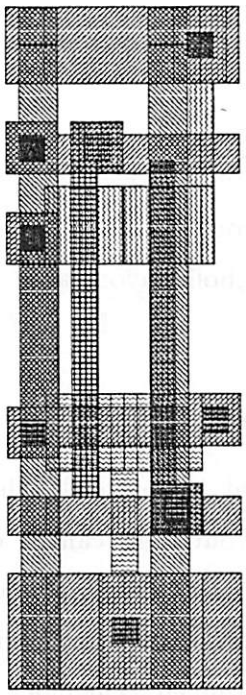


Figure 1.2: Layout of Figure 1.1 after compaction by SPARCS (Chapter 8).

satisfied by the system, the effort in layout generation can be concentrated on developing a good layout topology, rather than on insuring that there are no design-rule violations. Tools that produce layout are easier to construct and maintain because the detailed design rules are factored out. Symbolic layouts can be updated for new design rules more readily than mask-level layouts; for modest changes in the rules, they can simply be re-compacted. Significant effort is required to update a mask-level layout for nearly all design-rule changes. Since symbolic layouts contain more structure than mask layouts, they can be verified more easily. For example, the circuit components and the connectivity must be identified before a layout can be verified against the corresponding schematic diagram. The components, and sometimes the connectivity, are explicit in a symbolic layout; they must be extracted from a mask layout, a process which itself may introduce errors.

However, symbolic layout and compaction systems are only useful if they can be applied to realistic designs, and if the layouts they produce compare well to layouts produced by other methods. To date, these systems have only been applied to a limited class of problems. This is due to both conceptual and algorithmic limitations of the techniques that have been employed. These problems, and several proposed solutions, will be described in

detail in this dissertation.

1.3 Layout Automation

Many techniques for layout automation have been investigated. The purpose of this section is to show where symbolic layout and compaction fits within the spectrum of IC layout problems.

1.3.1 Spectrum of Problems

As a layout is synthesized, it can be classified according to its degree of completeness or refinement. Similarly, layout tools can be compared according to the input data they require and the transformations they perform on that data. Early in the design cycle of an IC, the major functional blocks and their logical interconnections are identified but their locations have not been determined. The blocks are loosely defined in that their aspect ratios and terminal positions are not specified. A **floorplanner** takes a design at this level and produces a more refined version; the block sizes, aspect ratios, and terminal positions are assigned, and the blocks are placed (i.e., they are assigned locations in the plane of the layout). A **placement** program solves a more constrained problem than a floorplanner. The input to a placement program is a set of functional blocks and their interconnections, but the block sizes and their terminal positions *are* specified. Like a floorplanner, the output is a placement of the blocks.

Compared to a placement program's input, a floorplanner's input is farther away from an optimal layout and less detailed. The move set[4] of a floorplanner is thus more general than that of a placer. Another way of viewing this is that a placement program is more constrained than a floorplanner in the transformations that it can perform.

This argument suggests that layout data and thus layout problems can be thought of as a continuum, as shown in Figure 1.3. Moving to the left implies that the layout data is less refined; i.e., the "quality" of the layout decreases to the left and increases to the right. Different tools are appropriate for different areas along the line. A tool moves the design to the right as it transforms its input into its output. The transformations (moves) that are allowed progress from coarse to fine as the design proceeds from the left to the right. Coarse moves are needed early in the design, when the layout is far from an optimum. Fine

[4]The term **move set** refers to the transformations a tool may perform in creating its output.

moves are used later, when the layout is closer to an optimum, to further refine the layout without undoing any of the optimizations already performed.

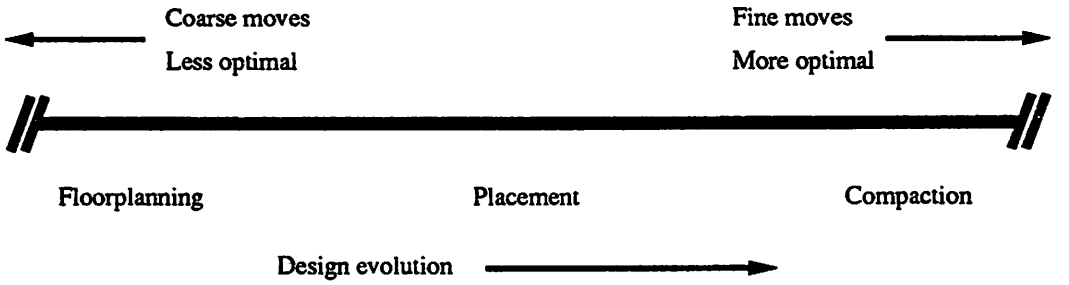


Figure 1.3: Spectrum of layout problems.

1.3.2 Compactor Move Set

Compaction occupies the region to the right of placement on the line in Figure 1.3. The input to a layout compactor is a representation that is complete; i.e., the circuit components are placed and the routing has been implemented. The coordinates that specify the positions of the layout elements express the topology of the layout, but not the coordinates of the final mask layout. The layout topology is one of the most important pieces of information in the input layout. This topology is preserved, in that the ordering of elements that are related by a spacing requirement is not changed. The ordering of unrelated elements is allowed to change, though. Compaction differs from placement in that the topological ordering of related elements does change during placement.

It is presumed that a layout to be compacted has already been optimized topologically, either by a human designer or by other programs. A compactor must perform some additional optimization of the layout without making any gross changes, that is, without changing its topology. As a result, the primary move that a compactor performs is *translation*. A component can be translated by changing either one or both of its coordinates in a single move, corresponding to **one-dimensional** and **two-dimensional** compaction, respectively. Examples of each type of translation are shown in Figure 1.4. Some compactors also perform a second move, which is the insertion of jogs in wire segments. Jog insertion is depicted in Figure 1.5. Wire jogging is fundamentally a two-dimensional move, since it is comprised of choosing a wire to jog, then choosing a location along the wire for the jog point. Any move besides translation and jog insertion (element swapping, rotation,

mirroring, etc.) is not considered to be a legal compaction move.

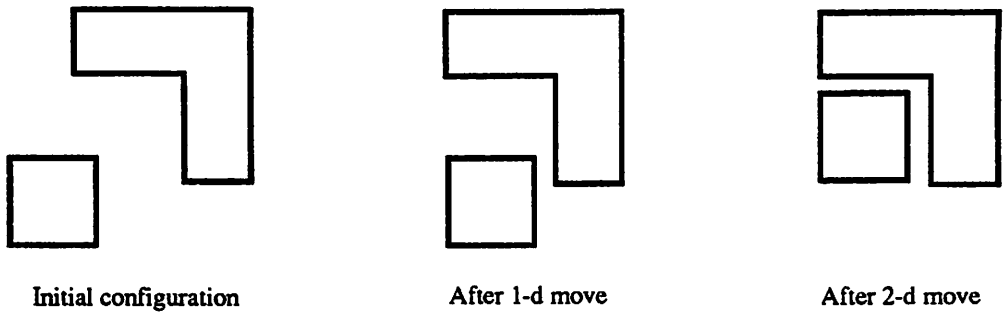


Figure 1.4: One and two-dimensional moves.

Figure 1.5: Jog insertion.

In the following section symbolic-layout methods and compaction algorithms are outlined, and the methods chosen for this work are described.

1.4 Symbolic Layout and Compaction Methods

Symbolic layout systems almost always include a compaction capability. Symbolic systems that employ compaction are called **relative-grid** systems; those that do not are called **fixed-grid** systems. In this section a relative-grid system is assumed. The detailed characteristics of a symbolic-layout description are referred to in this dissertation as the **symbolic-layout model**. That is, the symbolic-layout model is the representation of the geometric and connectivity information used by the compaction program to calculate spacing requirements between pairs of layout elements. Symbolic-layout models are classified in the following subsection.

1.4.1 Symbolic-Layout Models

Relative-grid symbolic-layout models can be distinguished according to whether the layout primitives are **typed** or **generic**, and according to whether the model is **hierarchical** or **single-level**. Models that can be used for only one level of the design hierarchy are single-level models. Most single-level models are designed for the lowest, or **leaf** level of the hierarchy, where the layout elements are transistors, contacts, etc. A model is hierarchical if compacted cells from a given level can be used as primitive elements in higher-level cells, which are themselves subsequently compacted.

A typed layout model is one wherein the system encodes specific information about its primitives in terms of a type designation. This information includes the shape of the element, and data used for computing spacing requirements to other elements. For example, a diffusion-metal contact, a rectangular NMOS transistor, and a non-rectangular NMOS transistor would all have different types.

The use of a typed layout model leads to area-efficient results for layouts comprised of the encoded primitives, since the spacing computation can be special-cased to exploit their type-specific characteristics. However, the use of types inherently limits the system to those primitives that are pre-encoded: the system must be modified if a new element is added. It becomes necessary to add new types when a system designed for one technology, such as NMOS, is to be used for another, such as bipolar. In addition, hierarchical compaction is cumbersome when a typed model is used. Hierarchical design implies that compacted cells are to be used as primitive elements on higher levels. Each distinct compacted cell corresponds to new a element of unknown type. Since the characteristics of a compacted cell cannot be pre-encoded, most hierarchical systems use a completely different layout model for levels above the leaf level. The most common higher-level model is limited to rectangular elements (i.e., the previously-compacted subcells) which are separated from one another by the largest spacing rule; hence the results are often poor in terms of area efficiency.

In a generic model all elements are treated as arbitrary cells. No distinction is made between primitives that represent contacts, MOSFETs, standard cells, etc. A generic model consists of geometric and connectivity data only; it is thus a lower-level model than a typed model. As a result, more effort is required to compute spacing requirements since a case-analysis approach does not apply. However, a generic model can lead to a system that is truly technology independent and hierarchy-level independent, unlike systems that use

typed models. Since all elements are modeled in the same manner, adding a new element or using the system for a different technology can be done without modifying the programs. Elements need not be rectangular, and it is not necessary to separate cells by the largest spacing rule; hierarchical operation is thus more natural and more area-efficient results are produced compared to other methods.

For these reasons, the generic modeling approach has been chosen for this research.

1.4.2 One-Dimensional versus Two-Dimensional Compaction

As noted above, the move performed by all compactors is translation in the plane. A component translation can be performed by changing either one of its coordinates (one-dimensional compaction) or both of its coordinates (two-dimensional compaction) in a single move. A true two-dimensional compactor[5] can minimize the layout area directly. One-dimensional compactors minimize area indirectly, by alternating horizontal and vertical compaction iterations. That is, they minimize area by minimizing the pitch of the layout in each dimension individually.

True two-dimensional algorithms will, in general, produce better results than one-dimensional algorithms. However, there are several compelling reasons to choose a one-dimensional compaction method. The general two-dimensional problem is NP-complete[6] [66], and hence it is not a useful formulation for a practical system. One-dimensional compaction has polynomial complexity, and the most general one-dimensional algorithms have complexities that approach $O(n \log n)$ for real examples. Heuristic two-dimensional algorithms of polynomial complexity have been proposed [86,70], but they are slower than well-implemented one-dimensional algorithms. In addition, it has not been proven that they are superior in terms of layout area to one-dimensional compaction with jog insertion [9].

These considerations, and others that will be given later, have led to the selection of one-dimensional compaction for this research.

---

[5] A "true" 2-d algorithm considers both dimensions at the same time with equal weight. Less-general 2-d algorithms change one of the dimensions preferentially.

[6] The class of NP-complete problems is that class for which there are no known polynomial-time algorithms.

1.4.3 One-Dimensional Compaction Methods

One-dimensional compaction methods are reviewed in detail in Chapter 2. The three algorithms that are the most common are the compression-ridge method [4], the virtual-grid method [81], and the constraint-graph method [31]. Constraint-graph (or "constraint-based") compaction is the most general method,[7] and it is the only method that uses a global model of the layout. The constraint-graph method has been employed in this research.

Overview of the Constraint-Graph Method

Like all one-dimensional methods, the constraint-graph method reduces the area of the layout by minimizing its pitch in a sequence of spacing steps in alternating directions. The layout is modeled as a weighted, directed, constraint graph. The constraint graph provides a global view of the layout in the spacing direction. Distinct graphs are used for the x and y-directions. Layout elements map to nodes in the graph, and spacing requirements (constraints) map to weighted edges. The edge directions represent the ordering of the layout elements. The mapping of elements to nodes can be done in many ways. For example, in SPARCS, a node is used for each element and for each wire segment that is perpendicular to the spacing direction. Segments that are parallel to the spacing direction are not represented in the graph. The constraint graph is created by a **constraint-generation** algorithm, which takes a symbolic layout as input and produces the graph as output.

The graph is analyzed to determine the new positions for the elements. A longest-path analysis on the graph gives the minimum legal locations for the elements.[8] Typically only a subset of the elements must be placed at their minimum locations to achieve the minimal layout; the elements in this subset are called **critical** elements. The other elements are called **noncritical** or **slack** elements, since they can occupy a range of positions without increasing the size of the layout. The critical-path method, which is comprised of two longest-path analyses, partitions the elements into these two subsets and determines the range for each noncritical element. The locations of the noncritical elements can be manipulated to satisfy a secondary objective, such as wire-length minimization.

From this overview, it is evident that the capabilities of a compactor are de-

[7] The other two methods can be modeled using the constraint-graph approach [86].

[8] For example, if the layout is being compacted to the left, the longest-path analysis generates the smallest legal x-coordinate value for each element.

termined by two factors, one pertaining to the input layouts and one pertaining to the constraints. The layout model, plus the constraint-generation algorithm, determine the spectrum of designs that can be processed by the system. The allowed constraint types, plus the constraint-analysis algorithms, determine the optimizations performed.

1.5 Goals and Contributions

Previous symbolic layout and compaction systems have been limited in terms of the IC technologies and layout styles they support. The basic goal of this work has been to develop general symbolic layout and compaction techniques that apply to a broader range of technologies and layout styles. The three main contributions that have been made towards this goal are described in the subsections below.

1.5.1 A Generic Symbolic-Layout Model

The symbolic-layout model and the layout rules together determine the set of designs that can be described to and processed by a system. Many symbolic layout models allow only simple rectangular elements with one terminal (connection point) per side. Such a model cannot capture, e.g., a standard-cell design, since standard cells have several terminals per side. Non-rectangular elements cannot be effectively accommodated by systems that use such a model. Some systems handle simple layout rules only, leading to losses in area efficiency or incorrect results. The standard symbolic-layout model assigns types to its primitive elements, which are used to calculate the element-to-element spacing constraints. A typed methodology inherently limits the system, as described above.

A new symbolic-layout model has been developed in this research. The model is generic, meaning that the symbolic elements are not typed. All elements are modeled in a single, consistent manner, regardless of whether they are compacted cells or leaf-level primitives. The model is comprised of blockages and connection areas. Any Manhattan shape is a legal blockage, and any number of rectangular terminals is allowed. It is demonstrated that this model is technology and hierarchy-level independent. The presented results also show that this model leads to layouts that are at least as area-efficient as those produced via other models, unlike previous work in generic-layout modeling.

1.5.2 Efficient Constraint Generation

The layout model and the constraint-generation algorithm are closely-related entities. Given a layout model, the constraint-generation algorithm must be able to efficiently process the layout and it must generate the set of constraints that leads to a minimal-area layout.

A new constraint-generation algorithm has been developed in parallel with the layout model. This algorithm generates constraint systems for leaf-level layouts that result in layout densities as good as those produced by methods that rely on typed elements. The same layout model and constraint generator can also be used at higher levels in the hierarchy without modification. The constraints are computed such that layout elements in higher-level designs are not separated by the largest spacing rule. Instead, elements at all levels of the hierarchy are separated by the per-layer rules, and elements at all level are allowed to merge (overlap) at their terminals. This produces much better area results in hierarchical compaction than the standard method. In addition, the proposed constraint generator is competitive in terms of execution time with other algorithms that are restricted to typed layout models.

1.5.3 Dependent Constraints

All constraint-based compactors employ lower-bound constraints to limit the minimum separations between layout elements. Many also allow upper-bound constraints, which limit maximum separations. Inclusion of both lower and upper-bound constraints results in a great deal of flexibility. However, there are situations where these two constraint types are insufficient. In particular, *dependencies* among constraints are needed for some designs.

A new constraint type, called an **active constraint**, has been defined and algorithms for resolving active constraints have been developed. Active constraints are a form of dependent constraint. That is, the spacing between a pair of elements in the layout is dependent on the spacing between some other pair. Active constraints can be used in a variety of situations where electrically-matched structures are needed. For example, symmetric layouts can be compacted such that they remain symmetric. Symmetry is very difficult to maintain without active constraints.

In a hierarchical cell, a given subcell is typically repeated several times. Consider a subcell A that is repeated more than once in the cell currently being compacted. The

standard method of hierarchical compaction requires that adjacent cells be stretched such that they pitch-match. Since the different instances of A typically have different cells as neighbors, the stretching step changes the various copies of A differently. If this occurs, the hierarchy is effectively lost, because the multiple instances of A are no longer identical. Active constraints can be used in the stretching step to force multiple instances of a given cell to remain the same. Other uses of active constraints are given later.

1.6 Outline of Dissertation

The remainder of this dissertation is organized as follows. A review of the previous work in symbolic layout and compaction is presented in Chapter 2, followed by a justification of the methods chosen for this work. A description of the factors that affect the design of a symbolic-layout model, and a description of the high-level characteristics of the model developed in this project, are given in Chapter 3. The subject of Chapter 4 is the cell model proposed in this research. In Chapter 5 the constraint generator designed for this model is described. The basic constraint solver is presented in Chapter 6. Active constraints, and methods for solving them, are the subject of Chapter 7. The SPARCS program, which is the compactor developed to test the methods presented in this dissertation, is outlined in Chapter 8. Overall results are presented in Chapter 9, followed by conclusions and future work in Chapter 10.

Chapter 2

Previous Work and Choice of Technique

2.1 Introduction

The primary purpose of this chapter is to review the previous work in symbolic layout and layout compaction. The review is then used as a point of reference for this work.

Symbolic layout and layout compaction are distinct but closely-related topics. Many different styles of symbolic layout have been proposed. Most of them are intended for use with layout compaction, although symbolic layout has been used in a stand-alone manner without a compaction capability. The major symbolic-layout styles are described in this chapter.

A number of compaction algorithms with a variety of capabilities have been reported as well. Nearly all take a symbolic layout as input and produce a mask-level (i.e., physical) layout as output. A few compactors have been reported which take a physical layout as input instead. These tools must convert the input layout's geometries into a form where components (transistors, contacts, etc.) are the basic objects before any compaction is performed; that is, they perform a physical-to-symbolic conversion as a pre-processing step. It is thus reasonable to think of all compactors as taking a symbolic layout as input. Previous work in compaction is described in this chapter, in terms of the algorithms used and in terms of the classes of layouts processed.

This chapter is organized as follows. The next section describes some basic con-

cepts and terms used in IC layout. Symbolic-layout methods are then classified and reviewed, followed by a classification and review of layout compaction. The goals of this project and the requirements implied by the goals are presented next. The chapter concludes with a justification of the methods chosen for this project.

2.2 IC Layout Terminology

Integrated circuits are planar structures composed of several layers of material, each differing in its electrical properties. They are fabricated by a sequence of processing steps; in a particular step specific areas of the semiconductor wafer are modified. The areas are altered by depositing a material, such as aluminum or polysilicon, or by exposing them to dopants to alter the semiconductor itself. The areas are defined using photolithographic techniques. Overlaying several areas results in structures which form the basic circuit components: transistors, contacts, capacitors, etc.. as shown in Figure 2.1.

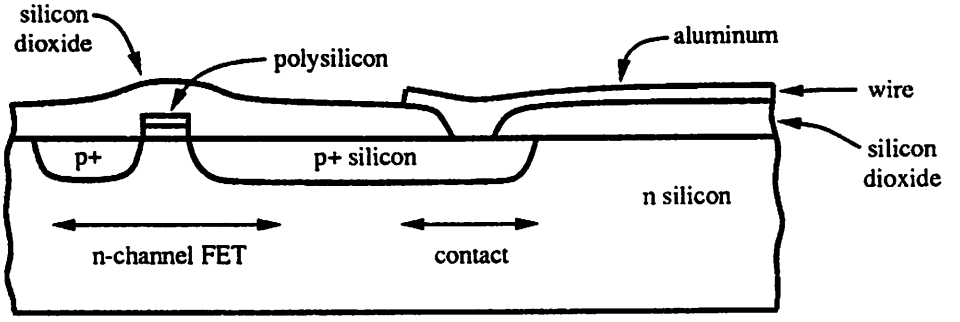


Figure 2.1: Cross-section of a circuit with a transistor, a contact, and a wire.

An IC **layout** is the geometric data that defines the areas to be processed. The term **layer** refers to a given material that is created during a given step in the process sequence. The layout data is broken down into per-layer sets of geometries. The set of geometries that is associated with a particular layer is called a **mask**, **layer**, or **level**. The mask-level layout of the components in Figure 2.1 is given in Figure 2.2.

A circuit consists of basic circuit components connected together by wiring. A component is comprised of one or more shapes on one or more layers that are size-invariant and that are manipulated as a unit. In an MOS technology, for example, a rectangular transistor is a component that can be specified by two overlapping rectangles, one on the diffusion layer and the other on the polysilicon layer. Many authors refer to transistors as

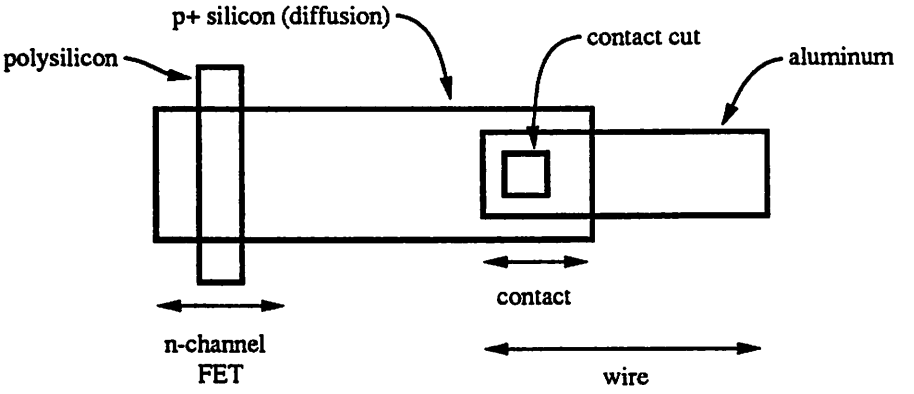


Figure 2.2: Mask-level layout of the circuit in Figure 2.1.

devices. A wire is a shape on a single layer that has a fixed width but a flexible length. A wire can be decomposed into a set of **segments**, each defined by two points and a width, as illustrated in Figure 2.3. A collection of components and/or wire segments that is grouped together and then treated as a unit is called a **cell.** Cells are often allowed to contain other cells; this permits a cell **hierarchy** to exist. The term **element** will be used from this point on to refer to a basic circuit component, a wire segment, or a cell. The terms **component** and **primitive** will henceforth be used to denote any element except for a wire segment. Figure 2.2 is redrawn as Figure 2.4 to indicate the components and wires. In this dissertation wires will often be drawn with centerlines included to indicate their directions. A component is **basic** if it cannot be expressed as a collection of lower-level elements. A transistor is a basic component. A standard-cell is basic, if it is at the lowest level of hierarchy, and it is not basic if its description is comprised of transistors, contacts, and wires.

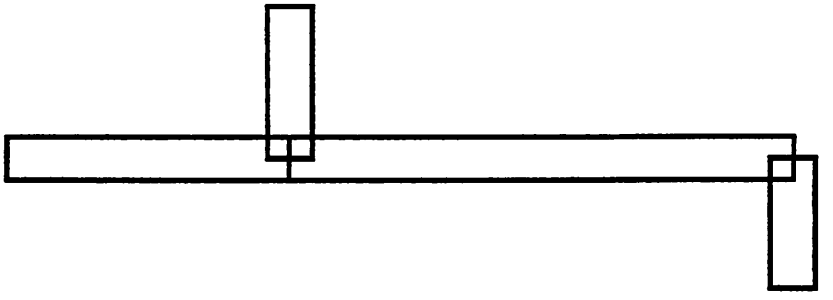


Figure 2.3: Wire segments.

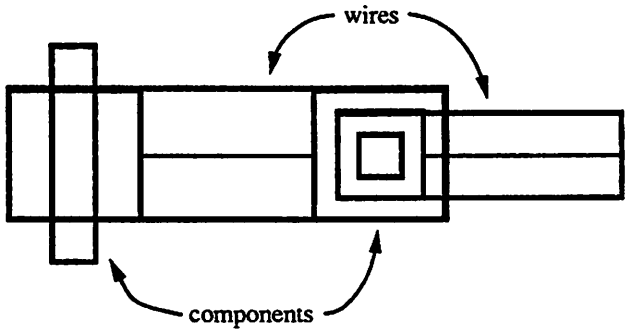


Figure 2.4: Layout of Figure 2.2 showing components and wires.

2.2.1 Layout Rules

To insure that a functional IC can be manufactured, a layout must satisfy a large set of rules that specify relationships between the mask geometries. These rules are called **layout rules** or **design rules**. Design rules for simple bipolar and NMOS processes are given in [28] and [56], respectively.

Design rules fall into two categories: those that describe the construction of components are the **intra-element** rules, and those that describe the spacings between components are the **inter-element** rules. The intra-element rules include minimum size, maximum size, minimum overlap, and minimum enclosure requirements. Figure 2.5 shows typical intra-element rules for a polysilicon-to-metal contact. Most inter-element rules are minimum-spacing rules, as exemplified in Figure 2.6.

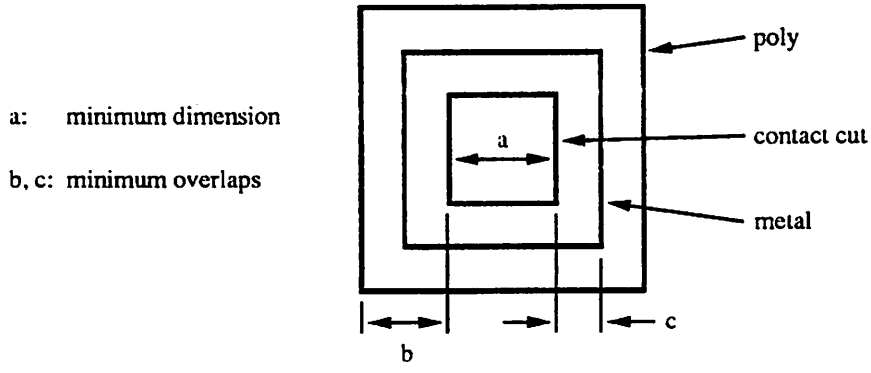


Figure 2.5: Intra-element rules for a poly-metal contact.

Design rules for industrial processes are complicated and often ill-specified. They

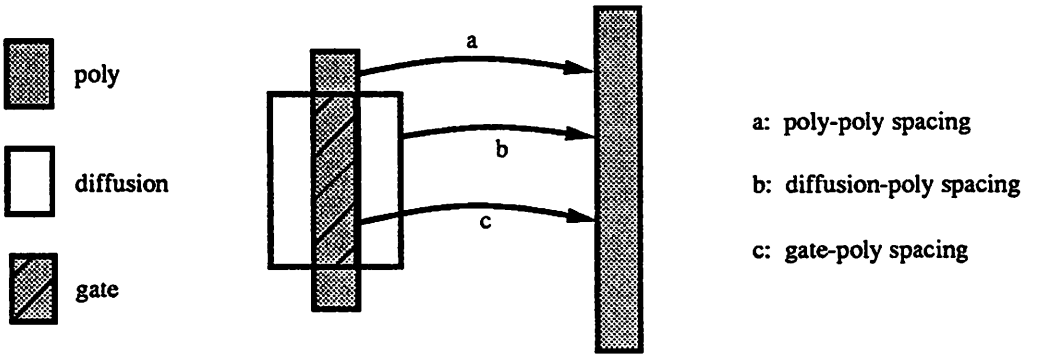


Figure 2.6: Inter-element rules between a transistor and a poly wire.

may exhibit dependencies on connectivity, context, width, and length. The importance of always satisfying all the rules is unclear; sometimes certain rules can be safely violated. There does not seem to be a canonical set of rules that can be used for all processes, and rules tend to appear and disappear over time. For example, the Mosis CMOS rules had a reflection rule in Revision 4 but not in Revision 5 [2], which seems to indicate that reflection rules are disappearing. However, sources such as [76] indicate that reflection rules will become more prevalent in the future. It is generally unclear how design rules will continue to change over time.

Dealing with design rules is one of the most difficult aspects of layout automation. As pointed out in [59], simple CAD algorithms that allow only simple rules may execute efficiently, but may produce poor quality layouts. Algorithms that handle more complex rules may produce better results, but at a higher cost in development time and execution time.

2.2.2 Fabrication Technologies

The term **technology** refers to a particular fabrication-process sequence. Bipolar and MOS are the most common technologies currently available. The two prevalent MOS-process variants are CMOS and NMOS. These two processes are very similar in terms of the layout primitives they provide, and hence they are not considered to be different technologies in this dissertation. A combination of the CMOS and bipolar technologies, called BiCMOS, is becoming popular as well.

Technology Independent Versus Technology Parameterized

The phrase "technology independent" is used with many different meanings in the description of CAD tools. A distinction is made in this dissertation between technology independence and technology parameterization. A CAD tool that works for a single IC process is **technology dependent**. A CAD tool that works for all technologies is termed **technology independent**. Most layout tools that operate on designs from a single process are parameterized with respect to the sizes of the basic primitives and the design-rule values. A tool that supports a single parameterized process, but no others, will be called a **technology-parameterized** tool. Other phrases commonly used for this notion are process independent, design-rule independent, and ground-rule independent.

Most symbolic layout and compaction systems are technology parameterized. An important focus of this research has been in developing symbolic layout and compaction techniques that are technology independent.

2.2.3 Hierarchy in Layout

Hierarchical methods are important in IC design, due to the complexity of the VLSI circuits currently being built. A design that is not hierarchical is called **flat**. There are two main reasons to use hierarchy in layout, one having to do with data storage and the other with computation. If a given cell is repeated, it can be described in detail once and then referenced multiple times, saving storage over a flat representation. It is sufficient to perform some analyses, e.g., design-rule checking, of a repeated cell once, saving computation time. Hierarchical methods can also be beneficial in a divide-and-conquer sense, whether cells are repeated or not.

Allowing a cell to be composed of subcells is the basic mechanism needed to enable hierarchical layout. In a cell at the lowest level of the hierarchy, which is called the **leaf level**, the elements that comprise the cell do not contain subcells. The elements in a leaf-level cell are called **leaf-level elements**. Transistors, resistors, contacts, etc. are leaf-level elements.[1] Cells that are not leaf-level cells are **higher-level** cells. Some hierarchical systems treat the leaf-level elements in the same manner as higher-level subcells, while others special-case the leaf-level elements.

The description of a cell in terms of its subcells (and/or its leaf-level elements) is

[1] They are also basic elements.

called its **definition**. To use hierarchy effectively, there must be a way to model a cell from the current level of the hierarchy in a simplified manner on the next level. A description of a cell that is less detailed than its definition, but that is sufficient to represent the cell on higher levels in the hierarchy, is called its **abstraction**. For example, a standard-cell can be defined in terms of transistors, contacts, and wires, and it can be abstracted as a set blockages and connection points.

Hierarchy-Level Independence

Layout tools can be classified according to the hierarchy level(s) of the data they process. Parasitic extractors, for example, usually operate on leaf-level data, while timing analyzers have been developed that operate on data from any level. It is advantageous to develop tools for hierarchical layout systems that can process any level of the design hierarchy in a consistent, uniform manner, as opposed to tools that use different modes of operation for different levels, or only operate on a subset of the levels.

A tool that is appropriate for any level of the hierarchy is called **hierarchy-level independent**. Nearly all symbolic layout and compaction systems are hierarchy-level *dependent*. A major focus of this work has been on the development of a symbolic-layout model and a constraint-generation algorithm that achieve hierarchy-level independence, without a sacrifice in efficient.

2.3 Symbolic Layout Classification

As noted in Chapter 1, symbolic layout originated as a means for reducing the effort needed to create IC layouts over that required to produce physical-level layouts. This reduction is achieved through the use of a shorthand representation for IC mask data that consists primarily of symbols rather than rectangles and polygons. The symbols usually represent the leaf-level components available in the fabrication technology (e.g., NMOS transistors, PMOS transistors, and contacts, in CMOS), plus wires. The component symbols vary in their level of abstraction: for example, some systems use symbols that resemble schematic symbols while others use symbols that resemble mask-level components. In any case, the symbols are less detailed than their mask equivalents. The components are interconnected by symbolic wires, which are typically represented as zero-width lines. The symbolic representation may, or may not, contain the connectivity of the layout. The coordinates in a

symbolic layout that specify the positions and sizes of the elements are pseudo-coordinates that express the topology of the layout only.[2] The initial topology of a symbolic layout is important since it is assumed that significant optimizations have already been performed, and thus compaction *does not* change this topology appreciably with respect to elements that are related by design rules.[3]

There are several criteria by which symbolic-layout systems can be categorized. One important criterion is whether or not the symbolic-layout method is used with a compaction program. The term **fixed-grid** [31] has been used to denote symbolic layout methods that *do not* employ compaction. The terms **relative-grid** [31] and **relative-location** [23] have been used to describe symbolic layout methods that *must* be used with compaction to generate legal layouts. These terms reflect the fact that the grid is used only to indicate the relative ordering of the layout elements. The gridline-spacing value of a relative grid does not correspond directly to the dimensions of mask-level features. The relative coordinate system is mapped to the coordinate system of the mask level by the compactor.

All layout systems employ a grid as a drawing aid and as a means of specifying the smallest geometric feature that can be entered. The spacing between gridlines is one characteristic that can be used to differentiate fixed-grid symbolic systems. The gridline spacing, or resolution, is said to be **coarse** if it is of the same order as the maximum spacing rule; it is **fine** if the spacing is significantly less than the maximum rule. So-called **gridless** systems are fine-grid systems with the gridline spacing equal to the minimum resolvable feature for the fabrication process.

Symbolic-layout systems can be classified according to a second criterion, namely the level(s) of hierarchy they can accommodate. Systems that can be used for only one level are called **single-level** systems in this dissertation. The standard term for single-level systems that operate on the lowest level of the hierarchy, where the layout elements are transistors, contacts, etc., is **leaf-level**. Single-level systems exist for levels above the leaf-level, where the elements are rectangular cells; these systems have been called **block-level** or **building-block** systems. A system that handles multiple levels is termed **hierarchical**.

Finally, symbolic layout systems can be categorized as **typed** or **generic**. A typed system performs case-by-case processing of the elements according to their types (e.g., driver

[2] Here it is assumed that the system includes compaction.

[3] The topological ordering of elements that are not related by design rules is usually allowed to change during compaction.

transistor, contact, load transistor, etc.). A generic system models all elements as arbitrary cells. A detailed comparison of typed versus generic symbolic-layout methods is given in Chapter 3.

2.4 Symbolic-Layout Review

Fixed-grid systems are reviewed in the following subsection. A relative-grid form of symbolic layout has been used in this work. Previous work in relative-grid systems is described later in this chapter.

2.4.1 Fixed-Grid Symbolic Layout

In fixed-grid systems layouts are created by placing symbols on a grid whose gridline-spacing value maps directly to a predetermined distance at the mask level. After the layout is composed a symbolic-translation step is performed, which replaces the symbols with their mask equivalents.

Larsen, in 1971, described a character-based symbolic form that is the output of a system for generating preliminary topological designs of random logic [45]. Larsen calls his character-array layouts "discrete topological schematics". In 1973 an interactive editor for them was presented [46]. Methods for automatically mapping a discrete topological schematic into a mask-level representation were described by Larsen in 1978 [47]. A simple example is given in Figure 2.7. This style uses a coarse grid with the gridline spacing set to the maximum spacing rule; hence no design-rule checking is needed.

Barnes and Davis described a fixed, coarse-grid system in 1975 [6] that also uses a worst-case grid spacing. The system they developed, called the Graphic Layout System, performs some extraction and electrical-rules checking (ERC) functions in addition to the symbolic entry, editing, and pattern-generation functions. The Graphic Layout System uses a stick-diagram style for the layouts themselves, instead of an alphanumeric-character style. The system accommodates CMOS, CCD, and IIL designs. According to Barnes and Davis, the notion of using a symbolic methodology as an intermediate form between the circuit-design activity and detailed layout was first proposed by J.O. Campeau in 1967 [6].

The SLIC system, by Gibson and Nance, [25,26] is a character-based coarse-grid system. This tool uses a gridline spacing that is smaller than the maximum spacing rule

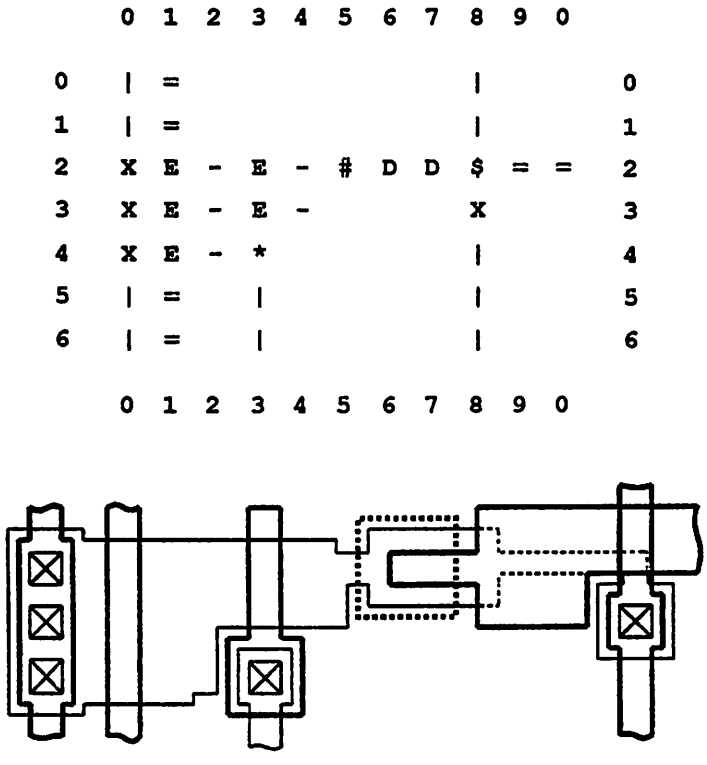


Figure 2.7: Symbolic and mask representations, as in [47].

but larger than the minimum feature or minimum spacing rule. As a result a symbol-to-symbol design-rule checker is included. The SLIC system also includes programs for net-list extraction, ERC, and net-list comparison. Gibson and Nance noted that their methodology is difficult to apply to complicated processes like CMOS while maintaining adequate area-efficiency [26].

Advantages and Disadvantages

The advantage of fixed-grid systems is their simplicity compared to relative-grid systems with compaction. However, their utility is limited by the very nature of the fixed grid. Since the gridline spacing cannot be modified, a fixed-grid method is most appropriate for technologies where the *range* of design-rule values is *narrow*, such as NMOS. Technologies like CMOS, where the range in design-rule values is broad, cannot be captured effectively via a fixed-grid style. A coarse gridline spacing wastes area. A fine spacing leads to better area utilization, but many symbols must be entered for the larger features that span several gridlines and many gridlines must be skipped between features related by the larger spacing rules.[4] The value of the gridline spacing is thus a critical parameter, because it represents a compromise between area efficiency and productivity. A worst-case spacing simplifies design entry and checking at the cost of wasted area. A finer spacing leads to better area utilization, but more data must be entered and design-rule checking must be added. As the spacing becomes finer the entry and checking effort approaches that needed for mask-level layout.

The fixed-grid systems outlined above are leaf-level, typed systems. Most fixed-grid systems are character based, because they were developed before computer graphics systems became inexpensive.

2.4.2 Relative-Grid Symbolic Layout

In relative-grid systems, the layout grid is used only to indicate the ordering of the elements in the plane. The element locations, i.e., the gridline locations, are changed in a compaction step to correspond to the mask-level coordinate system. All compactors use a relative-grid form of symbolic layout, and all relative-grid layouts must be compacted to guarantee that the design rules are satisfied.

[4] This is essentially the point made in [26] with respect to CMOS.

The Point-Component Model

Relative-grid systems generally model leaf-level layouts via a specific model called the **point-component** model [31]. The primitives used in this model are rectangular, with one connection per side, and they are symmetric about their centers in one or both dimensions. The elements given in Figure 2.8 are point components.

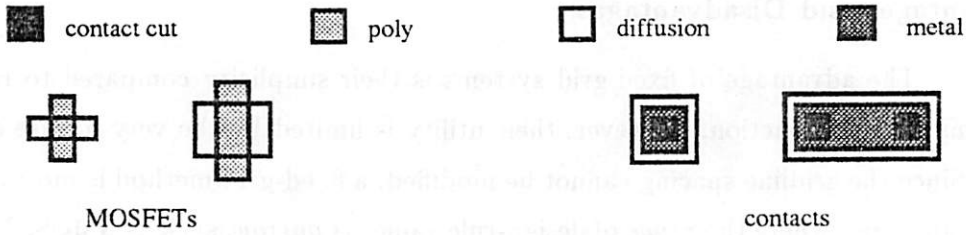


Figure 2.8: Point components.

A point component occupies a single gridpoint on the relative grid. Since an element may have only one connection per side, all wiring lies on the gridlines between components and electrical connections are established by intersecting the endpoint of a wire segment with the center point of a component. The layout in Figure 2.9 exemplifies this model.

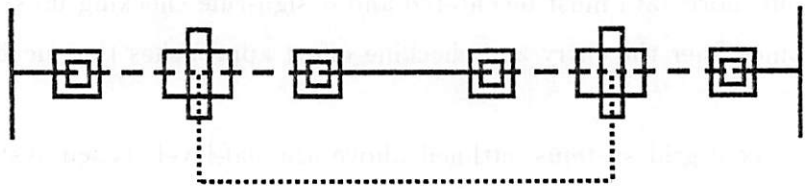


Figure 2.9: Point-component layout.

In hierarchical compaction spaced cells from the current level in the hierarchy are used as primitives on higher levels. Higher-level cells *cannot be modeled* via the point-component model, because they nearly always have multiple connections per side. As a result, systems that use the point-component model at the leaf level must employ a substantially different model for higher levels. The modeling techniques used in this case are described later in Section 2.6.4.

The point-component model has some advantages, but it also has several serious disadvantages. The advantages and disadvantages of this model are described in detail in

the following chapter.

2.5 Layout Compactor Classification

Compactors operate by analyzing the layout to determine spacing requirements, then moving components to reduce the size of the layout such that the spacing requirements are satisfied. Hence they can be classified according to the *input layouts* they accept, and the *spacing algorithms* they use.

Compactors are usually classified according to the dimensionality of the spacing algorithm, where dimensionality refers to the way that component translation is performed. Algorithms that change either the x-coordinate or the y-coordinate in a single move, but not both, are **one-dimensional** methods; those that change both coordinates in a single move, with each dimension considered equally, are **two-dimensional** methods. Several heuristic algorithms have been reported with dimensionalities between one and two; these approaches will be called **intermediate** methods.

The specific characteristics of the input layouts are an important classification criterion for two reasons. First, the symbolic-layout form determines the class of layouts that can be described, and thus processed by the system. Second, the symbolic layout must be analyzed to determine the spacing requirements between element pairs. A simple symbolic-layout form may be easy to analyze, but the class of layouts that can be described may be limited and the area-efficiency of the results may be poor. A more general form may describe a wider variety of layouts and may lead to better area utilization, but at the cost of increased computation time in calculating the element-to-element spacings.

The classification criteria having to do with the input layouts mirror those presented in Section 2.3. A compactor that can handle a single hierarchy level, regardless of what that level is, is called a **single-level** or **flat** compactor. A single-level compactor can be further classified according to the hierarchy level it processes. Most single-level compactors process layouts from the lowest level of the hierarchy, and thus are called **leaf-level** or **leaf-cell** compactors. Those that target a higher level, where the primitive elements are multi-terminal cells, are called **block-level** or **building-block** compactors. The leaf-level and block-level problems are usually treated quite differently from the perspective of computing spacing requirements, as will be explained later in this dissertation.

A tool that can compact layouts from any level of the hierarchy is called a **hier-**

archical compactor. It should be noted, however, that the term "hierarchical compaction" most commonly refers to a very specific design style, wherein all hierarchy levels above the leaf level are composed of pitch-matched subcells that connect by abutment. This stretch-and-abut style of cell assembly is described later in this chapter.

2.6 Layout Compaction Review

The following review of compaction is primarily organized by dimensionality. The types of layouts processed by each tool are mentioned as well.

2.6.1 One-Dimensional Compaction

One-dimensional compactors reduce the layout area via a sequence of horizontal and vertical compaction iterations. In a horizontal iteration the components and the vertical wire segments translate, and the horizontal wire segments change length. A vertical iteration is similar. Many one-dimensional compactors have been reported; the various algorithms they use are described below.

Compression-Ridge Compactors

The algorithm proposed by Akers, et al., in 1970, appears to be the first reference to compaction [4]. Although the method was not named in the original reference, other authors have referred to it as the compression-ridge method and as the shear-line method.

In this algorithm, excess space is removed from the layout by searching for a band of excess space, called a **compression ridge**, that spans the layout from left to right or top to bottom. If a continuous band cannot be found, then several partial compression ridges connected by orthogonal **shear lines** are used. Akers et al. called compression ridges "cuts" and shear lines "rift lines". Compression ridges have the property that, when they are removed, the two resulting parts of the layout can be pushed together and all connections are restored. Figure 2.10 shows several partial compression ridges and several shear lines. Figure 2.11 shows a sequence of compression-ridge compaction steps applied to a simple example. The algorithm used to find a set of partial compression ridges and shear lines is similar to a maze-routing algorithm, where the starting point is an empty grid on one edge of the layout and the target is any empty grid on the opposite edge.

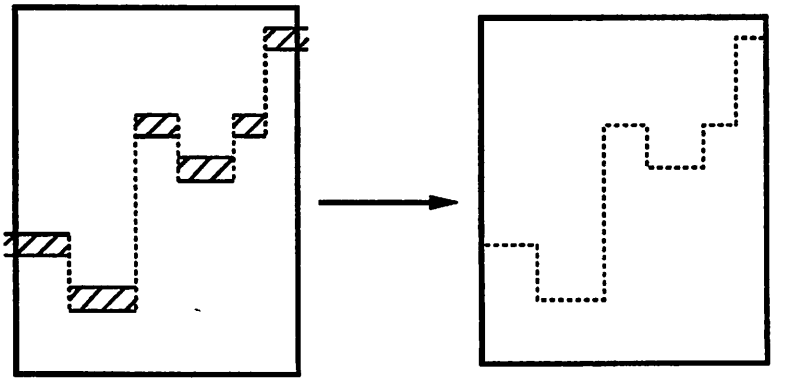


Figure 2.10: Partial compression ridges and shear lines, as in [4].

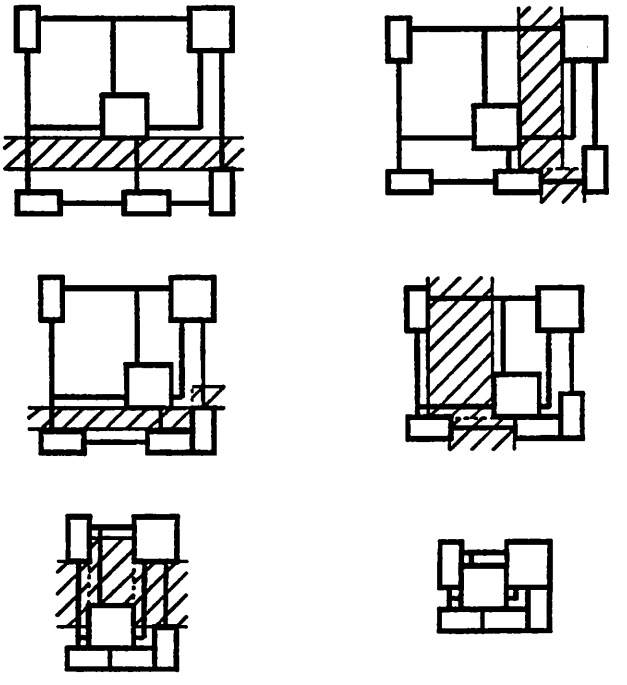


Figure 2.11: Compression-ridge example, as in [4].

Akers' tool is for leaf-level bipolar layouts with a single metal layer. A coarse, uniformly-spaced grid is used, with the gridline spacing chosen to accommodate the metal layer. Components are rectangular, with their sizes rounded up to fit the grid. Compression ridges are one grid wide in this scheme, and the input layouts must be design-rule correct.

According to Cho [17], the first actual implementation of the compression-ridge algorithm was by Dunlop in the SLIP program [23]. Compression-ridge compaction was also used as the global compaction phase in SLIM [22], which is described below. For efficiency reasons SLIP was targeted for small, leaf-level NMOS layouts. In SLIP a finer grid is used, hence component dimensions are not rounded up to fit a worst-case grid. The width of a compression ridge must therefore be calculated along with its path. The compression-ridge method does not increase element-to-element spacings that are too small. As a result, the sticks-style symbolic layouts processed by SLIP are converted into legal mask-level layouts before the compression-ridge algorithm is applied.

Other than these early cases, the compression-ridge algorithm has not been applied to leaf-level layouts. More recent applications of compression-ridge compaction have been to block-level layouts [35,33,42]

The compression-ridge method gave rise to the field of layout compaction. However, the algorithm takes a local, rather than a global view of the layout. It does not appear that the compression-ridge method can be naturally extended to handle user-defined constraints and secondary objectives, such as wire-length minimization. In addition, it is not efficient enough to handle large leaf-level problems, as noted in [23].

The STICKS System

The publication of Williams' STICKS system in 1977-78 [84,83] firmly established the relationship between symbolic layout and layout compaction. The term **stick diagram** originated with this work. A stick diagram, as shown in Figure 2.12, is a particular symbolic-layout style where the components are symmetric with a single connection per side and wires are represented by centerlines. Many variations on this style have been used. Unlike many other stick-style systems, STICKS could accommodate Manhattan "black box" cells with multiple connections per side, and a mix of mask rectangles with symbolic elements. The STICKS system was thus the first hierarchical symbolic layout and compaction system.

Compaction is performed first in this system, followed by symbolic translation. A

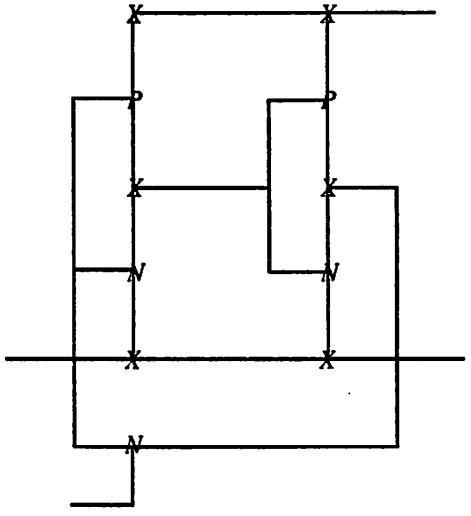


Figure 2.12: A stick diagram as in the STICKS system [83]

rule-based approach is used in the compaction step. For the horizontal step, collections of elements ("structures") are considered in increasing-x order; each collection is placed as far to the left as possible. The vertical direction is processed in a like manner.

Williams took an ad hoc rather than an algorithmic approach to the layout-compaction problem. However, he addressed several features that have come to be regarded as important for a flexible, useful system. These include "corrals", which are flexible boundaries that can be used to group elements together, "fractures", which are wire jogs, and "absolute micron distance" specifications between symbols, which are fixed constraints.

Constraint-Based Compactors

It appears that FLOSS, described in 1977 [16], was the first constraint-based compactor, although this was not disclosed in [16]. In fact, at least one other constraint-based compactor was reported [32] before the algorithms used in FLOSS were described in an oral presentation in 1979 [17]. The FLOSS algorithms have not been disclosed in writing. A large number of constraint-based compaction algorithms have subsequently been presented [5,39,20,62,24,14,53,18].

Constraint-based compactors map the layout into a weighted, directed graph. Two graphs are actually used, one for the x-direction and one for the y-direction. Layout elements map to nodes in the graph, and spacing requirements, i.e., constraints, map to weighted

edges. The edge directions specify the relative ordering of the corresponding nodes, while the weights model the separation requirements. A simple layout and one possible constraint graph for a horizontal compaction are presented in Figures 2.13 and 2.14, respectively. In Figure 2.14 the source node s represents the left edge of the bounding box of the layout and the sink node t represents the right edge. Design-rule separations between pairs of elements are represented by **lower-bound** constraints, which correspond to equations of the form $x_2 - x_1 \geq k$, where x_2 is the position of the right-most (top-most) element of the pair, x_1 is the position of the left-most (bottom-most) element, and k is their minimum legal separation. The constraint graph is a global model of the layout in the direction of compaction.

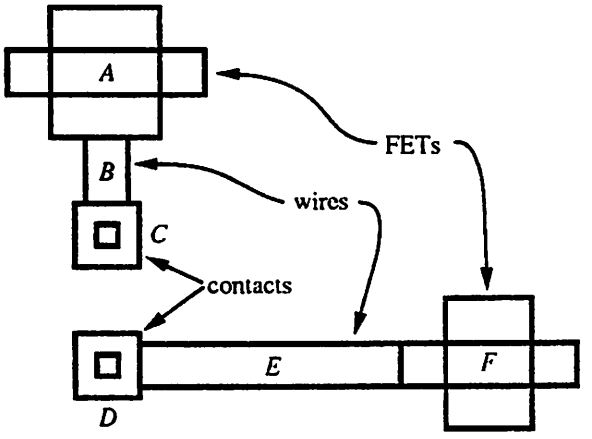


Figure 2.13: An example layout.

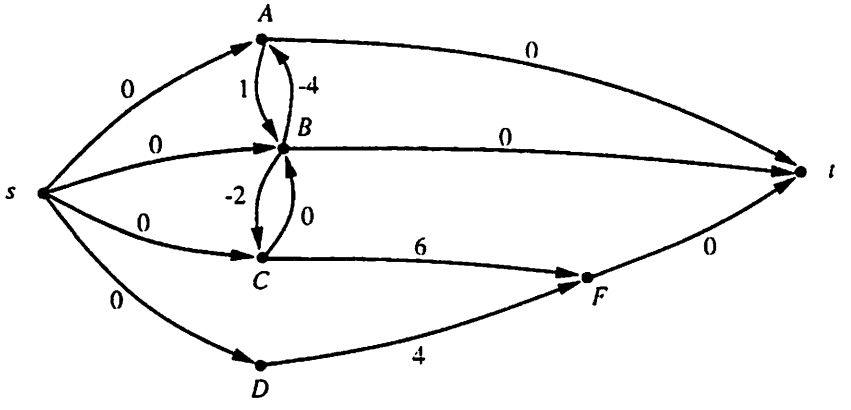


Figure 2.14: Constraint graph for the layout in Figure 2.13.

The graph is constructed by a **constraint-generation** algorithm, which processes the layout at the geometric level. A simple constraint generator would compare every pair of elements in the layout and create a constraint for every pair related by one or more spacing rules.[5] The mapping of elements to nodes can be done in many ways. For example, a connected set of elements maps to a node in CABBAGE [31]. If CABBAGE were run on the layout in Figure 2.13, elements A, B, and C would map to one node instead of three. In SPARCS, the compactor developed in this work, a node is used for each element and for each wire segment that is perpendicular to the spacing direction. Wire segments that are parallel to the spacing direction, like wire E in this example, are not in the graph because they do not generate constraints and because their endpoints can be deduced from the locations of the elements that they connect (D and F in this case). The graph in Figure 2.14 is the graph SPARCS would generate. When multiple elements map to a single node, all those elements move together. Since a horizontal compaction changes the element adjacencies in the vertical direction and vice versa, the constraint graph must be rebuilt after each compaction iteration.

The constraint graph is analyzed, or "solved", to determine the positions of the elements. A longest-path analysis from the source node to all other nodes in the graph gives the left-most position of each element. The longest-path problem is the dual of the well-known shortest-path problem [74]. This first longest-path analysis determines the minimum pitch of the layout in the compaction direction, i.e., the left-most position of the sink node. The sink is assigned this position, then a second longest-path analysis is performed from the sink to all nodes. This gives the right-most positions of the elements, subject to the condition that the pitch is minimum.

Those nodes whose minimum and maximum coordinates are the same are the **critical nodes**, meaning that the corresponding elements limit the pitch of the layout. An edge between two critical nodes, whose weight equals the separation between them, is called a **critical edge**. Generally some of the nodes are critical and some are not. The nodes (elements) that are not on the critical path are called **noncritical** or **slack nodes**: they may occupy the range of positions determined by the two longest-path analyses. The noncritical elements can be positioned to satisfy a secondary objective, which is most commonly wire-length minimization. The problem of optimizing the positions of the slack elements is often

[5] This is not a practical algorithm, because it would execute slowly and it would create many redundant constraints. Efficient constraint generation is considered in Chapter 5.

called the **slack-distribution** problem.

All constraint-based compactors use lower-bound constraints to model minimum spacing requirements. Many current tools allow **upper-bound** constraints as well. Upper-bound constraints have the form $x_1 - x_2 \geq -k$. An upper-bound constraint limits the maximum separation of two elements. An upper-bound constraint plus a lower-bound constraint of the same magnitude is used to implement a fixed constraint; i.e.,

$$x_2 - x_1 \geq k,\ x_1 - x_2 \geq -k \implies x_2 - x_1 = k.$$

When both lower-bound and upper-bound constraints are allowed the constraint graph can be cyclic. A graph is not solvable if it contains a cycle of positive weight, because the longest path can be made arbitrarily long by simply traversing the cycle many times. A positive cycle is often called an **overconstraint.** A practical constraint-based compactor must detect overconstraints and provide an indication to the user as to which layout elements are involved in the positive cycles.

The first constraint-graph compactor, FLOSS, was presented as an alternative to standard-cell style place-and-route. As a result it operates on block-level layouts, not leaf-level layouts. It is a single-level system, for MOS technologies. FLOSS appears to model terminals as areas, not points.

The CABBAGE system is also a single-level system, for NMOS designs. Unlike FLOSS, CABBAGE operates on leaf-level layouts rather than block-level layouts. The point-component layout form is used. Connectivity is not part of the layout data; it is extracted by the compactor in a pre-processing step. Automatic jog insertion is supported as well.

The SLIM program [22], a successor to SLIP, uses a combination of the compression-ridge and constraint-based methods. A constraint-based algorithm is used to cluster elements on critical paths together. Elements in the clusters are moved such that all free space appears at the ends of the clusters. Dunlop called this phase "local compaction". The compression-ridge method is then applied in a "global compaction" phase to remove the re-arranged free space. Multiple-connection symbols are allowed, and jog generation is performed.

The Python program was described in 1982 by Bales [5]. Python differs from previous approaches in that a generic layout model is used. The use of a generic layout model has many important advantages over the typed, point-component model; these advantages are described in detail later in this dissertation.

The initial constraint-based compactors used simple heuristics for positioning the slack elements. One of the first researchers to use wire-length minimization as the objective function for slack distribution was Schiele [67]. A number of other workers have addressed the wire-length minimization problem via a number of approaches, including network flow [24], and a graph-based Simplex algorithm [53]. Wire-length minimization as become the standard objective of slack distribution.

Efficient longest-path algorithms that handle both lower and upper-bound constraints have been presented by a number of workers, including Liao and Wong [51]. Two of the first compactors to provide some mechanism for recovering from positive cycles are described in [20] and [39].

Advantages and Disadvantages The constraint-based formulation is the most general one-dimensional method, as indicated by the fact that the compression-ridge and virtual-grid methods can be cast as constraint-based problems [86]. (The virtual-grid method is described below.) The constraint-graph method uses a global model of the layout, unlike the other two methods. The structure of this model enables a secondary objective to be added easily, and external, user-supplied constraints can be incorporated readily. The compression-ridge method has not been found to be well-suited to leaf-level problems. Constraint-based compaction produces denser layouts than virtual-grid compaction, without a significant speed difference [9,18].

Virtual-Grid Compactors

Virtual-grid compaction was proposed by Weste in 1981 [81,80] as part of the MULGA system. The original intent of the method, as stated in [80], was to produce adequate results quickly. The algorithm was designed for leaf-level MOS layouts, and the typed, point-component layout model is used.

Virtual-grid compaction differs from other relative-grid methods in that the layout grid is used to capture the geometric adjacencies of the elements as well as the topology. (The grid does not serve this purpose for constraint-based compactors.) Since the grid is used in this manner, all elements initially on a gridline must remain on that gridline throughout compaction as shown in Figures 2.15 and 2.16. In a horizontal iteration, for example, the algorithm scans the grid column-by-column. As a column is scanned, element pairs on the same horizontal gridline are compared to calculate a spacing requirement. The

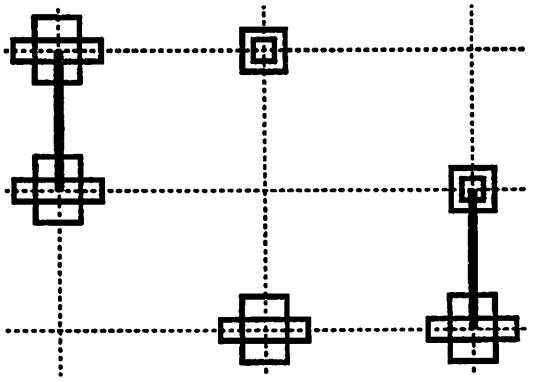


Figure 2.15: Virtual-grid layout.

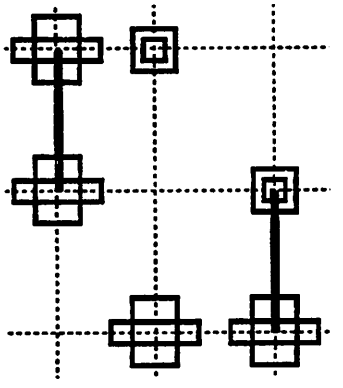


Figure 2.16: Partially compacted virtual-grid layout.

worst-case value for the column determines the gridline spacing.

Weste characterized virtual-grid compaction as a generalization of fixed-grid layout. The algorithm can also be viewed as a variant of the compression-ridge algorithm that does not use shear lines.

A number of enhancements to the virtual-grid method have been reported. In general, an element on gridline k influences gridline locations beyond gridline $k+1$; hence the original algorithm backtracked to gridlines $k-1$, $k-2$, etc. as needed. The algorithm reported by Boyer [11] speeds up the method by using an additional geometric data structure to avoid this backtracking. As noted above, virtual-grid compaction binds elements together if they are initially on the same gridline, regardless of whether they are physically connected or not. A number of "grid-splitting" methods have been proposed to allow the elements more freedom of movement [41,59,7].

Advantages and Disadvantages Initially virtual-grid compaction was advocated as a faster compaction method than constraint-based compaction. However, it appears that the speed advantage of the initial systems was the result of problem simplifications that may not always be desirable.

Several of the important problems with the virtual-grid method are described in [59]. Two of the problems result from undesirable binding of elements to gridlines. The third is the inability of elements on a gridline to move slightly with respect to each other: this prevents, for example, a contact and a wire from offsetting when the contact is wider than the wire. In [59] the authors state that solving these problems resulted in a factor of four increase in runtime over their standard virtual-grid implementation.

An important problem that was not mentioned in [59] arises when multi-terminal elements (i.e., those with more than one connection per side) are required. For example, consider the layout shown in Figure 2.17. The large cell occupies a number of grids. Given that all terminals must lie on gridlines and all elements on a gridline move together, there are two possible ways to perform a virtual-grid compaction in this case. One way is to allow all gridline spacings to change; however, this changes the size of the large element. The second possibility is to prohibit the gridlines that intersect the large element from moving with respect to each other, which preserves the size of the large element. Unfortunately, this can lead to area inefficiencies or rule violations among the other elements on this set of gridlines. This problem also occurs at the leaf level: a simple bipolar transistor spans three

gridlines in one dimension (and one in the other). If these three gridlines were significantly spread apart an unacceptable performance degradation would result.

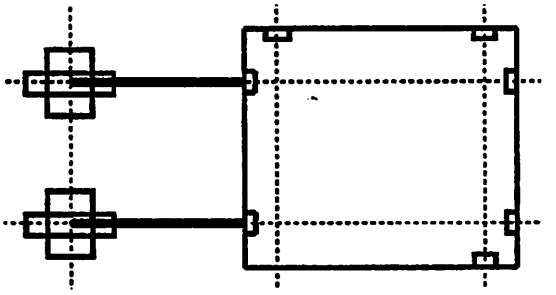


Figure 2.17: Large element on a virtual grid.

2.6.2 Two-Dimensional Compaction

The general two-dimensional compaction problem was shown to be NP-complete in 1982 by Sastry and Parker [66]. This is because this formulation of the problem *does* consider the ordering of related elements, unlike the methods described thus far. That is, for two components that have a design-rule constraint between them, there are four possible topologies as shown in Figure 2.18. The set of topologies can be described via a mixed-integer programming approach, where integer decision variables are used to select a particular topology. The example of Figure 2.18 can be described as follows:

$$x_b \geq c_{11}(k + x_a)$$

$$x_a \geq c_{12}(k + x_b)$$

$$y_b \geq c_{21}(k + y_a)$$

$$y_a \geq c_{22}(k + y_b).$$

with one of the c_{ij} equal to 1 and the rest equal to 0.

Branch-and-Bound

Several workers have attempted to solve the two-dimensional problem via branch-and-bound algorithms. The basic approach can be summarizes as follows. At each step a topology is selected by setting the decision variables. This results in two conventional

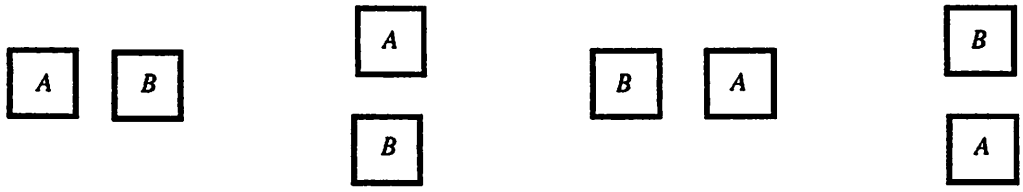


Figure 2.18: Four possible arrangements of two elements.

constraint graphs, one for x and one for y. These graphs are then analyzed using the same longest-path technique used in one-dimensional compaction, which results in a measure of the layout area for the given state of the decision variables. The branch-and-bound technique is used to explore the various topologies systematically, by changing the decision variables at each step.

The method reported by Schlag et al. [68] considers all potential topologies; i.e., for two elements, all four arrangements are possible. The starting layout is collapsed, meaning that none of the spacing constraints is satisfied. The spacing-constraint violations are then removed one-by-one, such that only the best layout is retained at each step. It appears that the implemented algorithm accommodates block-level layouts.

Kedem and Watanabe [36] formulate the problem such that fewer topologies are considered; for two elements, only two of the four possible arrangements are allowed. The starting layout is constructed by beginning with an expanded layout, then setting the decision variables using a greedy two-dimensional heuristic algorithm. The branch-and-bound algorithm then optimizes this starting layout. The implementation reported operates on leaf-level NMOS layouts described via the point-component model.

Advantages and Disadvantages The branch-and-bound approaches can guarantee an optimal result when all four topologies are considered, due to the fundamental properties of the method. However, finding an optimum may require exponential time, because the mixed-integer programming problem is NP-complete [61].

2.6.3 Intermediate Approaches

The large difference in computational complexity between one and two-dimensional algorithms has led several workers to consider heuristic algorithms that lie somewhere between the two. That is, both coordinates are changed during an iteration, but one dimension

is preferred and the input topology is mostly unaltered. These methods try to realize some of the advantages of two-dimensional compaction but at a lower cost in computation time.

Supercompaction

The supercompaction algorithm was introduced in 1983 by Wolf et al. [85,86]. This algorithm does not attempt to minimize area; instead, it minimizes pitch. To minimize pitch in the preferred dimension, the layout is permitted to grow in the orthogonal dimension.

A three-step method is used to monotonically reduce the preferred dimension. Let the preferred dimension be y. In the first step, the critical path in y is determined. The y dimension can only be reduced further if the critical path is broken. Accordingly, the second step breaks the critical path by pushing elements apart in the x dimension. The third step is simply a one-dimensional compaction of the modified layout in the y dimension. This sequence can be repeated until the pitch in y is irreducible. The second, or "shearing" step is the most important of the three. It finds a cutset of the critical-path graph that is feasible and that divides the graph such that the source and sink are in different components. The cutset is feasible if each of its edges can be broken by shearing apart the element pair in x. When no such cutset exists the layout is irreducible and the algorithm halts. Results were reported for leaf-level NMOS designs.

Zone Refining

The zone-refining algorithm is a heuristic two-dimensional approach proposed by Shin in 1986 [71,72,70]. The name follows from its resemblance to a physical process for removing impurities from a crystal ingot.

The algorithm minimizes pitch, with the addition of two-dimensional movements of the elements. The layout is initially compacted via one-dimensional compaction to produce a starting configuration. The zone-refining phase consists of one or more subsequent passes. In a vertical pass, the layout is processed from bottom to top on an element-by-element basis[6]. Elements are removed from the bottom of the unprocessed part of the layout (the "ceiling" of the top configuration) and reassembled on the top of the processed part (the "floor" of the bottom configuration). When an element is moved from the ceiling to the floor, its lateral position is altered as well, within a range about its current x-coordinate.

[6] In practice, small clusters of elements are moved as units.

The lateral moves are performed such that the pitch of the layout in y does not increase. Because the lateral movements are local, a sequence of several passes in the same direction can be beneficial. This implementation processes NMOS and CMOS layouts.

Advantages and Disadvantages The intermediate approaches are more powerful than one-dimensional compaction and less powerful than two-dimensional compaction. Likewise, they lie between the two in CPU usage. Their performance compared to the other approaches is likely to vary from design-style to design-style, due to their heuristic nature.

2.6.4 Hierarchical Compaction

A number of one-dimensional constraint-based compactors and virtual-grid compactors have hierarchical capabilities. The methods used are described below.[7]

Constraint-Based

The most prevalent hierarchical method used in constraint-based systems will be referred to herein as **stretch-and-abut**. It is described, for instance, in [39.24]. The basic approach, for any level above the leaf level, is to create a cell at the current level by tiling together rectangular subcells that connect by abutment. The "compaction" process is that of manipulating the subcells such that they fit together in this fashion. The stretch-and-abut method differs substantially from leaf-cell compaction; in fact, no process analogous to leaf-level compaction is performed.

To illustrate the method, let the leaf-level be level 0 and consider a cell at the first hierarchy level (level $i = 1$) above the leaf level. Each subcell is an instance of a rectangular leaf-level cell which has been compacted to its minimum dimensions.[8] The geometric model of a compacted subcell consists of a rectangular border with terminal locations on the border (Figure 2.19). In general the subcells from level 0 (the leaf cells) must be stretched, since they are required to connect by abutment on level 1. After stretching, each subcell's border is expanded by an amount equal to half of the maximum spacing rule for the given technology. The terminals are extended to the new cell borders, and finally the cell at level 1 is composed by tiling together the subcells. Expansion of the subcell borders is

[7] Some experiments in hierarchical zone-refining are described in [72]; otherwise, little work has been done in two-dimensional hierarchical compaction.

[8] In practice, as described later in this section, the subcell graphs are generated but not solved at this stage.

necessary because the subcell internals are not modeled; the expansion insures that they may be abutted without creating any design-rule violations (Figure 2.20).

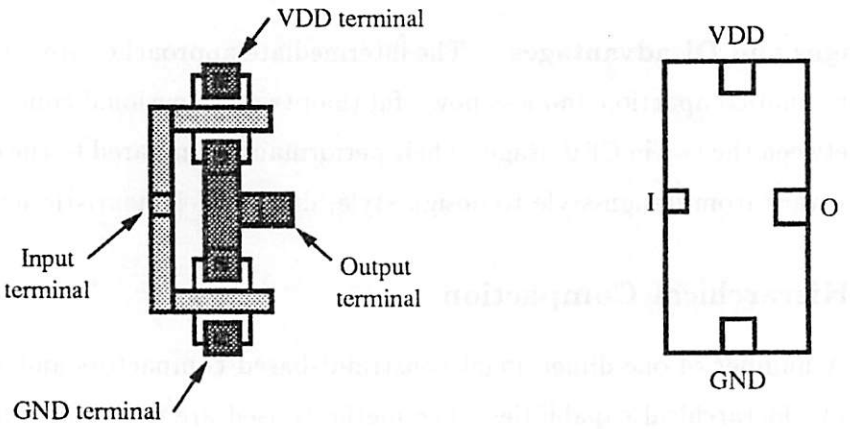


Figure 2.19: Compacted cell and border model.

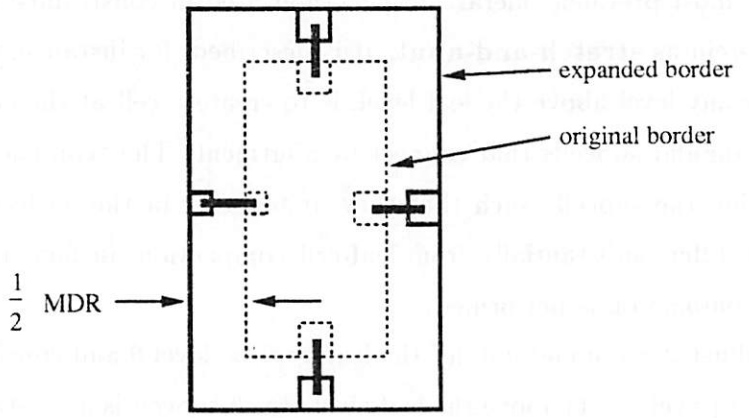


Figure 2.20: Border expansion.

The stretching step is performed as follows. Consider a cell Q comprised of two neighboring subcells as shown in Figure 2.21. A constraint graph for Q is generated by first representing each subcell by a graph that describes how its terminals can move relative to each other and relative to its borders. These graphs are called **port-abstraction graphs.** A port-abstraction graph can be derived from a constraint graph via a partial transitive-closure calculation [24].[9] The subcell terminals that must align are then bound together

[9]This is an oversimplification. A cell cannot be independently stretched in x and y if ordinary constraint graphs are used in both dimensions. Instead, one of the two constraint graphs must be augmented with additional constraints before the port-abstraction graph is derived [24].

with fixed constraints; such constraints are called **pitch constraints.** This creates a graph for Q which can be analyzed with the usual critical-path algorithm. After the graph for Q is solved the terminal positions for subcells A and B are known. The new terminal positions are applied to the constraint graphs for A and B; the graphs are then re-solved to yield the stretched subcells. The final result for Q is shown in Figure 2.22.

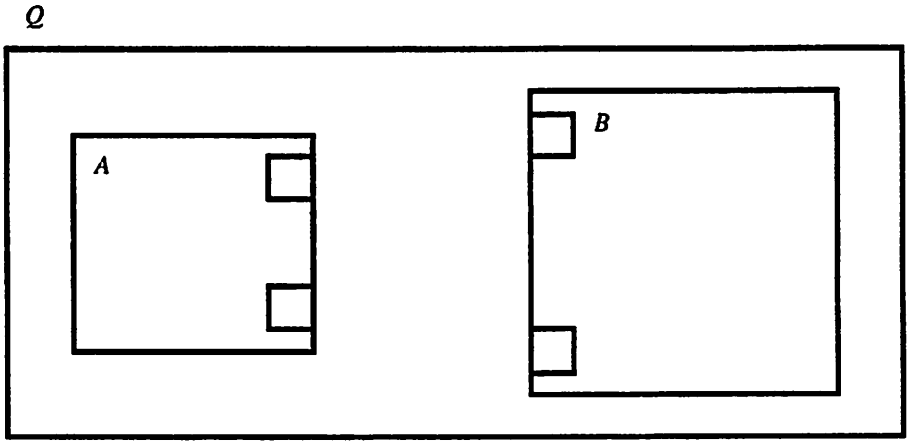


Figure 2.21: Cell Q before pitch-matching.

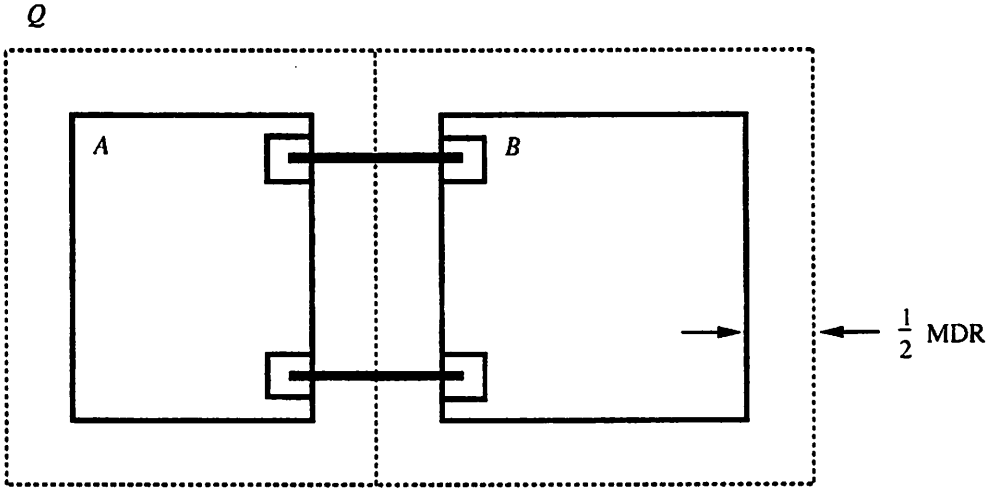


Figure 2.22: Cell Q after pitch-matching.

Multiple hierarchy levels are handled by repeated application of the steps outlined above: the overall method consists of three passes through the design hierarchy. The first pass is a bottom-up traversal that operates as follows. The constraint graph for level i is

generated by computing the port-abstraction graphs for the subcells at level $i-1$, instanciating them at level i, and finally adding the pitch constraints. The port-abstraction graph for level i can then likewise be computed. When the constraint graph for the top level is created the first bottom-up pass is completed. The second pass is top-down. Port positions are found for each level of the hierarchy, by solving the graph at level i and propagating the port positions down to level $i-1$ as fixed constraints. When the leaf level is reached all terminal positions are known and the leaf cells can be stretched to their final dimensions. Finally, a third pass through the hierarchy bottom-up is used to tile each level.

Advantages and Disadvantages The stretch-and-abut method can produce good results for some design styles. However, the method has a number of restrictions; namely, pitch-matched subcells are required, the subcells must be rectangular, the inter-cell spacing is worst-case, and copies of the same subcell are not necessarily the same after pitch-matching. Pitch-matching does not work well for irregular design styles such as macrocell. In some cases, non-rectangular cells are desirable. If copies of a subcell are stretched differently from one another, then the hierarchy is effectively lost.[10]

Perhaps the biggest problem with this method is the worst-case inter-cell spacing. Some technologies, such as CMOS, have a wide range of spacing-rule values. In Mosis CMOS, the largest and smallest rules differ by a factor of 9 [2]. This means that the subcell separation might be a factor of 9 larger than necessary.

A solution to the inter-cell spacing problem was proposed in 1986 by Reichelt [65]. In this scheme additional elements are modeled in the port-abstraction graphs. The method seems promising for reducing the inter-cell spacing, but it does not solve the other problems with the stretch-and-abut method.

Virtual-Grid

Virtual-grid systems process hierarchical designs in a manner similar to the stretch-and-abut method [81,82,3,73]. That is, the leaf cells are compacted, then higher-level cells are assembled by stretching and tiling.

According to Tan [73], the initial systems either used the worst-case inter-cell spacing [3], or produced results that were not necessarily design-rule correct [81,82]. The system described by Tan in [73] produces correct results without a worst-case spacing by

[10] A solution to this problem is proposed later in this dissertation.

iterating over the cell hierarchy until there is no movement of the gridlines. This method reportedly converges after four iterations in a typical case.

The advantages and disadvantages of the virtual-grid systems are essentially the same as those of the constraint-based systems.

2.7 Summary of the Previous Approaches

An ideal symbolic layout and compaction methodology has several advantages over mask-level layout generation, as outlined in Chapter 1. The preceding review indicates that a number of systems have realized these advantages, but only for a limited set of technologies and design styles.

The majority of the systems that process leaf cells handle digital MOS designs only. Although it is not always stated directly, it appears that all such tools (except Python) use a typed, point-component model as the leaf-level layout representation. The systems that perform hierarchical compaction generally use the stretch-and-abut method with a rectangular cell model; most also enforce a worst-case intra-cell spacing.

Very little work has been done in several areas, such as the application of symbolic layout and compaction to non-MOS technologies like bipolar and BiCMOS. The additional constraints imposed by analog designs have not been considered. The design and use of layout-modeling techniques that are technology and hierarchy-level independent has not been well-studied.

2.8 Goals and Requirements

The overall goal of this research has been to develop symbolic layout and compaction techniques that apply to a much broader class of IC designs than the previous systems. To achieve these goals, the system must deal with a wide variety of elements as primitives. It should likewise handle any technology. It must not impose any particular design style on the user or on other tools. It must operate hierarchically, with or without pitch-matched cells, and without a substantial area penalty over flat design. It must be efficient, in terms of runtime and in terms of the density of the layouts produced. It must be as modular and extensible as possible.

These considerations have led to the selection of the following symbolic layout and compaction methodologies.

2.8.1 Choice of Compaction Technique

No single compaction approach is best in all circumstances. It is thus important to choose a technique that is effective for the broadest class of problems. Fewer layouts will be produced manually as design automation matures. Automation has established a trend towards layouts that are regular, not irregular. Layout sizes are constantly increasing, as is the quality of machine-generated layouts. Most design styles tend to impose a number of constraints on the layout (bus orientation, pin positions, cell pitch, etc.), thereby constraining the compactor moves. These observations affect the choice of compaction technique. The compactor must produce good, predictable results in a reasonable amount of time. It must be able to handle large designs. It should be controllable, so that compacted cells can be easily used to compose higher-level cells.

There are several compelling reasons to choose a one-dimensional compaction method. From a practical perspective, two-dimensional compaction is superior to one-dimensional compaction *only* in cases where it can generate a *significantly* better result, in an acceptable amount of *excess* time, over a one-dimensional method. True two-dimensional algorithms will, in general, produce better results than one-dimensional algorithms. However, the general two-dimensional problem is NP-complete and hence is not a useful formulation for a practical system. Heuristic two-dimensional algorithms are slower than well-implemented one-dimensional algorithms, and it has not been proven that they are superior in terms of layout area to one-dimensional compaction with jog insertion [9].

One-dimensional compaction algorithms are faster than two-dimensional algorithms. One-dimensional compaction has polynomial complexity, and the most general one-dimensional algorithms have complexities that approach $O(n \log n)$ for real examples. In comparison, two-dimensional compaction is limited to smaller problems, and is likely to produce significantly better results only in those cases where the starting layout is irregular, far from optimal, and lightly constrained. By the above argument such layouts are likely to be rare.

The constraint-based method is the most general one-dimensional technique. It is the only one-dimensional method that employs a global model of the layout. It accommo-

dates external constraints and secondary objectives readily, and has been shown to be as efficient as other one-dimensional methods. These considerations have led to the selection of one-dimensional, constraint-based compaction for this research.

2.8.2 Choice of Symbolic Layout Technique

As noted above, a layout model is needed which can capture any layout, regardless of its technology or hierarchy-level. This argues strongly for a type-free representation, which includes geometric, connectivity, and logical data only. As a result, a generic modeling approach has been employed in this project. The design of this model is presented in the following two chapters.

Chapter 3

Symbolic Layout Models

3.1 Introduction

The symbolic-layout representation is a key factor in determining whether the desired advantages of symbolic layout and compaction can be achieved, for two reasons. First, the representation determines the class of designs that can be accommodated. A simple layout model might only handle NMOS designs at the leaf level, while a more general model might handle multiple technologies or hierarchical designs. Second, it plays an important role in determining the area-efficiency of the results, because the compactor must deduce, from the layout representation, the set of constraints that leads to an area-efficient result. In other words, the representation must contain sufficient detail that the degrees of freedom in the layout can be expressed and exploited. For example, a model that only allows rectangular shapes will lead to poor results when applied to layouts with polygonal shapes, since the irregularity cannot be described. Figure 3.1 illustrates this problem.

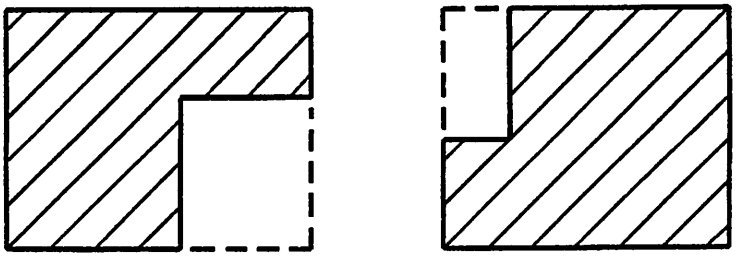


Figure 3.1: Polygonal shapes modeled as rectangles.

This chapter begins with a list of the requirements a layout representation should

fulfill. The existing symbolic-layout model is analyzed with respect to these requirements. An idealized layout representation is then presented, followed by an outline of the high-level characteristics of the layout form proposed in this project.

3.2 Symbolic-Layout Modeling Requirements

The following paragraphs summarize the requirements that must be fulfilled by a symbolic-layout model. These requirements follow from the goals outlined in the previous chapter.

Representable Elements and Extensibility. The layout model determines which elements (primitives) are allowed, and the amount of detail in their description. In addition, it determines how much effort is required to add a new element to the system. A new element might be a device from the same technology but with a different shape, or a device from another technology.

The layout model should either accommodate all possible elements, or it should be readily extensible so that new elements can be added easily.

Hierarchy-Level Independence. The layout model has a strong effect on the use of the system for hierarchical design. A symbolic layout and compaction system should employ a hierarchical data model; a spaced cell from level i in the hierarchy should be usable as a primitive component on level $i + 1$ in a natural and efficient manner. In addition, the model should not differ from one level of the hierarchy to another. That is, all layouts should be represented in a uniform and consistent manner, regardless of hierarchy level.

This requirement implies in part that the layout model should support arbitrary cells, because the set of all possible compacted cells is infinitely large.

Terminal Merging. The spacing rules for IC layouts are connectivity-dependent, meaning that features that are electrically connected can be located closer together than features that are not connected. In compaction terms, cells should conditionally overlap (merge) at their terminals, as a function of connectivity.

The problem of formulating the spacing constraints as a function of connectivity is called the **terminal merging** problem. The example shown in Figure 3.2 demon-

strates terminal merging for a leaf-level MOS design. Nearly all compactors that operate on leaf-level layouts implement some form of terminal merging. Terminal merging should also occur at levels above the leaf-level, as shown in Figure 3.3. The layout model should be capable of describing mergeable areas of the components, regardless of their hierarchy level.

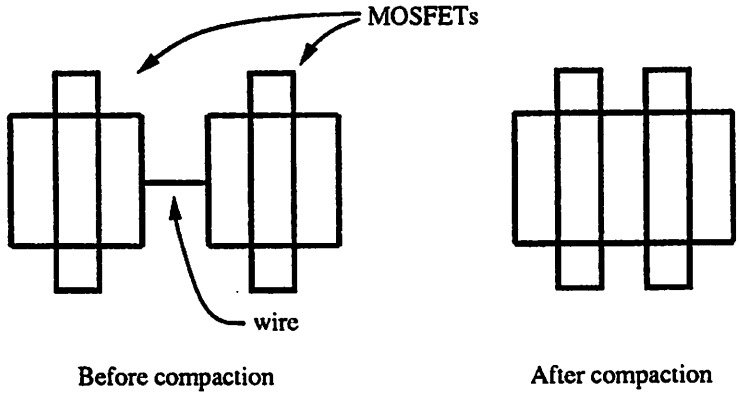


Figure 3.2: Terminal merging at the leaf level.

Tuneable Geometric Detail. The layout model should be constructed such that the level of geometric detail of the components can be adjusted by the user. This capability is especially important when a hierarchical design methodology is used.

A compacted cell should be abstracted geometrically before it is used as a primitive component on the next level, to suppress unnecessary detail and thereby reduce the computation time needed for compacting the next-level layout. A compacted cell denoted A_1 is shown in Figure 3.4. Several copies of the cell are arrayed together to form cell B_1 in Figure 3.5. In compacting B_1, the subcell A_1 given in Figure 3.4 is more detailed than necessary. A more abstract representation of A_1 is depicted in Figure 3.6. This representation is less detailed; hence B_2 (Figure 3.7) compacts more quickly than B_1. If the level of detail of the abstraction can be varied, then the tradeoff of CPU-time versus the area of the compacted result can be varied as desired by the user.

Technology Independence. The model should be general enough that any common IC technology can be accommodated. Large-scale program modifications should not be needed when a new technology is introduced.

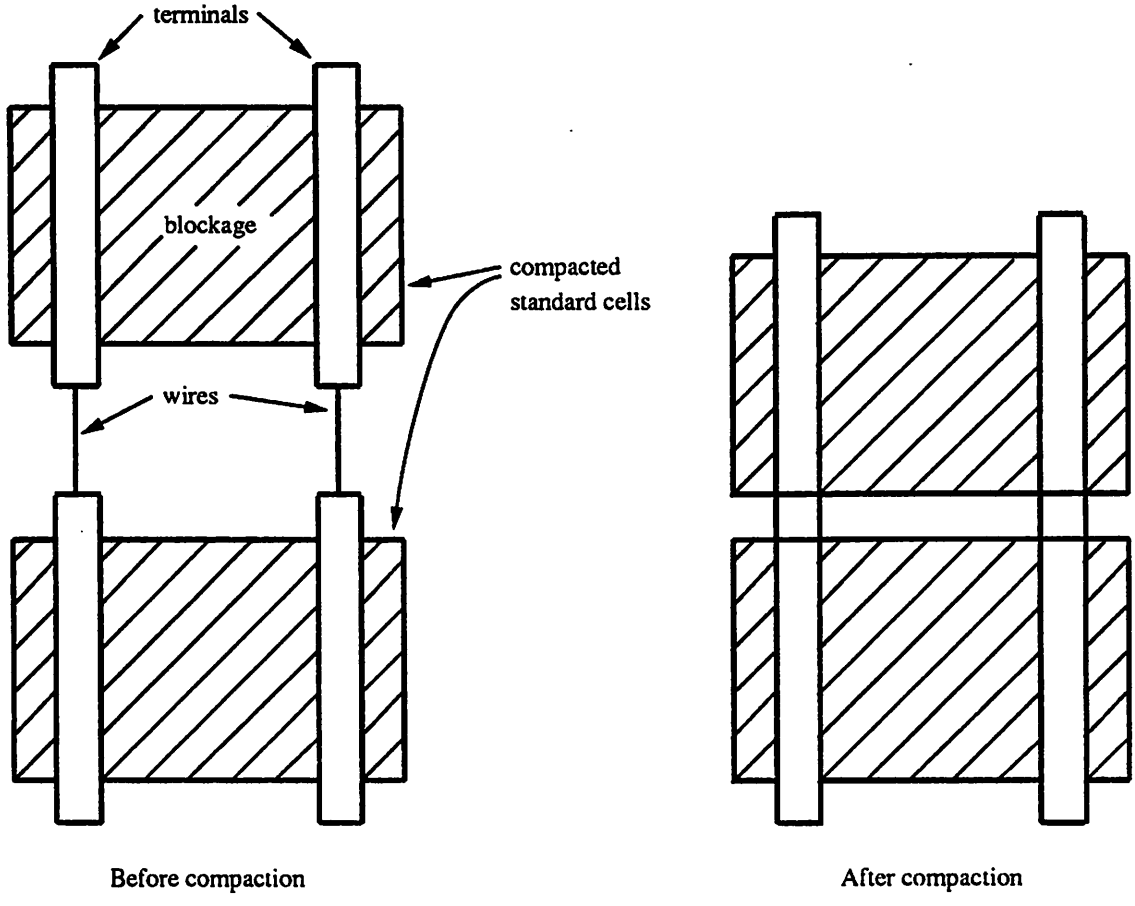


Figure 3.3: Terminal merging at higher levels.

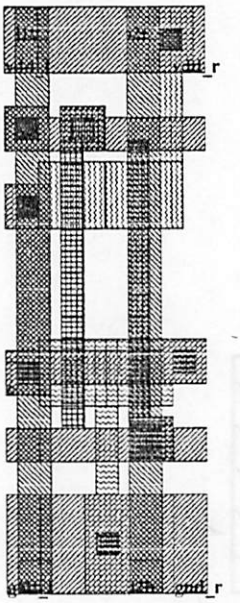


Figure 3.4: Compacted cell A_1.

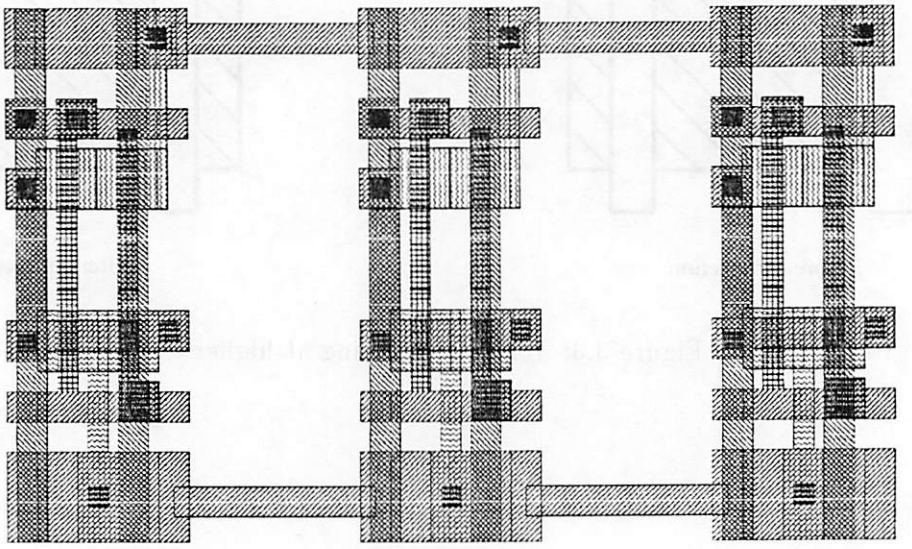


Figure 3.5: Cell B_1, using arrayed cells from Figure 3.4.

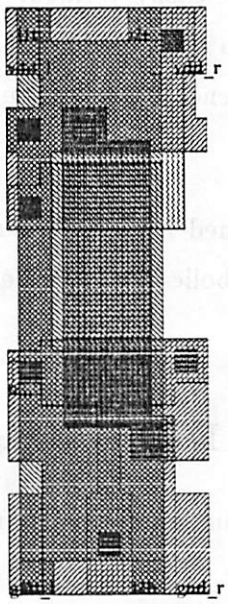


Figure 3.6: A more abstract representation of the cell in Figure 3.4.

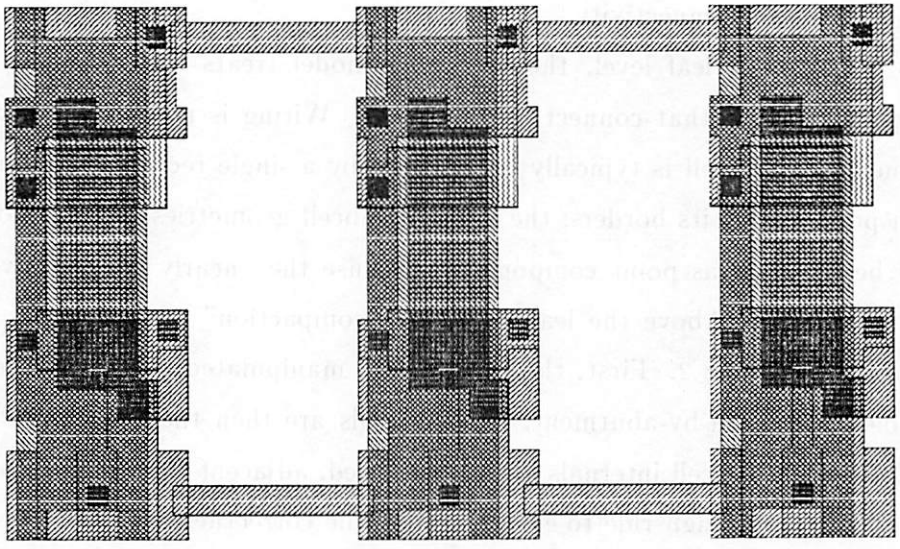


Figure 3.7: Cell B_2, using arrayed cells from Figure 3.6.

Level of Abstraction. The level of abstraction of the model should be selected such that the symbolic layouts can be easily verified. Often-used information should be stored with the model, rather than derived. In addition, the model should support a general form of annotation and a general mechanism for adding structural information to the data.

The requirements outlined here are the most important ones from a compaction perspective. Other issues in symbolic-layout modeling are described as they arise in following sections.

3.3 Advantages and Disadvantages of the Existing Model

Although symbolic-layout models vary from system to system, their characteristics are similar in a broad sense. The prevailing leaf-level model is the point-component model, which was outlined in the previous chapter. A point-component layout consists of typed symbols interconnected by zero-width wiring. The type value is used in the symbolic-translation step to map symbols into sized mask-level equivalents. It is often used in the compactor as well, to calculate spacing constraints between pairs of elements. In the compaction operation, the point components translate. The wiring translates and/or changes length to maintain connectivity.

Above the leaf level, the prevailing model treats the layout as a collection of rectangular subcells that connect by abutment. Wiring is not present, nor are leaf-level components. A subcell is typically represented by a single rectangular blockage with connection points along its borders; the internal subcell geometries are not modeled. Subcells cannot be modeled as point components, because they nearly always have multiple connections per side. Above the leaf level, the "compaction" operation has two phases, as described in Chapter 2. First, the subcells are manipulated such that they pitch-match, to enable connection-by-abutment. The subcells are then tiled together to complete the layout. Because the cell internals are not modeled, adjacent subcells are usually separated by the worst-case design rule to ensure design-rule correctness.

The level of abstraction of the traditional symbolic-layout model is between that of the mask level and the schematic level. A circuit schematic consists of functional units and their interconnections (connectivity). Most of the data at the schematic level is structural

and logical; little physical information is present. While a schematic may contain information on the sizes of the circuit elements, the placement and routing data is not present at all. On the other hand the devices that comprise a functional unit are associated with one another, and the description of the function that each unit performs is present. A design at the mask level contains all element size, placement, and routing data, but none of the grouping and connectivity data present at the schematic level. In particular, the geometries that comprise the basic electrical elements are not associated with one another.

A typical symbolic layout has some of the structural information present in a schematic but missing from a mask-level description. That is, the basic electrical components are identified, but other schematic-level information (e.g., logic function, connectivity) is usually not present. There is more geometric information in a symbolic description than in a schematic-level description, but less than that in a mask-level description. Hence the level of abstraction lies between the mask and schematic levels. It is a subset of these levels as well; it contains less geometric information than the mask level, and less logical/structural information than the schematic level.

3.3.1 Advantages

The simple nature of the point-component model leads to several advantages. The component sizes can be easily parameterized, and the connectivity of the layout can be extracted readily. A point-component layout can be scaled in a straightforward manner. The most important advantage of this model, however, is that the terminal-merging problem can be easily solved. This is because the restriction to a few types with simple, known configurations allows for the use of a case-analysis approach to merging.

Several researchers have described their use of type information in performing terminal merging. In CABBAGE [31], the element types are used along with layer and connectivity information to determine mergeability. The actual spacing requirements are determined by special-purpose subroutines that use this data as input. If a new element is added, new subroutines are written to accommodate it. The method described in [22] works as follows. If two symbols are electrically connected, then all spacing rules with respect to the interconnecting layer are ignored for that pair of symbols, if the widths of the terminals on the interconnection layer are the same for both symbols. This scheme allows, for example, a FET and a contact to merge if their widths are the same, but not

if they differ. The compactor described by Kingsley [39] develops all spacing values by pairwise analysis of symbols using their types. Several type-dependent operations used to enable FET merging have been described by Tan [73]. These include pre-merging the source and drain rectangles of adjacent FETs, and marking diffusion edges as invisible if they are part of a FET and are coincident with the FET's gate. In the compactor described in [30], transistors are modeled differently depending on whether or not they are being compacted against metal wiring. The virtual-grid system reported in [10] models the diffusion part of a MOSFET as a wire. Watanabe [79] uses a scheme which ignores spacings between two rectangles from different elements that are electrically connected. This will prohibit two FETs from merging fully, because the polysilicon gate of one device will generate a rule spacing to the diffusion terminal of the other device.

3.3.2 Disadvantages

Unfortunately, the typed, point-component model has several serious disadvantages. These follow from the simple shapes it imposes, and from the use of the types themselves.

Rectangular shapes are inadequate for asymmetric elements, such as the bent MOSFET shown in Figure 3.8. The simple connection mechanism (i.e., the intersection of a wire-segment endpoint with the center of a symbol) breaks down for some common IC technologies. For example, a bipolar device with a single base, emitter, and collector cannot be modeled, because a wire drawn to the center of the transistor intersects two terminals and thus the connection is ambiguous (Figure 3.9). Furthermore, bipolar transistors often have multiple emitter and/or collector terminals. Other leaf-level elements may have multiple connections per side as well.

If the layout model is typed, then the system is limited to those known types. Such a limitation makes it difficult to achieve technology independence, hierarchy-level independence, and extensibility. If types were only used in symbolic translation, a typed model might not be severely limiting. However, the types are often used within the compactor as well, as described in Section 3.3.1. Note that the type of an element also designates its *shape*. Hence a typed system is not only limited to certain classes of elements, but also to certain configurations of the supported types. For example, the two components in Figures 3.8 and 3.10 would have different types, even though they are both MOSFETs, because

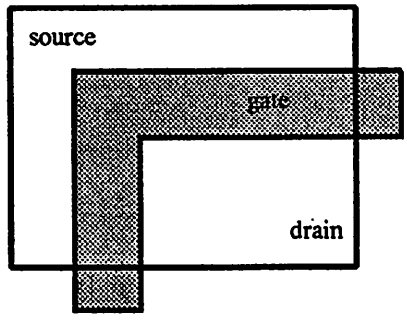


Figure 3.8: MOSFET that cannot be modeled as a point component.

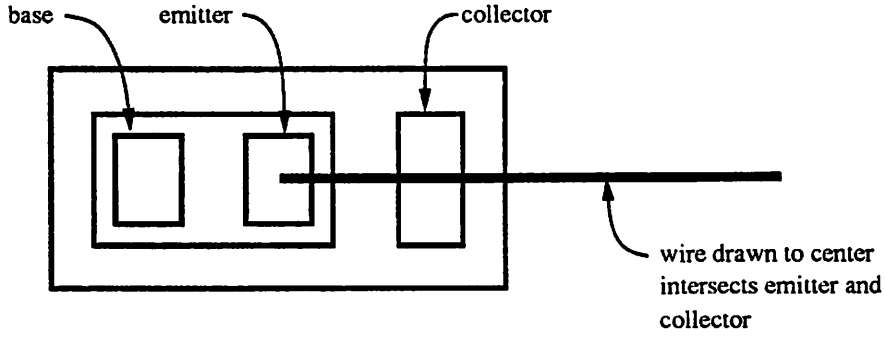


Figure 3.9: Ambiguous connection of a bipolar transistor.

their shapes, and hence their mapping and spacing requirements, differ.

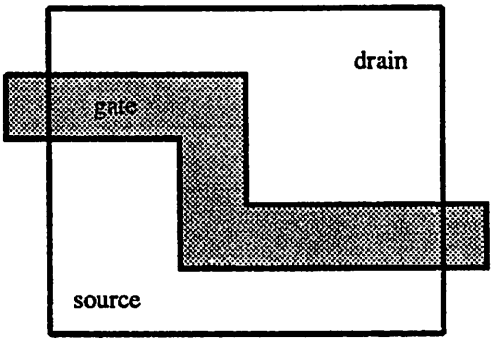


Figure 3.10: MOSFET with different type than that in Fig. 3.8.

The use of type information hinders extensibility, because the system must be modified to understand and manipulate new types whenever a new element is added. Technology independence is difficult to achieve, because incorporating a new technology usually involves dealing with new elements, hence new types. The compacted cells that are created in hierarchical compaction, which are used as primitives on higher levels in the design hierarchy, are effectively new types. The type of a compacted cell cannot be pre-encoded; thus typed systems use one model for the leaf level, and a substantially different one for all higher levels. As described previously, the higher-level model is comprised of abutting rectangular cells, and "compaction" above the leaf-level is actually stretch-and-abut cell assembly. A typed system is therefore hierarchy-level dependent.

In general, higher-level layouts should not be restricted to pitch-matched, rectangular cells. A pitch-matching design style does not work well in all cases, nor do rectangular cells. Adjacent cells should not be separated by the maximum design rule; they should be compacted together according to the per-layer rules, and they should merge at their terminals in the same manner as leaf-level elements. The typed model supports merging at the leaf level readily. However, performing merging through the use of typed elements precludes merging from occurring for cells on levels above the leaf level. Many layout styles employ mergeable cells above the leaf level, as exemplified in Figure 3.3.

3.4 An Ideal Symbolic-Layout Model

The previous section has illustrated the problems with the standard method of modeling symbolic layouts, both at the leaf level and for hierarchical cases. Fortunately, there is no fundamental reason to model IC layouts via the typed, point-component model. Assume instead that there is a way to describe IC layouts in a generic, type-free fashion, using only geometric and connectivity information. In this description cells would be allowed to have any Manhattan shape, and any number of terminals. Assume as well that compaction algorithms can be developed for this model that compute element-to-element spacings effectively, using only geometric and connectivity data. Under these assumptions, all the problems mentioned previously, for typed systems, can be solved.

For example, such a system would not be limited to simple elements (i.e., point components). New elements and new technologies could be readily accommodated, since all that would be required is the addition of the geometric definitions of the new primitives; no program modifications would be needed. Hierarchical compaction could be performed in a more natural fashion, since compacted cells would be modeled in the same manner as leaf-level primitives, and thus treated in the same manner by the compaction algorithms. This would remove the distinction between leaf-level and hierarchical compaction. In addition, a pitch-matched design style would be optional, not mandatory.

It appears that a generic layout description could lead to a system that is general, extensible, technology independent, and hierarchy-level independent. The remaining question is whether such a system would be efficient, in terms of both layout area and CPU time. It is shown later in this dissertation that an efficient system *can* be achieved when a generic model is used, provided the model itself is properly designed, and provided the constraint-generation algorithm used in the compactor is properly designed.

Symbolic layout and compaction systems use types in two operations, namely symbolic translation and element-to-element spacing calculations. A generic layout model would impact these two aspects of a symbolic system; the effect on each is described in the following subsections.

3.4.1 Symbolic Translation

Since a generic model has no types, there is no symbolic-translation operation. All layout elements are sized in a generic description. This *does not* mean that parameterized

layout descriptions are disallowed; it simply means that such descriptions exist outside the system.

The point-component model is one specific, high-level, parameterized layout form that is suitable for a certain class of MOS designs. This form is useful in some circumstances, just as other parameterized layout forms might be useful in others. A higher-level description can be processed by a generic system by simply compiling the high-level form into the generic form. It may be advantageous to produce the parameterized form as output of the system as well. For a point-component layout form this is not difficult, since each component is parameterized in a very simple fashion. The parameter values can simply be carried through the system, allowing for transformation back into the point-component form at the end.

3.4.2 Layout Compaction

Compaction cannot occur until all component sizes and wire widths are mapped in. Hence, the input layout to *all* compactors may be viewed as pre-sized. In principle, the typed/parameterized nature of the traditional model need not be of any consequence at the compaction level.

Unfortunately, many compactors make use of the element types in calculating the element-to-element spacings, as described previously. One reason for this is to facilitate terminal merging. Terminal merging is a difficult problem if the components are generic. However, it will be show later in this dissertation that this problem can be efficiently solved.

3.5 Proposed Symbolic-Layout Model

The following sections describe a new modeling approach that attempts to implement the ideal generic model described in Section 3.4. This model will be referred to as the Berkeley Layout Model, or BLM.

Distinction can be made between symbolic layouts at the database level, at a particular hierarchy level, and at the component level. The component-level representation will be called the **cell model**. The cell model of a component is its geometric specification. The term **layout model** will be used to refer to the representation of a symbolic layout *from a particular level* in the design hierarchy. The layout model consists of the components and wiring, plus the connectivity data, for that given level. Note that many tools, including compactors, processes one level of hierarchy at a time. The database representation will be

referred to as the **data model.** The data model contains *all* the design information; i.e., the hierarchy, connectivity data, geometry, functional information, etc.

The layout and cell models proposed in this dissertation are third-generation models. Their precursors are the models used in CABBAGE [31] and Python [5]. The data model is a second-generation model. The first-generation model was implemented in the Hawk/Squid system [37]. Comparisons with these previous models will be made as the proposed model is described.

3.5.1 Layout and Cell Models

At compaction time, an IC layout from *any* level in the hierarchy can be described as a collection of fixed-size cells interconnected by wiring. The wire segments are fixed-size in their width dimension and flexible-sized in their length dimension. Consider a leaf-level MOS layout. At compaction time the transistors and contacts are fixed-size components, even though they are often described in a parameterized manner before symbolic translation. Likewise, at compaction time, the wire segments have known, fixed widths. Similarly, a microprocessor layout at the top level of the hierarchy consists of a few large, multi-terminal, fixed-size cells (ALU, register file, controller, etc.) interconnected by wire segments. It is apparent that the microprocessor layout and the leaf-level layout can be modeled *in the same manner* if the cell model can capture, for example, symmetric components with a single connection per side, *and* asymmetric components with multiple connections per side.

A key ingredient in the BLM is a generalized cell model that can handle a very wide variety of components efficiently. The cell model is comprised of blockages and connection areas (terminals). The blockages are defined on a per-layer basis. Any Manhattan polygon is a legal blockage. A terminal consists of one or more rectangular areas. Terminals are not restricted in number or location. The cell model is described in detail in the following chapter.

Wiring is modeled as sets of two-point segments. An instance of a simple cell, called a **routing instance** or **routing terminal**, is placed wherever multiple segments intersect. Several examples are given in Figure 3.11. This mechanism leads to a uniform wiring model; all segments are two-point segments, and all segments are terminated at both ends by terminals.

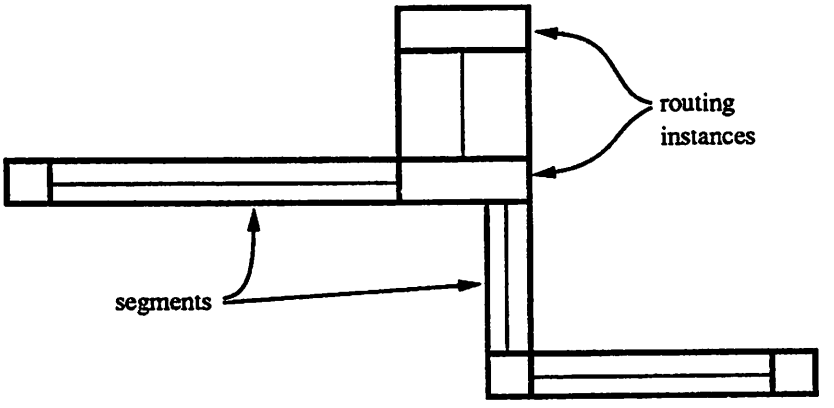


Figure 3.11: Wiring decomposed into segments and routing instances.

3.5.2 Data Model

The project described in this dissertation is part of a large IC design system under development at U.C. Berkeley. The data manager in this system is called *OCT* [19]. The OCT system is a low-level, object-oriented system, meaning that it provides mechanisms for storing objects, interrelating objects, accessing objects, etc. However, OCT itself imposes very little structure on the data. A data model is defined on top of OCT by specifying the relationships between objects. In OCT terminology, such a specification is called a **policy**; the data model for symbolic layouts is called the **OCT symbolic policy** [15]. The symbolic-layout policy is one of several policies that are currently implemented in the Berkeley system. Objects in OCT are interrelated by an operation called **attachment**. Attachments are directional; the parent object is said to **contain** the child object, and from the context of the child, the parent object is its **container**.

The highest-level object in OCT is the **cell**. A cell has one or more **views**, each of which has one or more **facets**. Editing operations occur at the facet level. For example, the process of creating a symbolic layout of an inverter circuit corresponds to creating a facet of a view of a cell, which might be named "*contents*", "*unspaced*", and "*inv*", respectively. The output of a compaction run on the inverter might be facet "*contents*" of view "*spaced*" of cell "*inv*". A facet is identified by a triple of the form `cellname:viewname:facetname`, e.g., `inv:spaced:contents`. When a cell is included in another cell to create a hierarchy, the child cell is represented by an **instance** object; the child is often referred to simply as an instance. An instance of cell C is a reference to C, not a copy of it, plus a transformation

which locates it in the coordinate system of the parent. The data that actually describes C (i.e., one or more facets) is called the **master** of C.

In the symbolic policy, each cell has two representations, a definition and an abstraction. The definition is stored in the **contents** facet of the cell; the abstraction is stored in the **interface** facet of the cell. The terms definition/contents and abstraction/interface are used synonymously in this dissertation. A cell's contents consists primarily of its subcells, their interconnections, and the inputs/outputs of the cell, which are called its **formal terminals**. A cell's abstraction is a summary of this information that is sufficient to represent the cell on higher levels in the hierarchy. The geometric part of the abstraction describes the cell's blocked areas and formal terminals. The connectivity part describes sets of formal terminals that are electrically equivalent.

Additional structure can be built upon OCT data using the **bag** object. A bag is an object that is used to group together other objects of any type. An example of bag usage is given below. The symbolic policy allows for annotation of the design via the **property** object. A property can be attached to any object, and it consists of a name, a type, and a value. For example, a formal terminal can be marked as a signal net by attaching to it a property named TERMTYPE, of type STRING, with value "SIGNAL".

Contents Facets

The contents facet `simple:unspaced:contents` for the symbolic layout shown in Figure 3.12 is depicted in Figure 3.13. The symbolic hierarchy is terminated at the leaf level by physical views. Since `simple:unspaced:contents` is a symbolic leaf cell, the masters of A and B are physical views. Assume A's master is `NMOS.2x3:physical:contents` and `NMOS.2x3:physical:interface`, and that B's master is `COND.4x4:physical:contents` and `COND.4x4:physical:interface`. The master of A has three formal terminals, named "*gate*", "*drain*", and "*source*". Whenever an instance is created, **actual terminals** are created by OCT in one-to-one correspondence with the formal terminals of its master. The actual terminals are attached to the instance object as shown in Fig. 3.13. An actual terminal is a logical reference to the formal terminal; that is, an actual terminal is not geometric. In this sense, an actual is to its formal in the same way that an instance is to its master.

Connectivity is represented explicitly, through net objects. According to the sym-

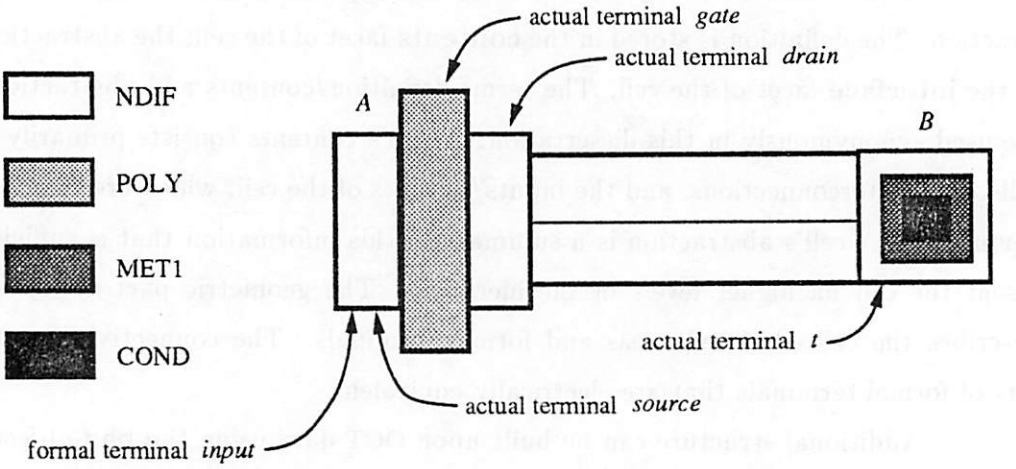


Figure 3.12: Symbolic leaf cell `simple:unspaced:contents`.

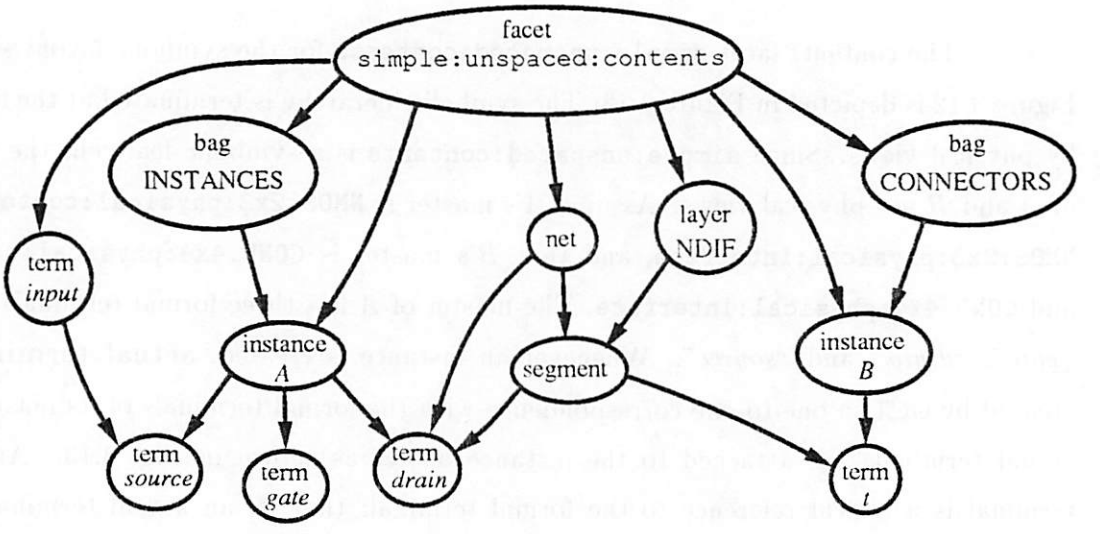


Figure 3.13: Contents facet for the cell in Fig. 3.12.

bolic policy, nets contain segments and the actual terminals of **major instances** (e.g., *A*). All instances other than routing instances are major instances. Nets do not contain actual terminals of routing instances (e.g., *B*); this allows an application to discriminate between "significant" actual terminals and those used only in the implementation of wiring. A net-listing program, for example, has no use for the routing terminals. Bag objects are used to differentiate between major instances and routing instances; all major instances are attached to a bag named INSTANCES, and all routing instances are attached to a bag named CONNECTORS. These bags illustrate the use of the OCT bag object as a mechanism for adding structure to the data model. Each segment contains the two actual terminals it connects. Segments are associated with layers by attaching them to layer objects. This simple, uniform connectivity model results in part from the decomposition of all wiring into sets of two-point segments.

Formal terminals at the current level are created by promoting actual terminals. In this example, there is one formal terminal, called "*input*". This terminal results from promoting the actual terminal "*drain*" of instance *A*. A terminal object is created and attached as shown in Fig. 3.13 to indicate this. The geometric model of "*input*" is part of the interface facet of the master of *A*.

Interface Facets

The symbolic cell `simple:unspaced:contents` must be abstracted, geometrically and logically, for use on the next level in the hierarchy.[1] This abstraction is stored in the interface facet of the cell, namely `simple:unspaced:interface`, shown in Figure 3.14.[2] The interface has the same formal terminal as the contents. The formal terminal is the only connectable part of `simple:unspaced:interface`; the remainder is blocked.

The OCT representation of the interface facet is shown in Figure 3.15. Interface facets do not contain hierarchy, as they are a summary of the hierarchy beneath them. The geometric implementation of a formal terminal is attached to the formal-terminal object as shown in the figure. Each of these geometries is called a **terminal frame**. The geometric objects that implement blockages are called **protection frames**. As in the case of segments, the geometric objects are assigned to layers by attaching them to layer objects.

[1] In a real example, the cell would probably be compacted first.

[2] The abstraction is created automatically, and its level of geometric detail can be varied as well. The manner in which this is done is described in the next chapter.

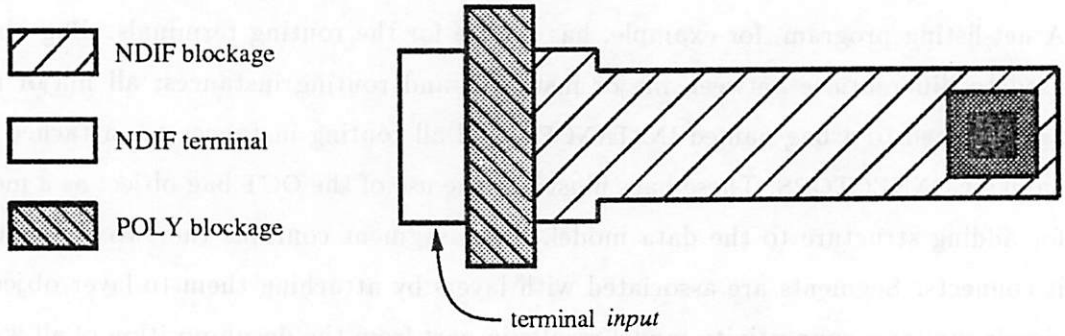


Figure 3.14: Abstraction of the cell from Fig. 3.12.

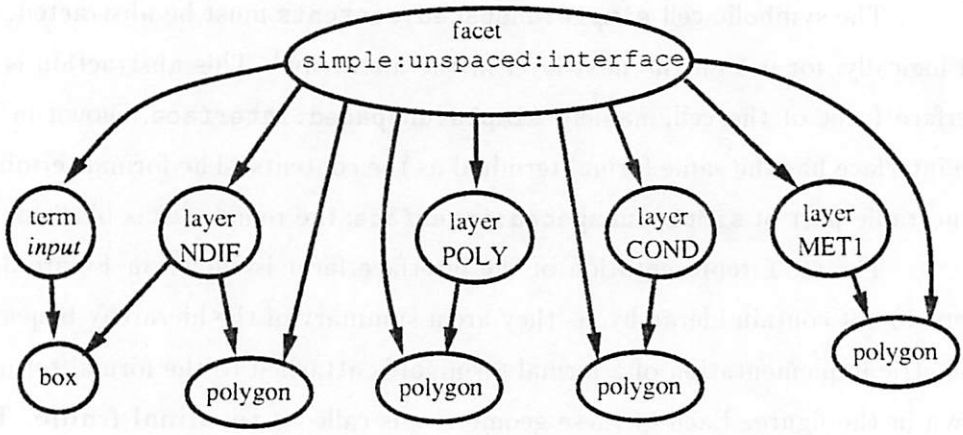


Figure 3.15: Interface facet for the cell in Fig. 3.14.

3.5.3 Design Flow Example

To illustrate how the BLM can be used, a bottom-up hierarchical compaction by the SPARCS program is described in this subsection.

Consider the symbolic view **unspaced** of cell C at the current level. Its contents facet `C:unspaced:contents` is created by placing instances and connecting them with segments. Initially SPARCS opens the contents facet of C and reads each of its subcells S_j (its instances) in turn. For each instance, its interface facet is opened and its protection frames and terminal frames are read. The segments from `C:unspaced:contents` are then read, at which point all necessary information about the input layout is known.

The data from the interfaces of the subcells, plus the segment data, is used to generate the constraint graph for C. The constraint graph is then solved, and the solution data is used to translate the subcells to their new locations and to grow/shrink the wire segments. Typically the output is stored in a different view, for example `C:spaced:contents`. The output is another symbolic view, not a physical view.

The interface representation `C:spaced:interface` is then computed. This interface representation can be used directly as a subcell on higher levels in the hierarchy.

3.5.4 Summary of Model Characteristics

The BLM data model satisfies the pertinent requirements listed in Section 3.2. The model is generic, hierarchy-level independent, and technology independent. That is, OCT does not have specific types for MOS transistors, bipolar transistors, capacitors, etc. The data model does not discriminate between the leaf level and higher levels. The data model is not specific to any particular technology.

The level of abstraction of the data model is a *superset* of the mask and schematic levels. Because the layout elements are sized, a mask-level representation can be generated from a symbolic representation by simply discarding the connectivity information. If the traditional layout model is used as the compactor's output form, then the output must undergo a symbolic-translation step to "flesh out" the symbols into their detailed geometric equivalents. If the output form is a mask-level description, then the compacted cells cannot be further processed by the symbolic system. The data model is a superset of the schematic level, because all schematic-level information can be represented along with the symbolic data. The extra information present in the BLM data model makes design verification

easier. For example, many verification processes require the detailed connectivity of the layout; this information is present in the model.

Many of the advantages of the BLM follow from the cell model, which is described in detail in the next chapter.

Chapter 4

Cell Model

4.1 Introduction

The layout model proposed in this project is type-free, i.e., it is comprised of geometric and connectivity data only, for the reasons given previously. This chapter presents the cell-model portion of the layout model in detail.

The cell model uses **protection frames** and **terminal frames** for blockages and connection areas, respectively. The basic protection-frame/terminal-frame idea is not new [5,37]. However, the model, and the algorithm for creating the model from a compacted cell, have been completely re-formulated in this project. The new model leads to substantially denser layouts than the previous one. In addition, it has been designed to complement a new, efficient constraint-generation algorithm which is described in the following chapter.

The next section introduces the notion of using edge types rather than element types in cell modeling. Several options for representing blockages and terminals are then described. The basic configuration of the BLM cell model is presented next, first for a single mask layer then for multiple layers. This is followed by a description of the advanced features of the model. The chapter concludes with a summary of the model and a comparison with its predecessor.

4.2 Edge Types

As noted in the last chapter, SPARCS avoids the use of element-type information to broaden its range of applicability. To do so, the layout must be described at a lower level.

using geometric and connectivity data alone. To describe the layout at this level, the cell model is defined in terms of two classes of shapes, namely *blockage areas* and *connection areas*. These basic shapes are arranged in specific ways with respect to each other; for example, a blockage on layer L_i can abut a connection area on L_i, but they cannot overlap; also, a blockage that abuts a connection area is interpreted differently than one that is near a connection area but not touching it.

To achieve high density, the basic mechanism used in this abstraction is to assign types to *edges* rather than to components. In a sense the edge types provide the information that other systems encode as element types. The edge-typing mechanism is strictly geometric in nature; hence it is technology and hierarchy-level independent. A simple set of rules describes the relationships between the various types of edges. For example, one type of edge always results in the generation of constraints to other edges, regardless of their types, when the other edges are from different elements and on layers with which there is a spacing requirement. Also, some edge types behave in a conditional manner as a function of connectivity data.

4.3 Blockages and Terminals

Consider the spaced NAND gate shown in Figure 4.1 whose contents facet is `nand:spaced:contents`. A mechanism is needed to construct a geometric abstraction of this cell, which will be used to represent it on higher levels in the hierarchy. The same mechanism must also handle the modeling of the subcells of `nand:spaced:contents`; that is, the transistors and contacts.

The abstraction (the cell model) must discriminate between blocked areas and connection (terminal) areas. In Fig. 4.1, only a few areas of the cell are connectable on the next level. The remainder of the cell must be modeled by blockages, so that other elements on the next hierarchy level can be placed to avoid these areas. The masters of the instances in Figure 4.1 require a similar model. The MOSFET sources and drains are connectable areas on the diffusion layer. The gate extensions are connectable areas on the polysilicon layer, while the active area is blocked. Contacts do not contain blockages; they are comprised of terminal areas only.

There is tremendous latitude in the design of a blockage and terminal model. The next two subsections enumerate some of the options, and present the ones selected for the

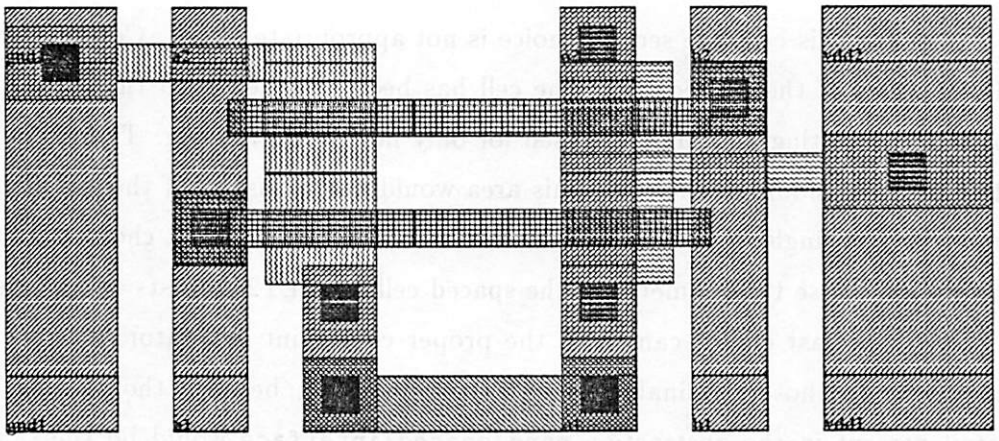


Figure 4.1: Spaced NAND gate `nand:spaced:contents`.

BLM cell model.

4.3.1 Blockages - Modeling Options

The following list contains several options for modeling the blocked part of a cell, in order of increasing accuracy.

1. Use a single bounding-rectangle blockage that encloses all geometries.
2. Use per-layer bounding-rectangle shapes.
3. Use per-layer bounding-polygon shapes.
4. Use the actual geometries.

The CPU time needed for constraint generation depends on the total number of edges in the layout description. Note that the number of edges, and hence the processing time needed for constraint generation, increases from choice one to choice four.

The first choice is often too crude to be useful, especially when the ratio of the largest spacing rule to the smallest spacing rule is large. To ensure a correct layout, adjacent cells modeled in this fashion must be separated by the maximum design rule for the technology. In Mosis CMOS this ratio is 9 [2]; the largest rule is 9λ, which is the well-to-well spacing when the wells are at different potentials. For some orientations, adjacent instances of `nand:spaced` can actually *overlap* by as much as 9λ. Hence the first choice could unnecessarily increase the pitch of a cell composed of two instances of `nand:spaced` by up to 18λ.

For this cell, the second choice is not appropriate either. Consider the MET1 and MET2 layers of the spaced cell. The cell has been designed such that MET1 is used only for vertical routing and MET2 is used for only horizontal routing. There is a free track on MET1 in the middle of the cell. This area would not be usable if the MET1 blockage were modeled by a single bounding-box. However, for modeling MET2, choice two is adequate in this case because the geometry in the spaced cell on MET2 consists of a single rectangle.

The last choice can, with the proper constraint generator, produce areas nearly equivalent to those obtainable using flat compaction, because the amount of geometric detail present in the abstraction `nand:spaced:interface` would be the same as that in `nand:spaced:contents`. However, the amount of CPU time needed for constraint generation would also be about the same as that needed for flat compaction, so such an abstraction is pointless.

The third choice is the most appropriate, and it has been selected for the BLM. However, there is not a unique set of polygons for most cells, including the current one. One possible choice for `nand:spaced` is shown in Figure 4.2. The blockage for the MET1 layer is shown in Fig. 4.3; note that two polygons are used, and that small gaps between features have been filled. Using this abstraction, adjacent instances of `nand:spaced` can be compacted together such that there is no loss in area efficiency compared to an abstraction using all of the geometry. In addition, the number of edges in the abstraction in Fig. 4.2 is significantly less than the number in `nand:spaced:contents`, leading to CPU-efficient compaction on higher levels. In the BLM, a blockage is referred to as a protection frame.

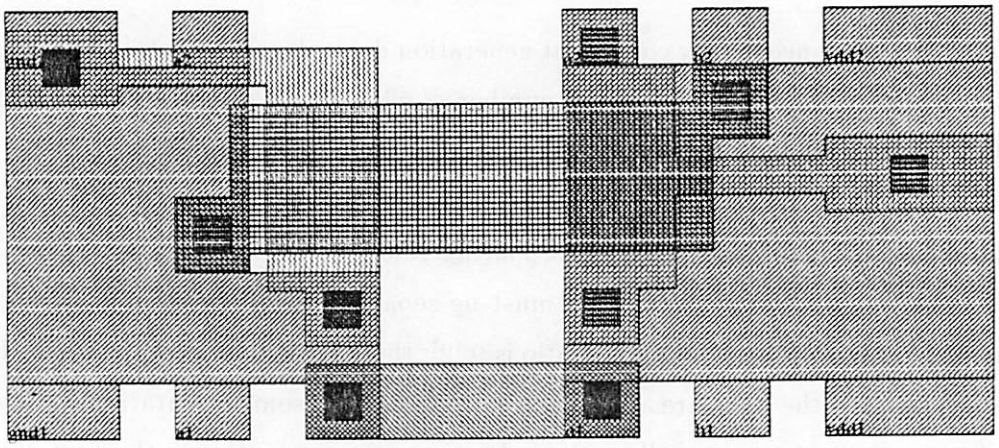


Figure 4.2: Abstraction `nand:spaced:interface` for `nand:spaced:contents`.

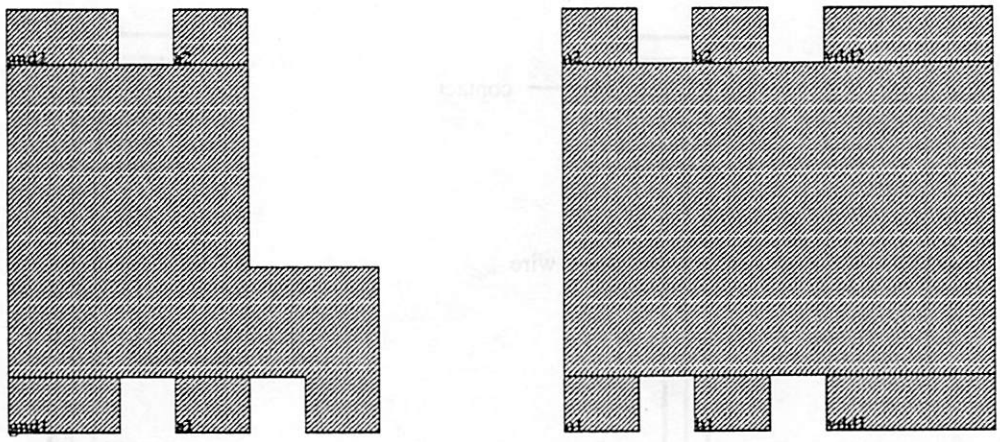


Figure 4.3: MET1 frames from Figure 4.2.

4.3.2 Terminals - Modeling Options

The connection areas of a cell can also be modeled in several ways. Three possibilities are:

1. Use per-layer points.
2. Use per-layer rectangular shapes.
3. Use per-layer polygonal shapes.

Early systems allowed for connections at points only. The connection point modeled the center of the actual connectable region. This model may waste area whenever the terminal area is wider than the wire segment that connects to it. This is a common situation; for example, contacts between MET1 and other layers are usually larger than the minimum widths of MET1 and the other wiring layers. If the wires are allowed to slide along the terminal areas instead, smaller layouts can result as shown in Figure 4.4.

A more powerful model is to use rectangular areas for terminals, *and* allow terminals in the same electrical net and on compatible layers to merge. Note that one dimension of a terminal area describes the interval along which a wire can slide, while the other describes the depth of the mergeable region.

The third option listed above is more general still. However, it is not clear whether it is worth the effort required to implement it. Polygonal terminals cannot make much difference in area over rectangular terminals unless segments are allowed to slide around

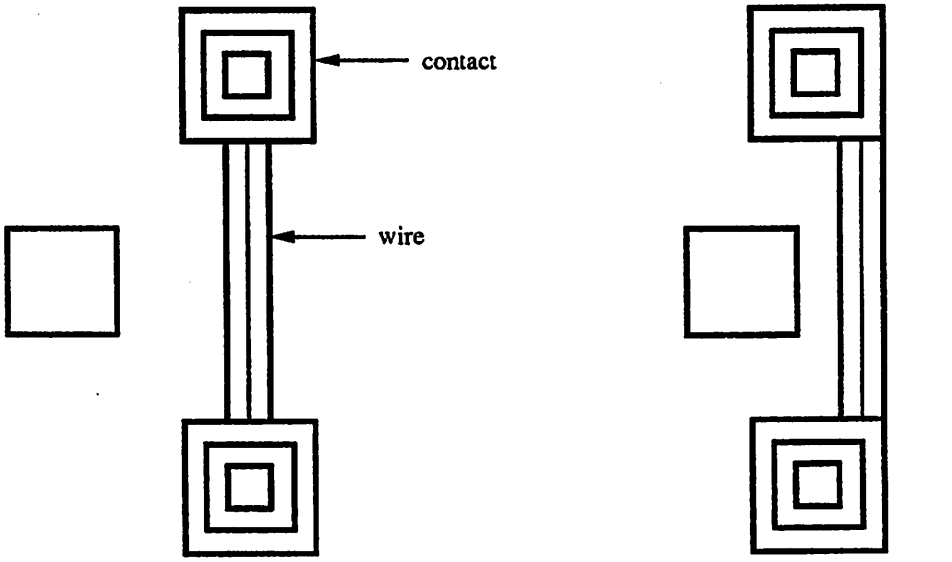


Figure 4.4: Area savings due to sliding terminals.

corners, as shown in Figure 4.5. This implies that the direction of the segment would change from horizontal to vertical, or vice-versa. Such a change is a topological change which is outside the realm of the usual compaction move set. Polygonal terminals have not been implemented, since they would complicate the system and since it appears that they would seldom make a significant area difference. The second option, i.e., per-layer rectangular areas, has been selected as the BLM terminal model instead. The rectangle that represents the terminal area on a given layer is called a terminal frame.

4.4 Basic Single-Layer Model

The BLM cell model will be described by first considering cases in which only a single layer is present. The multi-layer case will be described after the single-layer case.

4.4.1 Protection Frames

To correctly model a cell, the protection frame(s) on a particular layer must cover all unconnectable material on that layer. The frame(s) may, and is likely to, cover *more* area than that, to simplify the representation of the cell on higher levels. The precise

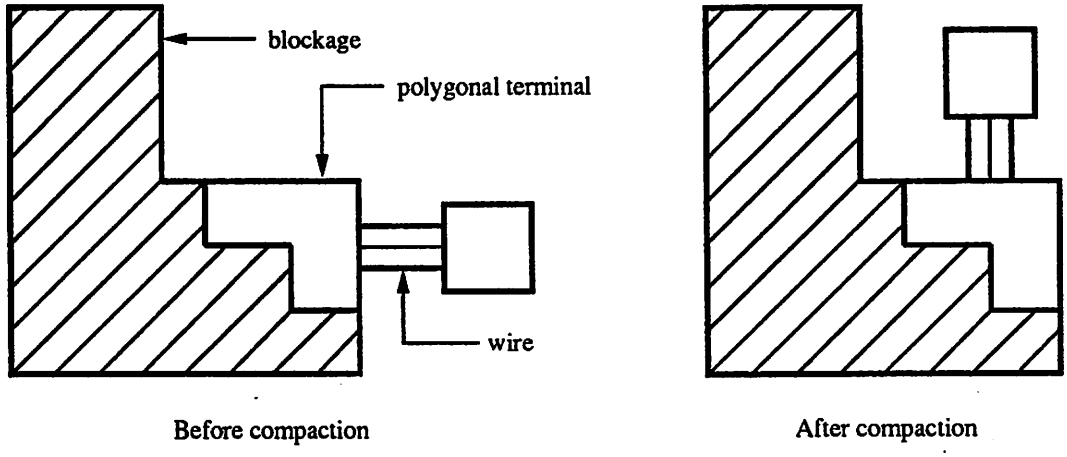


Figure 4.5: Polygonal terminal area.

definition of a protection frame has a large effect on the layout area. In fact, for maximum area-efficiency, protection frames must also play a role in modeling the connectivity of the cells they represent. This consideration is described in detail later in this chapter.

For the purpose of illustration, a definition for a crude set of protection frames is given here. The definition that is actually used in the BLM will be presented later.

Definition 1 *Crude Protection-Frame Set*
A set of Manhattan polygons on a particular layer, which is guaranteed to cover all blockages on that layer. That is, the protection frames cover everything on the layer except for the connection areas.

For the geometries in Figure 4.6, a number of protection frames are valid. Some examples of valid frames are given in Figure 4.7, Figure 4.8, and Figure 4.9. The frames in Figure 4.10 are not valid, because they do not cover all of the geometry they model.

Compaction constraints are generated by comparing the edges of neighboring shapes. For example, if two instances of the abstraction from Figure 4.8 are placed as shown in Figure 4.11, the spacing constraints needed to compact them in x are found by comparing the right-hand edges of the shape on the left to the left-hand edges of the shape on the right. To facilitate this process, edges are assigned **types** in this cell model, as mentioned earlier. Edges are also assigned **polarities**: right-hand edges and bottom edges have polarity `min`, while left-hand and top edges have polarity `max`. Edge polarities are indicated in Figure 4.11.

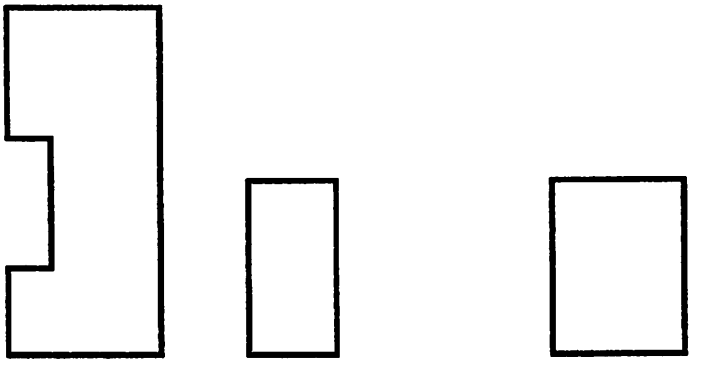


Figure 4.6: Example geometries to be modeled by protection frames.

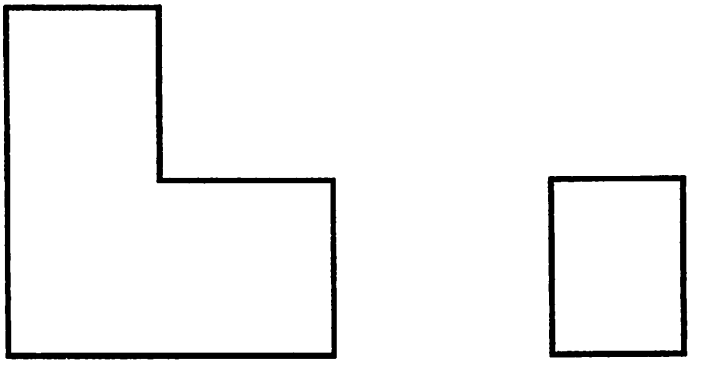


Figure 4.7: Possible protection frames for geometries in Figure 4.6.

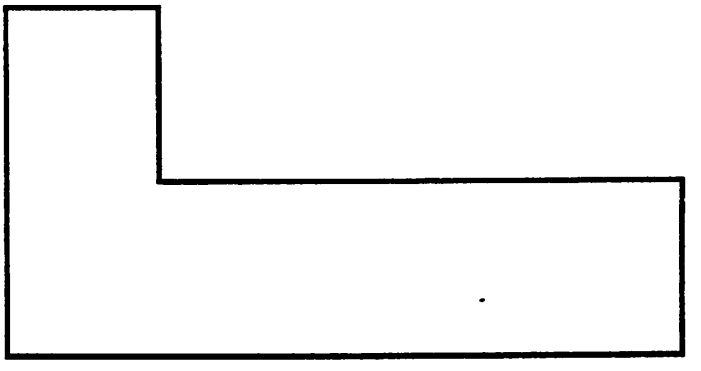


Figure 4.8: Possible protection frames for geometries in Figure 4.6.

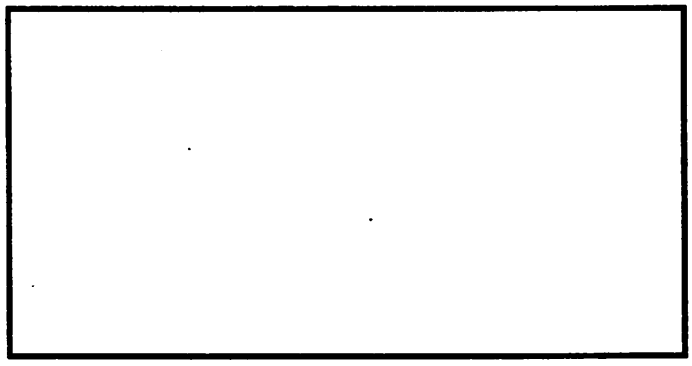


Figure 4.9: Possible protection frames for geometries in Figure 4.6.

Figure 4.10: Illegal protection frames for geometries in Figure 4.6.

Edges from protection frames that *do not abut edges from terminal frames on the same layer* are denoted as type `pf`. In Figure 4.12, edges a, b, and c are of type `pf`, but edge d is not. The type and behavior of edges like d, as well as the other edges in Figure 4.12, are described below. Since they model blockages, a `pf` edge E_{ki} from element k on layer i generates a constraint of value S_{ij} to any other opposite-polarity edge E_{lj} from element l on layer j, where S_{ij} is the spacing rule between layers i and j, *regardless* of the type of E_{lj}. In other words, blockages (as defined thus far) must always be spaced away from any other shapes by the appropriate spacing rule, irrespective of whether the other shapes represent blockages, wires, or terminals. The spacing rule between two blockages is synthesized by comparing the `max` edges of the lower blockage to the `min` edges of the upper blockage; e.g., the right-hand edges of the left-most blockage are compared with the left-hand edges of the right-most blockage (Figure 4.11).

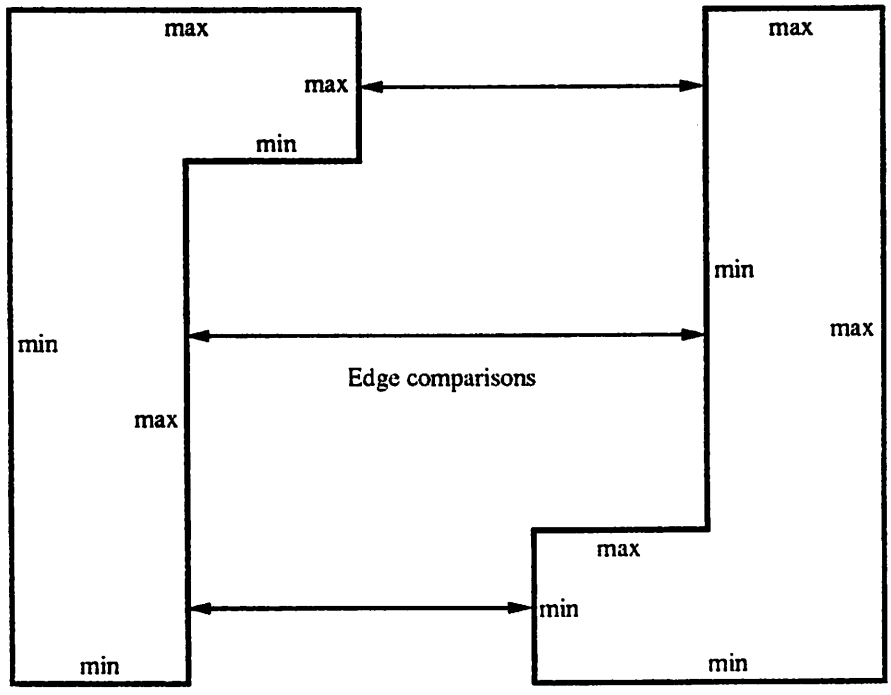


Figure 4.11: Edge comparisons for constraint generation.

The process of creating the (crude) protection frames, which is called **framing**, is given by the following algorithm.

Algorithm F1

1. Read p = set of all blockages on layer L from contents facet.

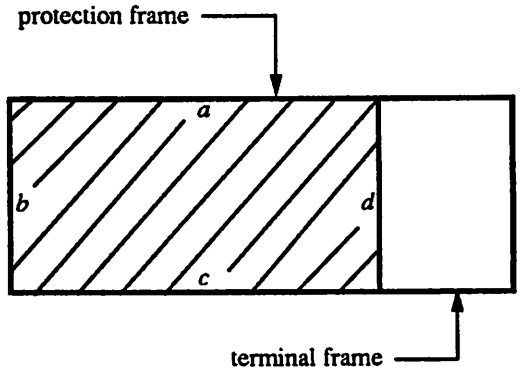


Figure 4.12: Edges of type pf.

2. p_{gms} = grow(p, α). (Grow blockages by $\alpha \geq 0$)

3. p_{gms} = merge(p_{gms}). (Merge any intersections that result)

4. p_{gms} = shrink(p_{gms}, α). (Shrink by the grow amount)

5. Write protection-frame set $P = p_{gms}$ into interface facet.

Algorithm F1 produces an interface facet from a contents facet. The protection frames generated are guaranteed to cover at least all the area covered by the original blockages p, since $\alpha \geq 0$. Varying α controls the granularity of the result: any notches of width up to 2α will fill, and any shapes separated by 2α or less will merge. The protection frames shown in Figure 4.7 correspond to a small α value, while that shown in Figure 4.8 corresponds to an α value of ∞. An α value of zero returns the original blockages p.

4.4.2 Terminal Frames

The BLM terminal frames are defined as follows.

Definition 2 *Terminal frames*
Per-layer rectangles which cover the legal connection areas for other terminal frames or for wire segments that are in the same net and on compatible[1] *layers.*

There are no restrictions on the number of terminal frames or on their locations. However, to connect to a terminal frame at least one of its edges must be accessible. That is, a terminal

[1]Compatibility is defined later. A layer is always compatible with itself.

frame surrounded on all four sides by a protection frame on the same layer (Figure 4.13) is not connectable. In addition, for legal connections, the wire segments must be no wider than the terminals; consequently, the connection shown in Figure 4.14 is illegal.

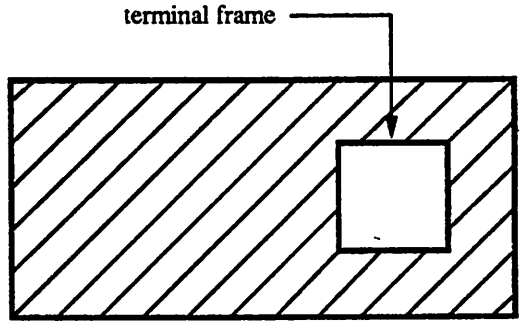


Figure 4.13: Terminal that is not accessible.

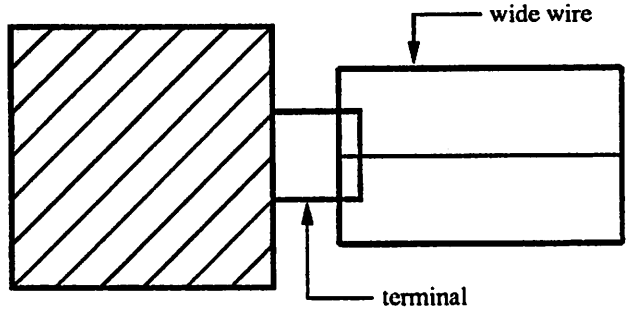


Figure 4.14: Illegal connection.

On a given layer the terminal frames *do not* overlap the protection frames: however, the terminal and protection frames may abut. As a result the union of the protection frames and the terminal frames on a particular layer covers *all* shapes on that layer. Terminal frames on layer L_i may overlap protection frames or terminal frames on other layers.

In addition to indicating connection areas for wiring, terminal frames denote mergeable areas. In Figure 4.15, the master of routing instances A and B is modeled by a terminal frame; no protection frames are present. Since they are mergeable areas on the same layer and in the same electrical net, *any* relative spacing between A and B is legal. Two possible post-compaction configurations are shown in Figure 4.16. Instance A in Figure 4.17 is modeled by a terminal frame abutting a protection frame on the same layer. The minimum-area layout for this case is also shown in Figure 4.17, and again

electrically-connected areas on the same layer have merged.

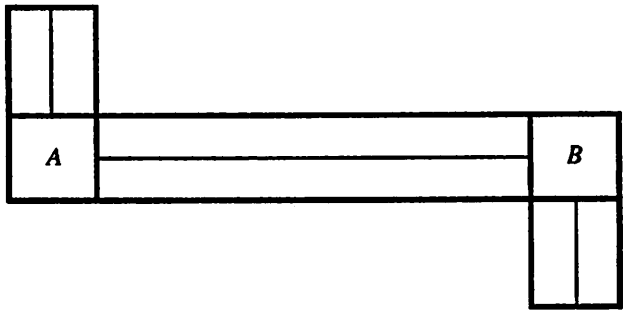


Figure 4.15: Routing instances and segments.

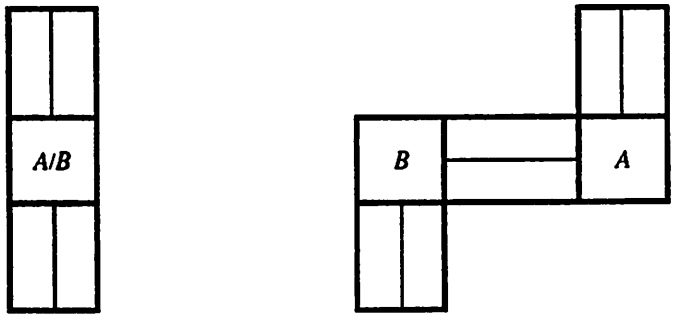


Figure 4.16: Two possible compaction results for the example in Figure 4.15.

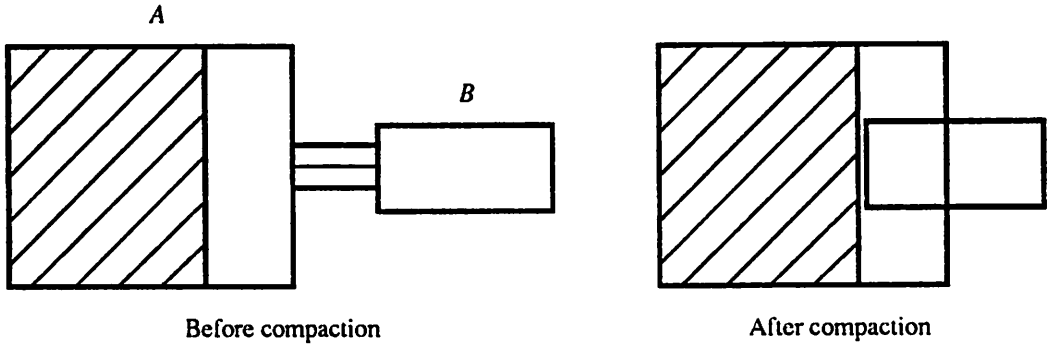


Figure 4.17: Merging example.

Terminal-frame edges must be treated differently than protection-frame edges to achieve merging. Protection frames are never mergeable, and edges of type `pf` always generate constraints to other edges, regardless of type. On the other hand, the behavior of a terminal-frame edge differs depending on whether the edge it is being compared to is

from the same net or not. Terminal frames can merge if they are on the same layer[2] and in the same net; if the nets differ, then they behave like protection frames. Figure 4.18 depicts two pairs of connected terminal frames; all shapes are on the same layer. In the horizontal direction terminal frames A and B are mergeable, and C and D are mergeable, but A (and B) cannot merge with either C or D since they belong to different nets. Accordingly, edges a_2 and b_2 behave like `pf` edges with respect to edges c_1 and d_1, even though they are terminal-frame edges. The horizontal constraint graph that results is shown in Figure 4.19.

Figure 4.18: Two pairs of connected terminals.

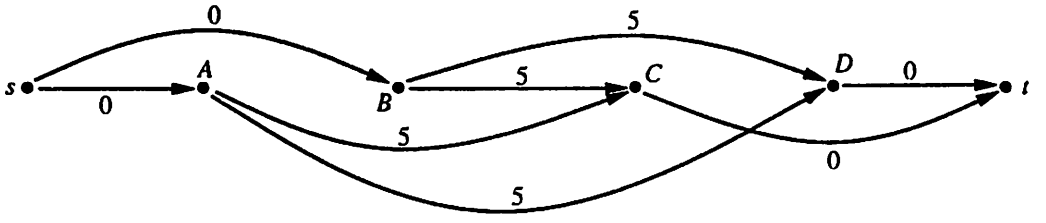


Figure 4.19: Constraint graph for example of Figure 4.18.

In the BLM terminal-frame edges are assigned two different types. If a terminal-frame edge *is not* adjacent to a `pf` edge on the same layer then its type is `tf`. All edges of the routing instances in Figure 4.18, for example, are `tf` edges. Edges a, b, and c'' in Figure 4.20 are `tf` edges. If a terminal-frame edge is adjacent to a same-layer protection frame edge, then its type is `btf`. Edges c' and d in Figure 4.20 are of type `btf`. The `btf` edge-type is described in the next subsection. An edge from a terminal frame that is not connected to anything defaults to type `pf`.

[2]Terminal frames on different layers may sometimes merge as well. This issue is described later in this chapter.

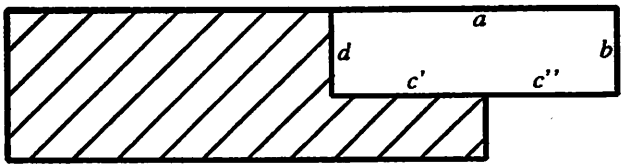


Figure 4.20: Examples of `tf` edges.

A `tf` edge behaves as follows. Like `pf` edges, `tf` edges are usually compared to edges of opposite polarity[3]. If a `tf` edge is compared to an opposite-polarity `tf` edge *from the same net* then no constraint is generated. An example of this is the a_2-b_1 relationship in the example of Figure 4.18. On the other hand, if the two `tf` edges being compared are from different nets then a constraint of value S_{ii} is generated, just as if both edges were `pf` edges. An example of this case is the b_2-c_1 relationship in Figure 4.18.

Algorithm F1 can be readily modified to generate terminal frames as well:

Algorithm F2

1. Invoke Algorithm F1 to create p_{gms} for layer L.
2. Read t = set of all terminal frames of formal terminals on layer L from contents facet.
3. $P = p_{gms} \cap \bar{t}$. (Subtract t from p_{gms} to form the final protection frames)
4. Write protection-frame set P, terminal-frame set $T = t$ into interface facet.

Note that the protection and terminal frames produced by Algorithm F2 may abut, but they do not overlap.

4.4.3 Protection Frames and Terminal Frames Together

Most cells consist of both blocked areas and connectable areas, and hence they are modeled by both terminal frames and protection frames. Generally, such cells are partly mergeable, as opposed to cells containing only terminal frames, which are fully mergeable, and cells containing only protection frames, which are not mergeable at all.

[3]The one exception to this is described later.

Consider the example shown in Figure 4.21. Edges a_3 and b_1 are electrically-connected `tf` edges, hence there is no constraint between them. If the protection-frame edge behind the terminal (edge a_1) is modeled as a `pf` edge, then a constraint of value S_{ii} is added between a_1 and b_1 as shown in Figure 4.22 and the terminal areas do not fully merge. If a_1 is modeled as a `tf` edge, then no constraints are added and the illegal layout of Figure 4.23 results.

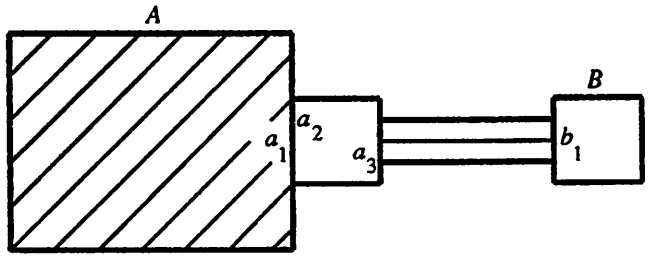


Figure 4.21: Example with protection and terminal frames.

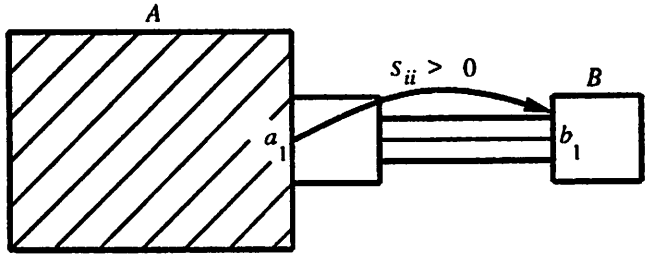


Figure 4.22: Unnecessary constraint that leads to incomplete merging.

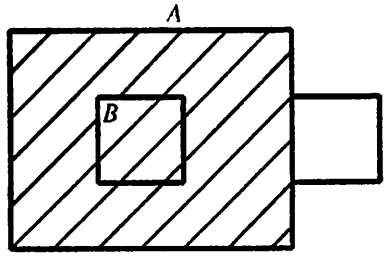


Figure 4.23: Illegal layout due to over-merging.

The correct minimum-area layout for this example is shown in Figure 4.24. Instance B is allowed to merge with A such that edges a_2 and b_1 are coincident. This degree

of merging is legal, because a terminal frame is, by definition, a mergeable area for other terminal frames on the same layer and in the same net. However, edge b_1 *cannot* be any farther to the left than a_2. The amount of merging is limited because there is a blockage in the neighborhood of the terminal. Note that this configuration is achieved if the constraint between A and B is between edges a_2 and b_1 and has value zero, as shown in Figure 4.25.

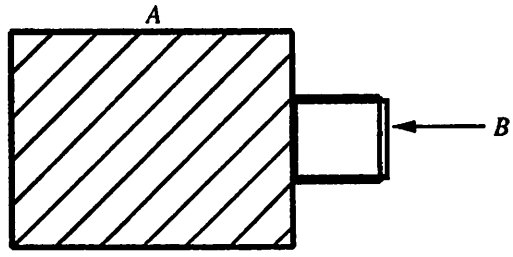


Figure 4.24: Correct minimum-area result for example of Figure 4.21.

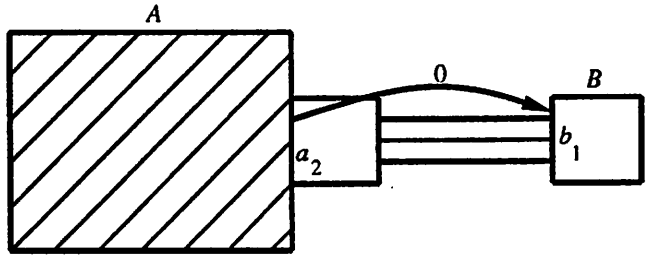


Figure 4.25: Limiting constraint that leads to result of Figure 4.24.

In the cases described in the previous sections, opposite-polarity edges were compared to generate constraints. No relationships between same-polarity edges have arisen. This is because of the fact that, if the starting layout is non-overlapping, then an opposite-polarity edge will always be encountered before a same-polarity edge from the same shape (Figure 4.26). Because layout features have positive size, opposite-polarity relations like p_1-q_1 render same-polarity relations like p_1-q_2 redundant for such cases.

In cases like that depicted in Figure 4.21, however, the relevant edges (a_2 and b_1) have the *same* polarity. The rule value *is not* S_{ii}, the layer i to layer i spacing rule; rather, the rule value is zero. Furthermore, a constraint relation has been suppressed (edge a_1 to edge b_1). This illustrates the basic differences between a merging situation and a non-merging one; namely, *different edges are compared and different constraint values are*

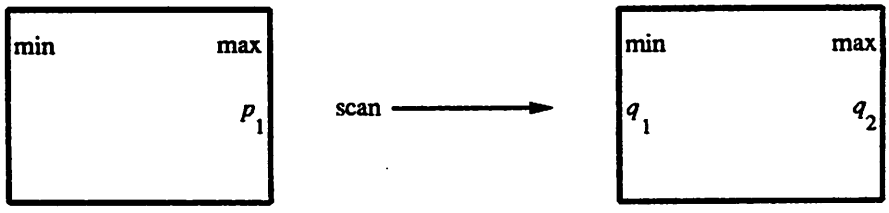


Figure 4.26: Opposite-polarity edge is encountered before same-polarity edge.

used. This means that the layout model must facilitate the conditional comparison of edges, as a function of connectivity information.

The behavior described in this subsection is achieved through the use of a third edge type. That type is denoted the `btf` type, which is a mnemonic for the **b**ack edge of a **t**erminal **f**rame. A terminal-frame edge that abuts a protection-frame edge on the same layer is defined to be a `btf` edge. In the example given in Figure 4.21, a_2 is a `btf` edge.

Since the `btf` edge delimits the mergeable region, the protection-frame edge at the same location can be suppressed (e.g., a_1). The suppression is done by marking the protection-frame edge as invisible over the interval that the protection frame and `btf` edges are coincident. This marking operation will be referred to as **soft masking.**[4] Masking makes intuitive sense; if there are two edges from the same element on the same layer at the same location, then one of them must be redundant. It should be noted that terminal-frame edges *are not* assigned the `btf` type and and protection-frame edges *are not* soft-masked at the cell-model level; instead, the edges are so indicated dynamically during constraint generation.

4.5 Basic Multiple-Layer Model

This section describes the basic behavior of the BLM for multiple-layer cells.

4.5.1 Protection Frames

No new concepts are needed to model blocked areas on multiple layers. A distinct set of protection frames is used for each layer. Edge types are assigned according to the considerations described above for single-layer problems. That is, protection-frame edges

[4]The modifier "soft" is used to distinguish this form of masking from another form which is described later in this chapter. This masking is termed soft because one of the edges remains visible.

that do not abut terminal-frame edges on the same layer are assigned the `pf` type. A protection-frame edge that does abut a terminal-frame edge on the same layer is masked (made invisible). There are no restrictions on the edge relationships across layers; i.e., within the model of a particular cell, protection frames on layer L_i can abut or overlap protection frames on any other layer L_j arbitrarily.

4.5.2 Terminal Frames and Compatible Layers

Terminal areas are often connectable on more than one layer. In the BLM, the model of such an area consists of a terminal frame on each layer present in the logical definition of the terminal. For example, a POLY-MET1 contact in a CMOS process can be connected to by either a POLY wire or a MET1 wire. In this case, the cell is modeled by a terminal frame on POLY, a terminal frame on MET1, plus an additional terminal frame on the contact-cut layer COPO for each contact-cut present. The contact shown in Figure 4.27 therefore has four terminal frames and no protection frames. The contact-cut geometries are modeled as terminal frames, even though wiring is not allowed on the COPO layer, because they are mergeable areas.

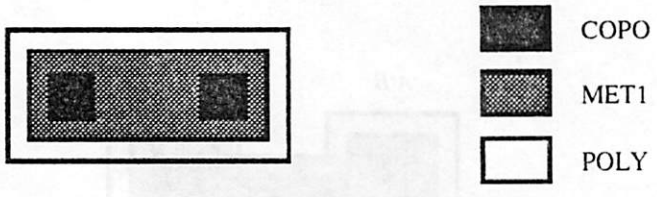


Figure 4.27: Terminal frames of POLY-MET1 contact.

Terminal frames that are in the same net and on the same layer are always mergeable. Same-net terminal frames that are on *different* layers may or may not be mergeable. Assume that there is a rule S_{ij} between layers L_i and L_j when they are not in the same net. If those layers *are* allowed to merge when they are in the same net, then they are defined as **compatible**. For example, a terminal frame on COND (the contact-cut layer for NDIF-MET1 contacts) can merge with a terminal frame on NDIF when they are in the same net. However, if their nets differ they must be separated by 5λ under the Mosis rules [2]. Hence NDIF and COND are compatible. On the other hand, a terminal frame on POLY can never merge with one on NDIF, so POLY and NDIF are not compatible. A layer is always compatible with itself. When a spacing requirement exists between two layers, the layers

(or edges on them) will be referred to as **related**. Note that the compatibility notion is not required between layers that are not related, such as POLY and MET2. The compatibility information is determined by analyzing the design rules for the target technology.

If two opposite-polarity `tf` edges from the same net are being compared, and if they are compatible, then no constraint is generated. If they are not compatible, then the edges behave with respect to each other as if they were `pf` edges and a constraint is generated. In the example shown in Figure 4.28 all instances are electrically connected and all consist of only `tf` edges. The first two instances (A and B) are comprised of compatible layers and hence they merge as shown in the spaced result. However, NDIF and COND are both incompatible with POLY and COPO, hence C does not merge with A or B.

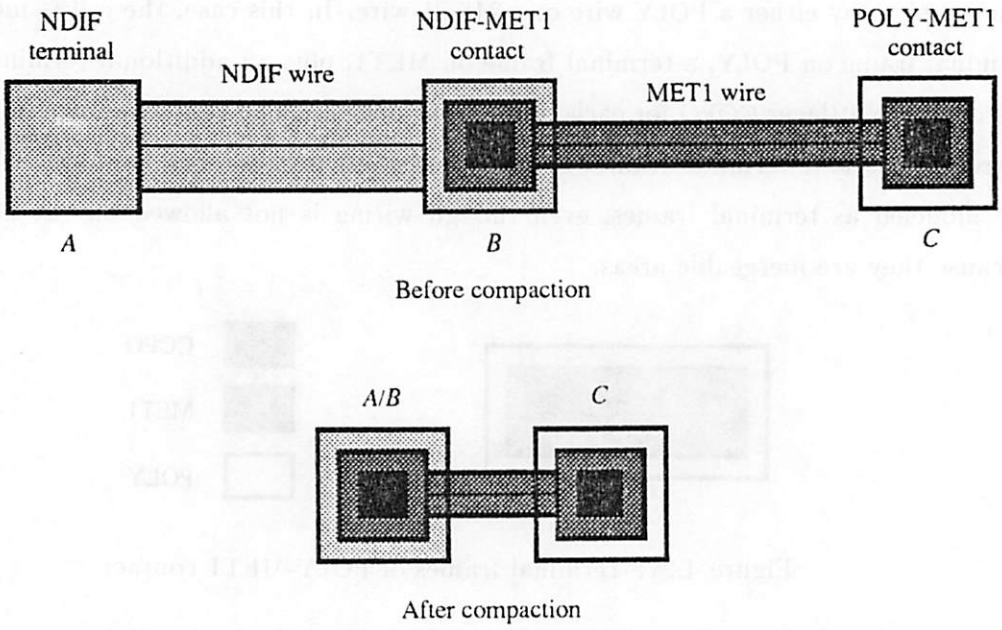


Figure 4.28: Illustration of compatible layers.

4.5.3 Protection Frames and Terminal Frames Together

If a protection frame abuts a terminal frame on the same layer, the protection-frame edge is masked and the terminal-frame edge is assigned type `btf`, just as in the single-layer case.

4.6 Advanced Features of Cell Model

The preceding sections of this chapter describe the basic architecture of the cell model. However, several other mechanisms must be incorporated into the model to achieve maximum-density layouts. These features are described in this section.

4.6.1 Adjacent Protection Frames and Terminal Frames

The areas of cells that are adjacent to terminals must be carefully modeled to insure that elements are allowed to merge as much as possible. In the BLM, as described to this point, it has been implicitly assumed that any protection frame that abuts a terminal frame originated from a geometry that was electrically connected to the terminal area. This assumption was used to suppress (i.e., soft-mask) the protection-frame edges that touch `btf` edges. Algorithm F2 *does not* guarantee that abutting terminal-frame and protection-frame edges originate from electrically-equivalent shapes. In this subsection, the need for guaranteeing that this condition is satisfied is described. A new framing algorithm will be given below, which does maintain the desired condition.

Under Algorithm F2, there are two ways that a protection-frame edge may come to abut a terminal-frame edge. In one case, the protection-frame edge is from a geometry that is electrically connected to the terminal frame. An example is given in Figure 4.29, where the segment that attaches to the terminals in `case1:spaced:contents` becomes a protection frame in `case1:spaced:interface`. The second way in which the edges may come to abut is due to blockages that are not connected to a terminal, but which grow into the terminal area as a result of Algorithm F2. An example of this circumstance is given in Figure 4.30.

Note that in the second case (Figure 4.30) the *closest* that a different-net blockage can be to a terminal in the contents facet is the S_{ii} rule, assuming that the contents facet (the layout that Algorithm F2 is applied to) satisfies the spacing rules. In other words, the two protection frames in `case2:spaced:contents` must be separated from the terminal frame by at least S_{ii}; otherwise `case2:spaced:contents` is illegal. Hence the adjacency of the frames in `case2:spaced:interface` is due to Algorithm F2 alone.

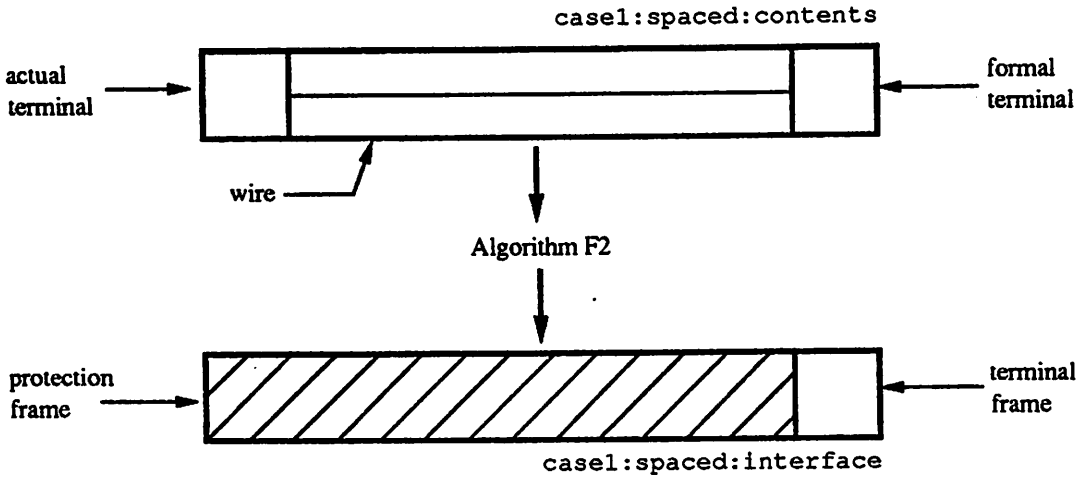


Figure 4.29: First case of abutting frames.

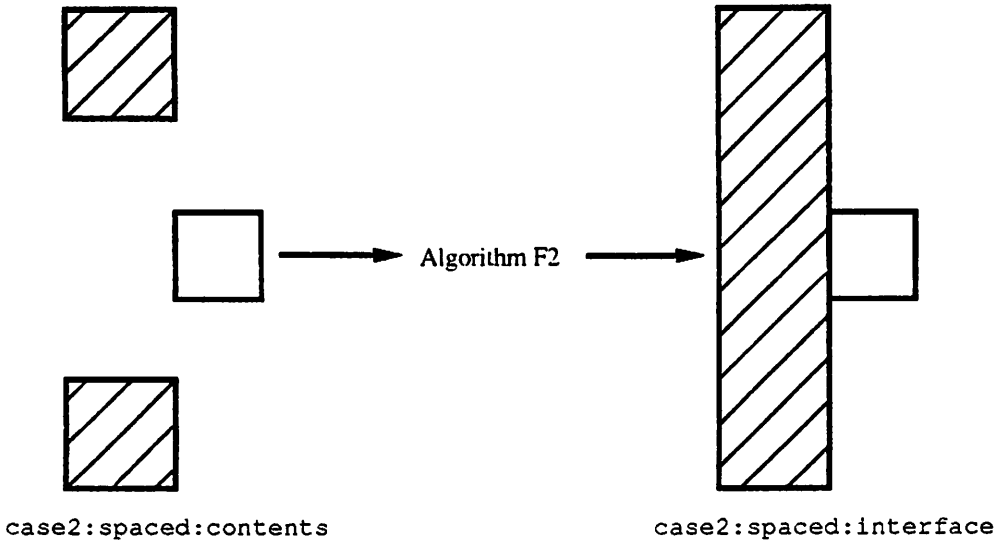


Figure 4.30: Second case of abutting frames.

Problems with Connectivity-Independent Blockage Modeling

Maximal merging is not achievable when cells are abstracted via Algorithm F2, since it cannot be assumed that abutting protection-frame and terminal-frame edges arise from electrically-connected geometries. This problem is illustrated, for example, by the single-layer layout given in Figure 4.31; the master of the large instance is shown in Figure 4.32. The horizontal dimension after compaction is unnecessarily large by S_{ii} plus the width of the small instance, due to the corner constraints shown in Figure 4.31. Note that the fully-merged result would be legal, if the cell were modeled by its definition rather than its abstraction. In addition, a design-rule check cannot be run when the large instance is represented by its interface, since the wire connected to the terminal would be flagged (incorrectly) as an error.

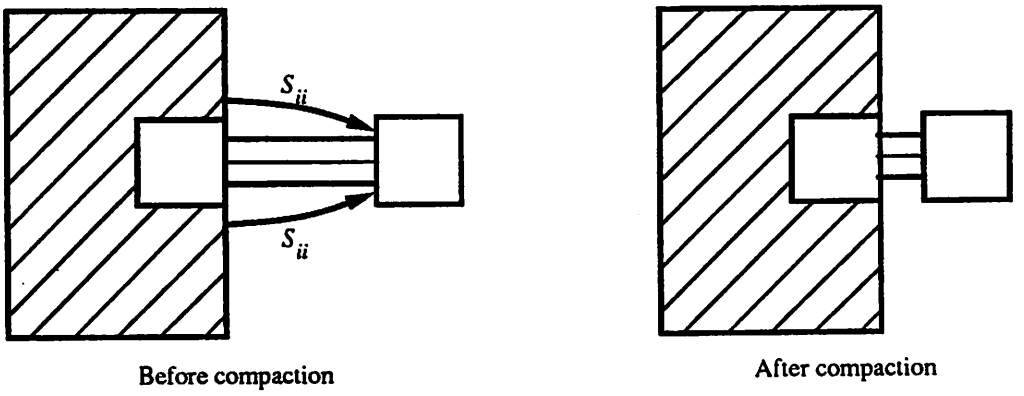


Figure 4.31: Density degradation in single-layer case.

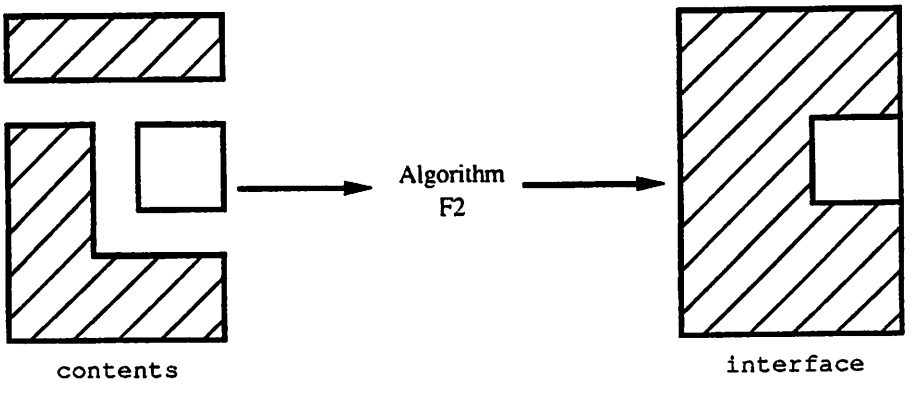


Figure 4.32: Master of large instance in Figure 4.31.

A density penalty results in multi-layer cases as well. Consider the layout shown in Figure 4.33. The Mosis rule for NDIF to COND is 5λ when they are from different nets. Edge a_1 of the left-hand instance could have arisen from an NDIF edge from a *different* net than the terminal, located to the left of the location of a_1 by the NDIF-NDIF rule (3λ), as shown in Figure 4.34. Since this may be the case, edges a_1 and b_2 must *always* be separated by the NDIF-COND rule minus the NDIF-NDIF rule, which is $5\lambda - 3\lambda = 2\lambda$. As shown in Figure 4.33, the final layout would thus be larger than necessary by 1λ if a_1 had actually arisen from a shape in the same net as the adjacent terminal frame.

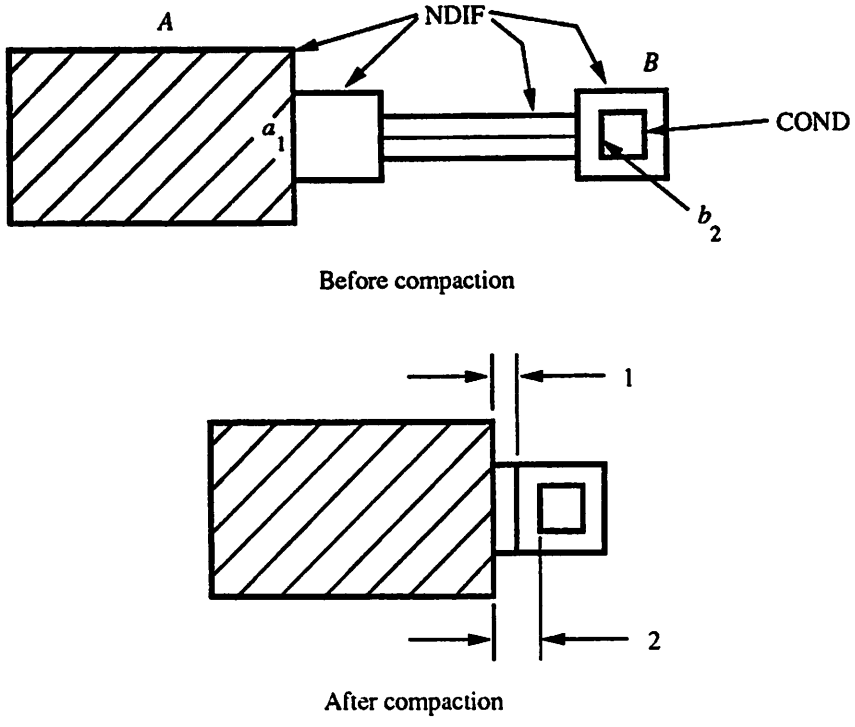


Figure 4.33: Density degradation in multi-layer case.

To solve these problems, the cell model must be extended such that the blockages are modeled in a connectivity-dependent manner.

Connectivity-Dependent Blockage Modeling

It might appear that the above problems could be avoided by always stripping back protection-frame edges from terminal-frame edges by one design rule (S_{ij}) in the framing procedure. For the cell in Figure 4.32, stripping the protection frames back produces

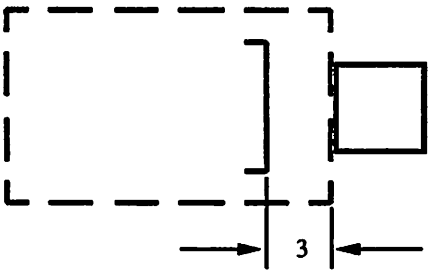


Figure 4.34: Possible location of edge that generated a_1.

the frames shown in Figure 4.35. This mechanism would solve the single-layer problem (Figure 4.31). Unfortunately, it does not eliminate the density problem in the multi-layer case as described above. Furthermore, it can obliterate features as shown in Figure 4.36, which could lead to illegal compaction results on the next hierarchy level.

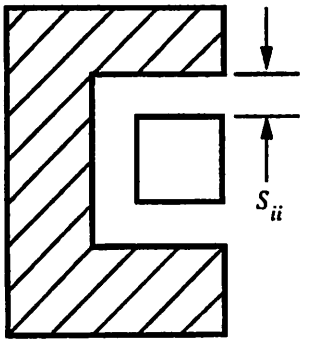


Figure 4.35: Frames for instance in Figure 4.32 stripped back by one rule.

Another possible solution is to explicitly propagate net identifiers from terminal frames to protection frames. This solution is unwieldy because several terminal frames might be coincident with one protection-frame edge. Also, it would increase the time required for constraint generation because protection-frame edges that are coincident with `btf` edges would need to be processed rather than suppressed.

The mechanism for connectivity-dependent blockage modeling that has been implemented in the BLM is both simple and correct. It is specified by the following protection-frame definition, which applies on a per-layer basis.

Definition 3 *Protection-Frame Set*

A set of Manhattan polygons on a particular layer, which is guaranteed to cover all blockages on that layer. The protection frames do not overlap terminal frames on the same layer.

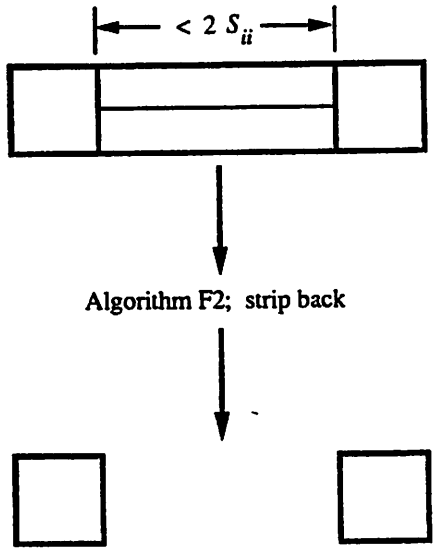


Figure 4.36: Feature obliterated by strip-back.

The protection frames are allowed to abut same-layer terminal frames only if they arise from electrically-connected shapes. The protection frames are separated from same-layer terminal frames by at least the S_{ii} rule if they are not from electrically-connected shapes.

The algorithm which creates these protection frames, and which is actually used in the system, is:

Algorithm F3

1. Read p = set of all blockages on layer L from contents facet.
2. Invoke Algorithm F1 to create p_{gms} for layer L.
3. Read t = set of all terminal frames of formal terminals on layer L from contents facet.
4. t_g = grow(t, S_{ii}). (Grow terminals by S_{ii})
5. $P = p_{gms} \cap \overline{t_g}$. (Subtract t_g from p_{gms} to strip back protection frames from terminals)
6. $P = P \cup p$. (Replace same-net protection frames that originally abutted terminal frames)

7. Write protection-frame set P, terminal-frame set $T = t$ into interface facet.

Algorithm F3 is implemented in the *vulcan* program [44]. The operation of Algorithm F3 is depicted in Figure 4.37. In essence, Algorithm F3 approximates the actual geometry in the region about the terminal more closely than Algorithm F2. Algorithm F3 solves the density problems described above. Since a protection-frame edge is only allowed to abut a terminal-frame edge if they are from the same net, the protection-frame edge can be suppressed as described earlier, and there is no need to propagate net identifiers to protection-frame edges. It is easy to see that P and T cover all of the original geometry; due to Step 6 of Algorithm F3, P covers at least p, and $T = t$ as indicated in Step 7. Hence the strip-back cannot obliterate features.

4.6.2 Elements that Violate Spacing Rules

Some leaf-level layout primitives are seemingly comprised of shapes that are closer together than the rules would ordinarily allow. For example, consider the minimum-length MOSFET in Figure 4.38. The diffusion areas (source and drain) are separated by 2λ, while the diffusion-diffusion spacing rule is 3λ. The polysilicon gate abuts the two diffusions, whereas diffusion and POLY must be separated by 1λ under all other circumstances. It is shown in this subsection how these "violations" of the spacing rules can lead to layouts that are unnecessarily large, if the cell model is not designed with such elements in mind. These potential inefficiencies can occur due to both layer i – layer i (same-layer) and layer i – layer j (cross-layer) rules.

Consider just the NDIF layer in the example in Figure 4.39. The instance on the left (A) is a minimum-length NMOS device. Terminal d is in the same net as terminal t of instance B. As a result, there is no constraint between edges a_6 and b_1. Edge a_2, if treated as a `tf` edge, is compared to `min` edges of shapes to its right. Thus, it is compared to b_1, and since a_2 and b_1 are in different nets a constraint of value S_{ii} (3λ) is added between them. The spaced result will thus be wider than necessary by 1λ, as shown. For maximum density this constraint must be suppressed. Note that this constraint between a_2 and b_1 *would not* degrade the area of the layout if the diffusion terminals of the MOSFET were separated by the minimum NDIF-NDIF spacing of 3λ.

A second undesired constraint arises due to the cross-layer rule between edge a_4

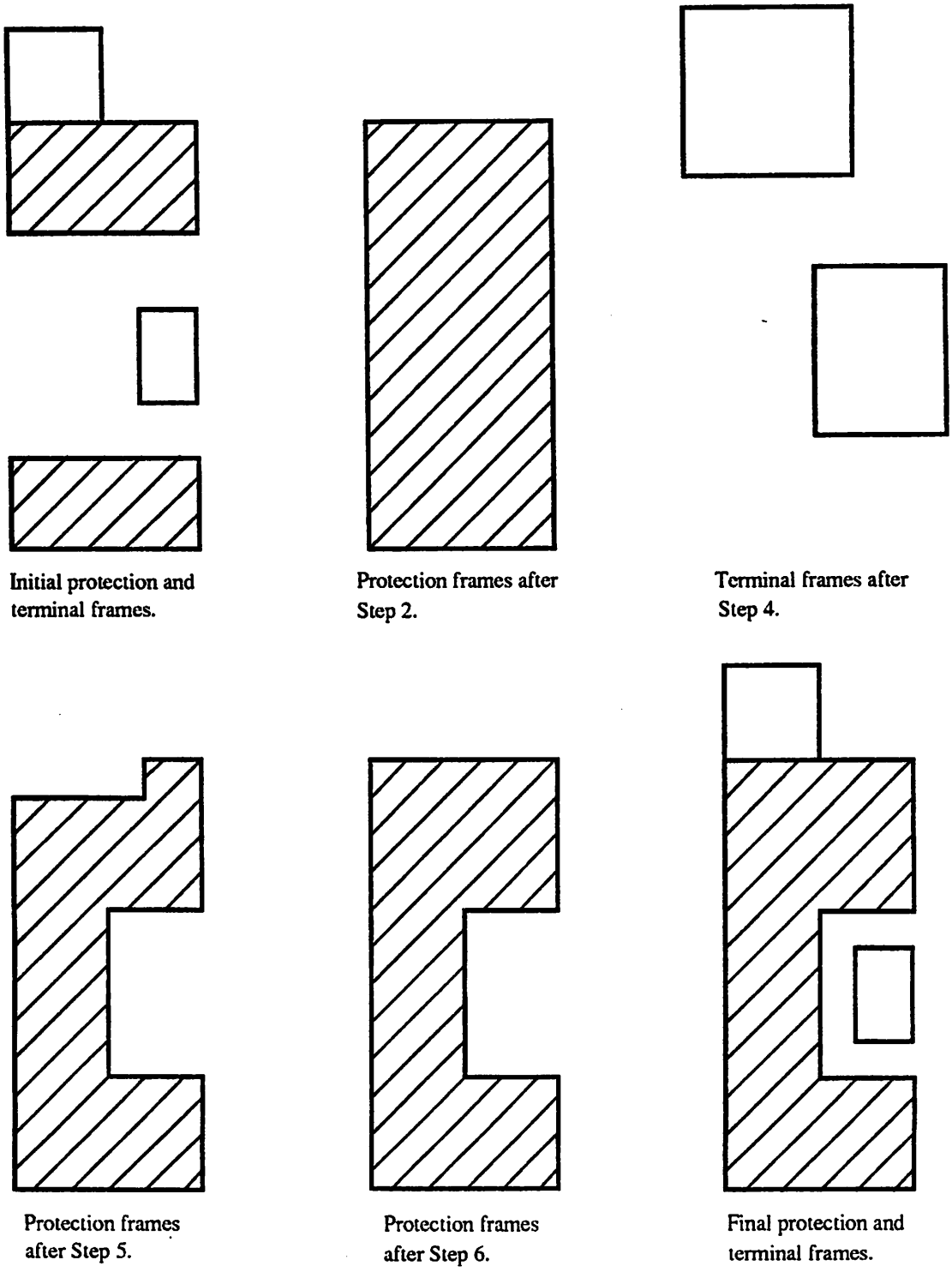


Figure 4.37: Operation of Algorithm F3.

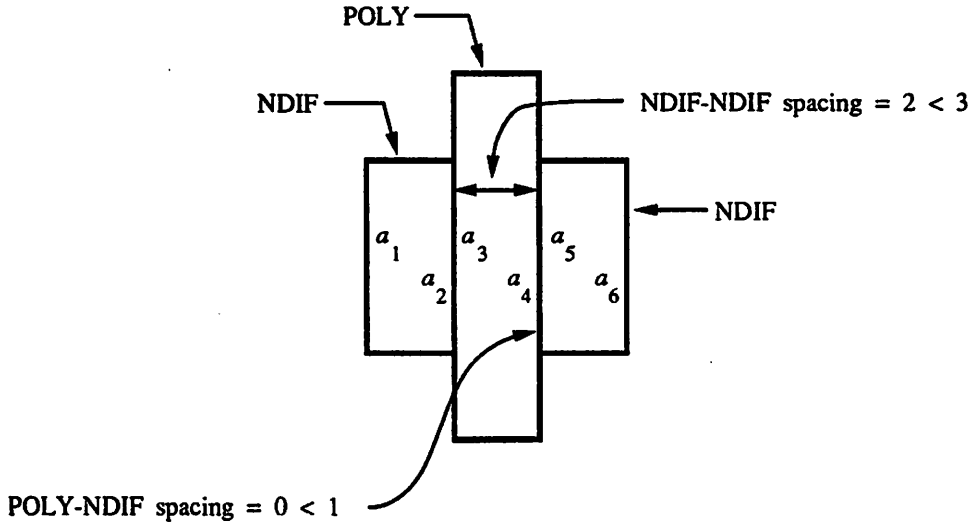


Figure 4.38: Spacing-rule "violations" of a MOSFET.

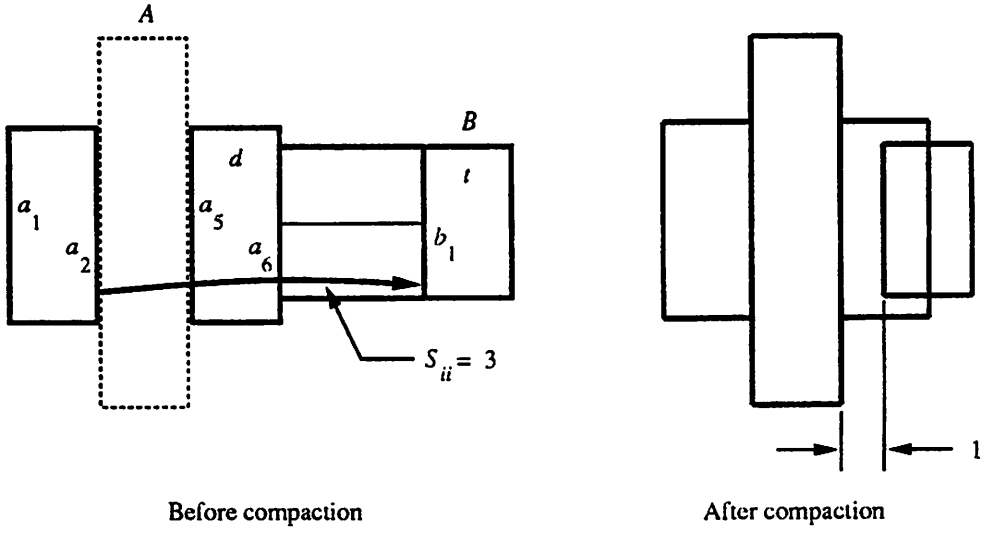


Figure 4.39: Same-layer density degradation.

on POLY and edge b_1 on diffusion. If no special mechanisms are employed, then a constraint will be added between a_4 and b_1 with value equal to the POLY-NDIF spacing (1λ). This constraint also results in a penalty of 1λ, as shown in Figure 4.40. If the MOSFET gate and diffusion terminals were "legally" separated by the POLY-NDIF rule then this constraint would not cause an area penalty either.

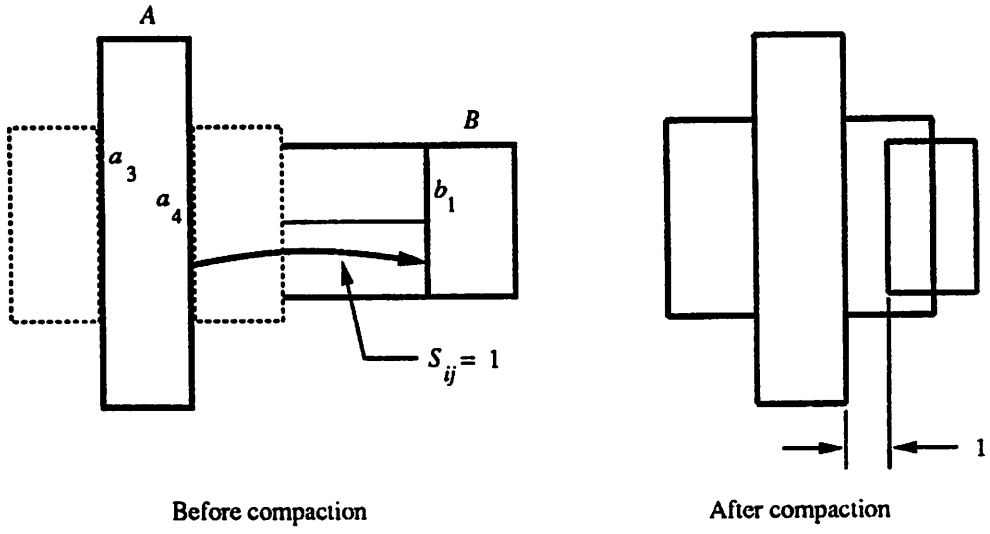


Figure 4.40: Cross-layer density degradation.

A MOSFET is one example of a primitive that is comprised of geometries that are closer together than the rules would otherwise allow. The BLM cell model is able to accommodate such elements without a density degradation, via the extensions described below.

Internal Edges and Notches

Edges such as a_2 and a_5 in Figure 4.39 are internal to the MOSFET, and they should not be processed in the same way as external edges such as a_1, a_3, a_4, and a_6. External contours of cells are compared to generate compaction constraints; e.g., external **max** edges are processed in horizontal constraint generation by searching to the right for external **min** edges of other elements. In Figure 4.39, edge a_2 is a **max** edge, but it is internal to the cell because the region between a_2 and a_5 cannot be occupied by a related shape from another element. Any related shapes to the right of a_2 must lie at least as far to the right as a_5, if they are connected to d, or at least as far as a_6 if they are not connected.

Hence edges a_5 and a_6 determine the necessary constraint relationships with shapes to the right of A. The same-layer density-degradation problem can be solved by recognizing that edges such as a_2 are internal, and thus not subject to the usual processing.

Internal edges typically arise from notches in geometries that are too small to be useful. It is helpful to consider the size of a notch in a shape that is usable. A usable notch is one that is wide enough to fit another shape into. For most layers, a notch is not useful unless another shape on the same layer can be placed there. Using this criterion, the minimum width of a useful notch is

$$n_{min} = 2S_{ii} + w_{min},$$

where w_{min} is the minimum width for features on the layer in question. Any notch smaller than n_{min} may as well be eliminated since it cannot be used. A notch can be eliminated by filling it with a protection frame without any loss in area efficiency.

In a fabricated MOSFET, there is no diffusion in the region between the source and drain terminals, hence this area can be thought of as a notch in the diffusion. For the Mosis rules, $n_{min} = 8\lambda$ for diffusion, which is four times larger than the length of a minimum-length MOSFET. As a result, MOSFETs are modeled in the BLM by filling the region between the source and drain terminals with a protection frame on diffusion. Edges a_2 and a_5 thus become `btf` edges rather than `tf` edges, and the same-layer density degradation does not occur.

Terminal-Frame Masking

The cross-layer problem illustrated in Figure 4.40 arose from a mergeable `tf` edge (b_1) that is mistakenly visible to another edge (a_4) on a different layer. In general, a `tf` edge that is legally merged with a terminal frame of another element *should not be otherwise visible to that element*, provided that it is guaranteed to remain in the mergeable area. Terminal d of A is a mergeable region for compatible `tf` edges from the same net. This implies that edge b_1 can legally lie anywhere within the area defined by d. As long as b_1 does not move to the left past a_5, then by definition there is no violation between instance A and edge b_1, and b_1 should otherwise be invisible to A.

This argument implies that two operations should be applied to edge b_1. First, a constraint of value zero should be added between edges a_5 and b_1 to limit the leftward motion of b_1. Second, edge b_1 should be marked as invisible to *other* edges of instance A:

this avoids the undesired constraint between a_4 and b_1. The marking operation must be performed on an instance-by-instance basis. That is, edge b_1 must be marked as invisible to instance A *only*; b_1 *is visible* to edges from instances other than A.

It is possible to include these two operations for *all* pairs of same-net, same-polarity, different-cell `tf` edges. However, this is not desirable because it incurs considerable unnecessary overhead, and because it can lead to degraded area-efficiency. Many cells consist of just terminal frames, hence they are fully mergeable. It would be costly to process (unnecessarily) the edges of such elements with respect to same-polarity edges at all times. Furthermore, if the merging-limiting constraint were added between all such pairs of same-polarity `tf` edges, the layout area might suffer since the corresponding elements would be artificially constrained to remain in their initial order.

What is desired is to perform these operations *only* in the cases where they are absolutely required, namely when a terminal on layer L_i is closer than a rule to related shapes of the same element that are not part of the terminal. They are needed for a MOSFET, for example, because the drain terminal is closer than a diffusion-polysilicon rule to the gate.

One possible solution would be to process `tf` edges (like b_1) with respect to all edges of nearby instances (like A) in a single operation. However, this sort of feature-oriented constraint-generation scheme is not as efficient as the edge-oriented algorithm described in the following chapter. A second possibility would be to add another edge type to the cell model. The hypothetical type would apply to the back edges of certain terminal frames. These edges would be processed with respect to same-polarity `tf` edges to both add the merging-limiting constraint and to mask the `tf` edges as invisible to the instance the back edge belongs to. Ideally such edges should be identified according to the context in which the cell is used. For example, a terminal that is not connected is the same as a protection frame; hence only its outer edge needs to be processed and its back edge does not need to be specially identified. Identifying the special back-edges would be cumbersome. To identify a_5, a_5 would first have to be recognized as a back edge of a connected terminal frame. Then the other edges of instance A would need to be searched to see if there are any related edges on different layers L_j within S_{ij} of a_5. If one were found (a_4 in this case) then the type of a_5 could be altered.

The method used in the BLM to trigger the two operations is appropriate for edge-oriented constraint generation and it does not require a new edge type. It makes use

of `btf` edges and the notch-filling scheme described above. As already defined, the `btf` type generates a constraint to limit merging. This is augmented by a masking operation termed **hard masking**. When the same-net, same-layer, same-polarity `tf-btf` relationship is found, the `tf` edge is masked such that it is invisible to the element that the `btf` edge belongs to. That is, the `btf` edge completely ("hard") masks the `tf` edge. However, the `tf` edge remains visible to all other elements. For this method to operate properly, the edges must be processed in a particular order, which will be described in the next chapter.

This scheme is correct and relatively simple, but it is not completely general because it is triggered by a `btf` edge. A `btf` edge is present only if there is both a protection frame and an adjacent terminal frame in the masking element. This is true for MOSFETs, because of the notch-filling mechanism outlined above. Although one could imagine elements that could not be modeled in this manner, they are not present in the common IC technologies (bipolar, MOS); hence the hard-masking method has been found to be adequate to date.

4.7 Summary of Cell Model

There are three edge types in the BLM cell model. The edge types are determined from the protection frames and terminal frames of the interface facet of the cell being modeled. The types are defined as follows.

Configuration	Edge Type
coincident terminal frame and protection frame on same layer	terminal frame edge: `btf` protection-frame edge: masked
all other protection frame edges	`pf`
all other terminal frame edges	`tf`

Table 4.1: Edge typing rules.

The typed edges generate spacing constraints via a simple set of rules, which are summarized in the following table. In addition, the `tf-btf` interaction causes the `tf` edge to be hard-masked (make invisible) with respect to all other edges of the element that the `btf` edge belongs to. Edge relationships that do not appear in the table are ignored.

Current Edge	Second Edge				
	Type	Polarity	Net	Layer	Rule
pf	pf	opposite	—	any	S_{ij}
pf	tf	opposite	—	any	S_{ij}
tf	tf	opposite	different	any	S_{ij}
tf	tf	opposite	same	compatible	none
tf	tf	opposite	same	incompatible	S_{ij}
tf	btf	same	same	same	0

Table 4.2: Constraints generated by the various edge types.

4.8 Comparison with Previous Model

The first use of a cell model based on blockages and connection areas was in the Python compactor [5] and the Hawk/Squid framework [37]. Many important ideas, including automatic frame generation, were contributed by these projects. Unfortunately, the previous model did not support terminal merging. Merging is essential in achieving layout densities that compete with the standard symbolic-layout model. For example, the minimum POLY-POLY pitch for adjacent MOSFETs that is achievable using the previous model is 9λ, as shown in Figure 4.41. The model developed in this project is able to achieve the minimum legal pitch of 4λ.

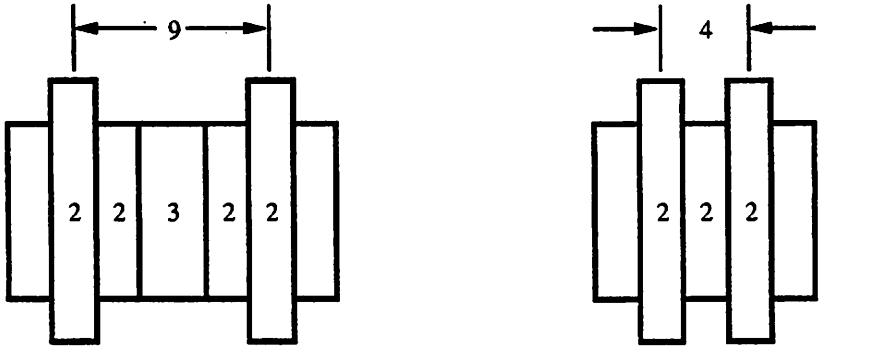


Figure 4.41: Comparison of previous cell model and proposed cell model.

To produce a generic model that also supports merging, the cell model was completely re-formulated in this project. The edge-typing notion and the two forms of masking are both part of the new formulation. The relationships between protection frames and terminal frames are quite different and much more specific. For example, in the previ-

ous model, terminal frames were required to overlap protection frames; also, protection frames were not produced in a manner that allows for discrimination between same-net and different-net blockages. The frames of the new model allow this net-based discrimination to occur, which further enhances mergeability. The framing procedure has been re-formulated as well to produce the new model.

Chapter 5

Constraint Generation

5.1 Introduction

In the constraint-generation phase of layout compaction, the spacing requirements needed to satisfy the design rules and to maintain the connectivity of the layout are determined between all pairs of interacting elements. These requirements are represented via a constraint graph, which is a global model of the layout that can then be analyzed to determine the new element locations.

The generic-element layout abstraction used in SPARCS, i.e., the BLM, is powerful in terms of its ability to capture a wide variety of designs. However, it is a significant problem to generate the constraint graph for a design described in this manner, primarily because the constraint generator must operate without element-type information. The constraint generator must exploit the geometric information present in this low-level layout model without a CPU-time degradation compared to methods that operate on higher-level, typed layout models. A new constraint-generation algorithm which satisfies this requirement has been developed. The new algorithm is described in detail in this chapter.

Chapter 5 is organized as follows. The next section contains a statement of the constraint-generation problem, plus the additional requirements imposed by the goals of this project. Previous work in constraint generation is then presented. The constraint-generation algorithm developed in this project follows. The chapter concludes with the theoretical and measured performance of the algorithm, and a summary.

5.2 The Constraint-Generation Problem

The constraint-generation process takes as input a symbolic layout and the design rules. Its output is a constraint graph which is complete,[1] meaning that all constraints which are necessary to guarantee that a legal layout can be realized are present. Most of these constraints, which will be referred to as **implicit constraints**, arise from minimum-spacing requirements and from the requirement that the layout remain electrically connected. Other, **explicit constraints** may also be added. For example, constraints might be included to place the power and ground busses at specific locations in the compacted layout. Such constraints are termed explicit because they cannot be deduced from the layout or the design rules. The explicit constraints are often called **user constraints** in the literature. For convenience, a source node and a sink node are typically appended to the graph as well.

The constraint generator must produce, for element Q, constraints with respect to all elements R_i which might interact with Q and which are related to Q by one or more design rules. This analysis is similar to a design-rule check, but with an important difference. In design-rule checking, the layout elements do not move. Hence, for element Q, it is only necessary to examine the elements within a worst-case distance of Q. In compaction, the layout elements *do* move. It is not sufficient to confine the constraint-generation analysis to a fixed-size window about Q, because elements that are initially far from Q may move close to Q during compaction.

In one-dimensional compactors constraint generation is performed one dimension at a time, because compaction in a particular dimension changes the element adjacencies in the orthogonal dimension. In horizontal compaction the analysis compares the vertical edges of the elements; in vertical compaction, the horizontal edges are compared.

In addition to being complete, the constraint graph should be minimal. A minimal graph has no redundant constraints. The constraint AC in the graph shown in Figure 5.1 is redundant, since the path ABC is longer than the path AC. Minimizing redundancy in the constraint graph is important, because the runtimes of the constraint-solving algorithms grow with the number of constraints.

[1] This does not mean complete in the graph-theoretic sense.

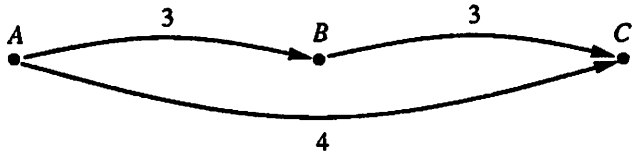


Figure 5.1: Constraint AC is redundant.

5.2.1 Visibility Considerations

In a horizontal (vertical) compaction iteration, a simple but inefficient constraint generator would compare every vertical (horizontal) edge of every element to every vertical (horizontal) edge of every other element. The graphs generated by this algorithm would be correct. However, the generation algorithm would require time proportional to the square of the number of vertical (horizontal) edges in the layout, and many redundant constraints would be generated. Practical constraint-generation algorithms, including the one described in this chapter, use *visibility* techniques both to generate fewer redundant constraints and to speed up the constraint generator itself.

Visibility considerations can be used to prune the search space in two ways. Consider a horizontal compaction step. A given vertical edge E needs only to be compared to edges within the region defined by the projection of E on the vertical axis, as depicted in Figure 5.2. Edges above and below this region, e.g. edge E_j, do not interact with E because their vertical projections do not intersect.[2] That is, edge E_j is not visible to edge E. From now on, the projection of an edge will be referred to as its **interval**; when the intervals of two edges intersect they will be said to **overlap**. This type of visibility analysis will be called **perpendicular visibility**.

The second form of pruning that follows from visibility analysis, which will be termed **parallel visibility**, concerns edges whose intervals *do* overlap. Figure 5.3 shows four rectangles arranged so that their intervals overlap. All rectangles are assumed to be blockages that are on the same layer. Each rectangle needs only to be constrained with respect to the rectangles visible to it, i.e., to its immediate neighbors. For example, rectangle r_3 is not visible to r_1 because it is blocked by the intervening rectangle r_2. Hence the three constraints shown in Figure 5.3 are sufficient to model this layout for horizontal

[2] Actually, the interval of E must be enlarged to enable the generation of corner constraints. This factor will be described later in this chapter.

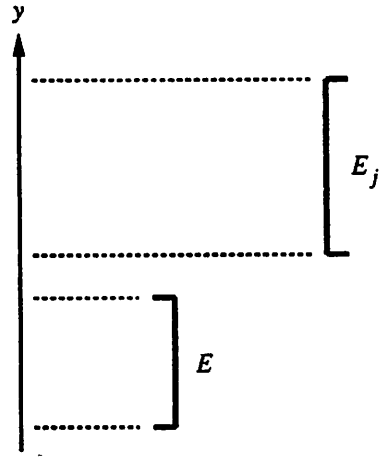


Figure 5.2: Visibility perpendicular to the spacing direction.

compaction. This conclusion also follows from a simple transitivity argument.

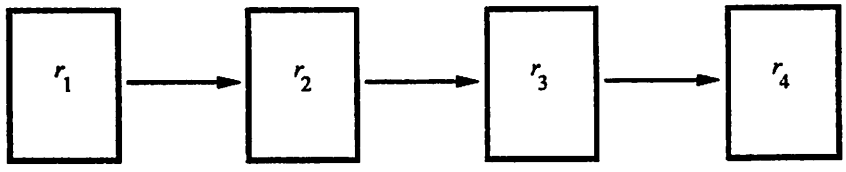


Figure 5.3: Visibility parallel to the spacing direction.

5.2.2 Additional Requirements

All constraint generators should produce a complete, yet minimal, set of constraints. They should also be as efficient as possible in terms of memory and CPU-time usage. In addition, several other requirements are imposed by the goals of this project, and by the layout model chosen. These requirements are described in the following paragraphs.

Per-Layer Manhattan Shapes. The BLM cell model is defined in terms of per-layer Manhattan geometries. The constraint generator must take advantage of this information by generating the constraints on a per-layer basis using per-layer rules. Also, the polygonal shapes must be treated as such by the constraint generator. This means that, for a given element, the boundary of the element on each layer must be modeled as a set of edges, not a single edge.

Merging of Generic Elements. The generator must support terminal merging without using element-type information. This amounts to exploiting the edge types that have been defined in the cell model.

Non-Transitive Spacing Rules. The constraint generator must not be restricted to transitive spacing rules. Referring to Figure 5.4, the rules are **transitive** if $S_{ik} \geq S_{ij} + W_j + S_{jk}$, where W_j is the minimum width for layer j, for all combinations of layers. Note that constraint S_{ik} *must* be generated if the rules are not transitive, whereas it is redundant if they are.

Some constraint generators assume that the rules are transitive in order to simplify the algorithms and to enhance runtime efficiency. However, real technologies, such as CMOS, have non-transitive rules.

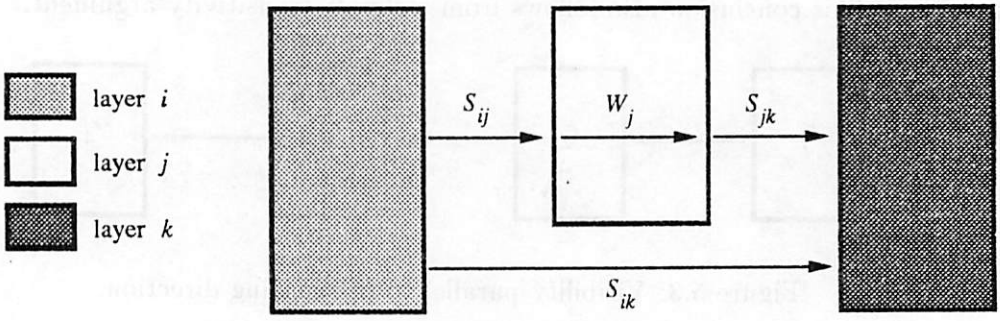


Figure 5.4: Definition of transitive spacing rules.

5.3 Other Constraint-Generation Algorithms

The shadow-propagation method and the intervening-group method are the two most common techniques for constraint generation.[3] Each method is outlined below for a horizontal compaction step; constraint generation for vertical compaction is similar. An alternative constraint-generation approach has been used in this project. It is described in detail following this section.

[3]The most-recent-layers algorithm [11] can be thought of as a variant of the shadow-propagation method that applies to the virtual-grid environment.

5.3.1 Intervening-Group Methods

Intervening-group methods use longest-path information to determine which pairs of layout elements must be constrained. The first intervening-group method was proposed by Kingsley [39]. An efficient implementation of the algorithm is described by Hedges et al. in [29].

In generating a horizontal constraint graph, the layout elements are examined in turn from left to right. When the current element e_c is processed, it is compared against all elements to its left, i.e., against all previously-processed elements. However, constraints are not generated between all pairs.

Let the (already-processed) element that e_c is currently being compared to be denoted e_r. First, e_c and e_r are checked to insure that their intervals overlap. Assuming that they do, the graph as completed thus far is searched to determine if a path already exists from e_r to e_c. If a path exists, and if the length of that path is greater than or equal to the maximum design rule (MDR), then no constraint is needed between e_r and e_c. If not, the geometries of e_c and e_r are compared in detail to generate the necessary constraint. Constraint generation for e_c continues with the next element to the left of e_r, e_{r-1}.

A key fact that is exploited by the algorithm is that any elements further to the left, that are constrained to be at least an MDR away from e_r, cannot affect the position of e_c. That is, the "intervening" element e_r blocks such elements with respect to e_c, hence they do not require constraints to e_c. Let e_{r-1} be one such element. It follows in turn that any elements to the left of e_{r-1} that are constrained to be at least one MDR to its left cannot affect e_c either.

The implementation of Hedges et al. [29] uses bit vectors to store this longest-path information. Each element has a bit vector, with one bit for each element to its left in the layout. For an element e, the value of bit i is 1 if element e_i is already constrained to be at least an MDR to the left of e. In the above example, if a constraint is generated between e_r and e_c *and* if the value of the constraint is greater than or equal to the MDR, then the bit corresponding to e_r is set in the bit vector of e_c. The bit is likewise set if it was determined that e_r and e_c were already constrained by a path at least one MDR long. Assume that e_r is constrained to be an MDR or more from e_c. The information that elements like e_{r-1} are likewise separated from e_r (and hence cannot affect e_c) is then inherited by e_c, by ORing e_c's bit vector with that of e_r. When e_c is being compared to e_{r-1}, the bit corresponding

to e_{r-1} is found to be 1, so no further consideration of e_{r-1} is necessary.

As described above, a longest-path search is performed from e_c to e_r if there is no known path between them greater than or equal to one MRD in length; i.e., if the bit corresponding to e_r in e_c's bit vector is 0. In the simplest case, each such longest-path computation requires time proportional to the number of edges in the partially-completed graph. To reduce the runtime of the algorithm, the search is carried out in breadth-first order to a limited depth only, typically from 2 to 4 nodes [29]. The runtime is thereby reduced, but some redundant constraints are generated. If the longest-path search depth were ∞, no redundant constraints would be created.

Weaknesses

The intervening-group algorithm always performs a quadratic number of element comparisons, because each element is compared with every element to its left in the layout. In the best case, a linear number of longest-path analyses are performed as well. If bit vectors are used to store the longest-path information, then the amount of storage needed for them is quadratic in the number of elements. These factors, namely quadratic storage and quadratic runtime, indicate that the intervening-group method is best-suited to small examples.

5.3.2 Shadow Propagation

The shadow-propagation algorithm was first described by Hsueh [31]. The algorithm has since been implemented in a large number of compactors.

In the shadow-propagation method the layout is scanned *in the direction of compaction.* In horizontal compaction, the layout is scanned from left to right. During the scan, spacing constraints are generated by processing the right-side edges of each element with respect to the left-side edges of other elements further to the right in the layout. The method takes advantage of the fact that a `max` (right-hand) edge only needs to be processed until it "sees" an appropriate `min` (left-hand) edge to its right. Consider the example given in Figure 5.5, where all three shapes are blockages on the same layer. The right-hand edge of A (a_2) does not need to see any edges past element B, since the presence of B obviates the need for a constraint between A and C. That is, the constraint between A and C is dominated by the sum of the AB and BC constraints; constraint AC is redundant. In

visibility terms, as outlined earlier, a_2 is **covered** by element B, and therefore a_2 does not need to be processed once B has been reached.

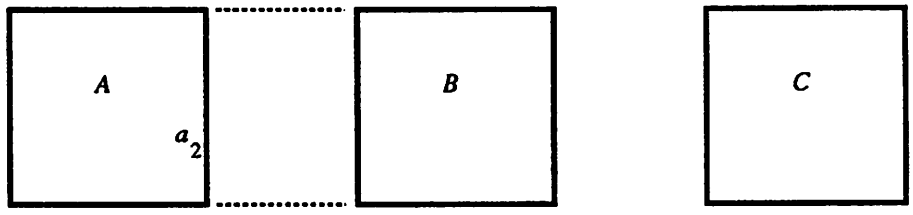


Figure 5.5: Shadow coverage.

The progression of the algorithm, again for a case where all shapes are blockages on the same layer, is depicted in Figures 5.6 through 5.10. At position X_2 in Figure 5.6, edge a_2 enters the **shadow**, which is a data structure that contains the currently-active **max** edges. At all times, the shadow is comprised of the **max** edge segments that are visible to an observer looking to the left from the current position in the layout. When position X_3 is reached constraint AB is generated, and at X_4 AC is created, as shown in Figure 5.7. At X_5 and at X_6 portions of a_2 become covered by the addition of edges b_2 and c_2 to the shadow (Figure 5.8). At the next position, X_7, three constraints are generated as indicated in Figure 5.9. The last part of edge a_2 is covered at X_8 as depicted in Figure 5.10.

The covering operations are trivial in this contrived example; they are not this simple in real examples, especially when terminal merging is allowed. For example, a **pf** edge cannot be covered by a **tf** edge. Consider the three-instance example shown in Figure 5.11, where all elements are on the same layer. Element A has a single protection frame, while elements B and C are composed of terminal frames only and they are in the same net. (The wire that connects them is omitted from Figure 5.11.) If edge a_2 is (mistakenly) covered by edge b_2, the illegal result shown in Figure 5.11 could be produced. To guarantee correctness, a_2 cannot be covered until a **max** edge from a protection frame is reached. That is, both **pf** and **tf** edges must be propagated until a same-polarity **pf** edge is reached.

The complexity of shadow propagation depends strongly upon the data structures used, the design rules allowed, and the degree of terminal merging allowed. A simple implementation on a worst-case layout has quadratic complexity [31]. A scanline traversal of a layout with n edges perpendicular to the compaction direction requires $O(n \log n)$ operations. Assuming (unrealistically) that all other operations can be performed in constant

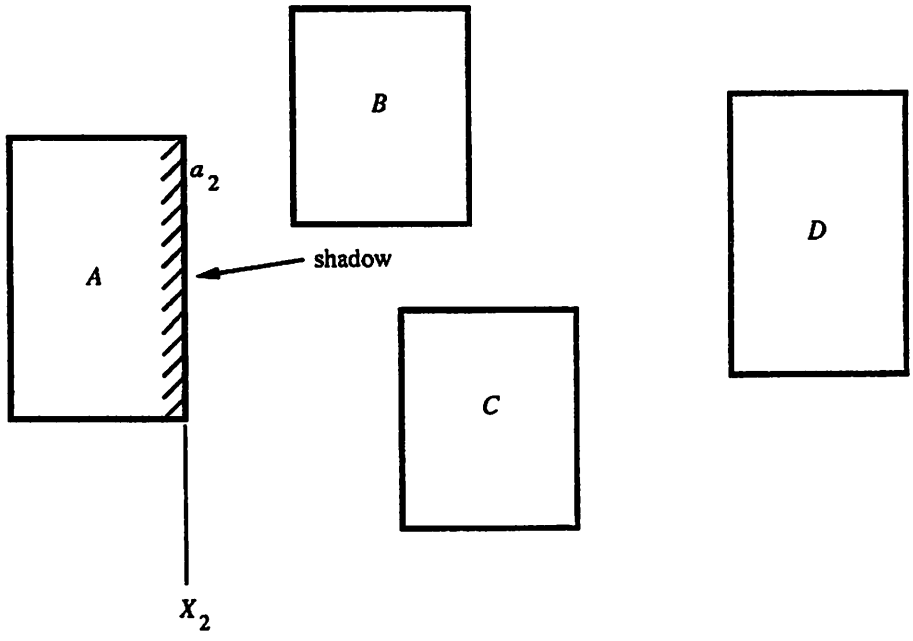


Figure 5.6: Example of shadow propagation.

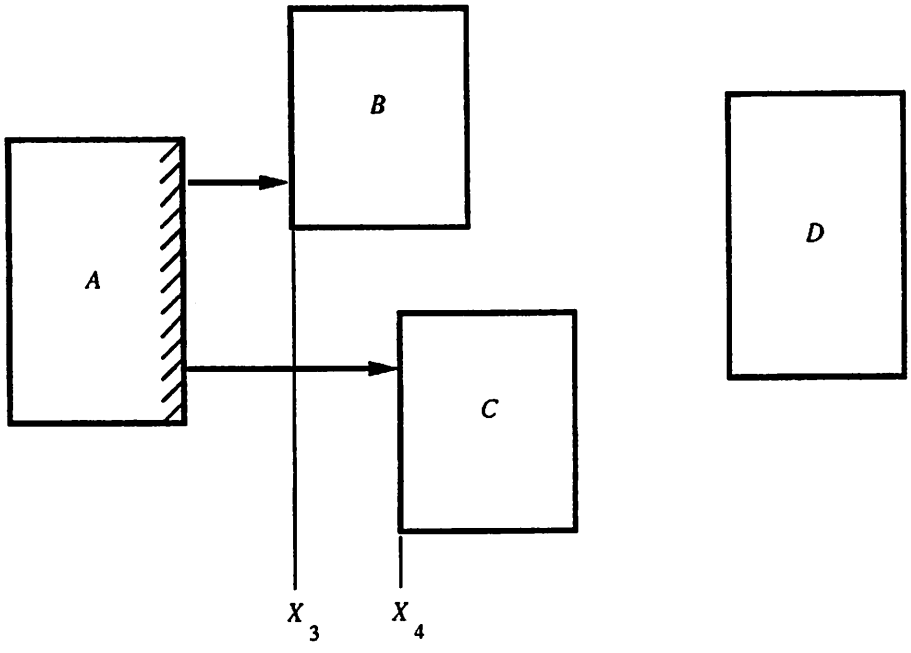


Figure 5.7: Example of shadow propagation, continued.

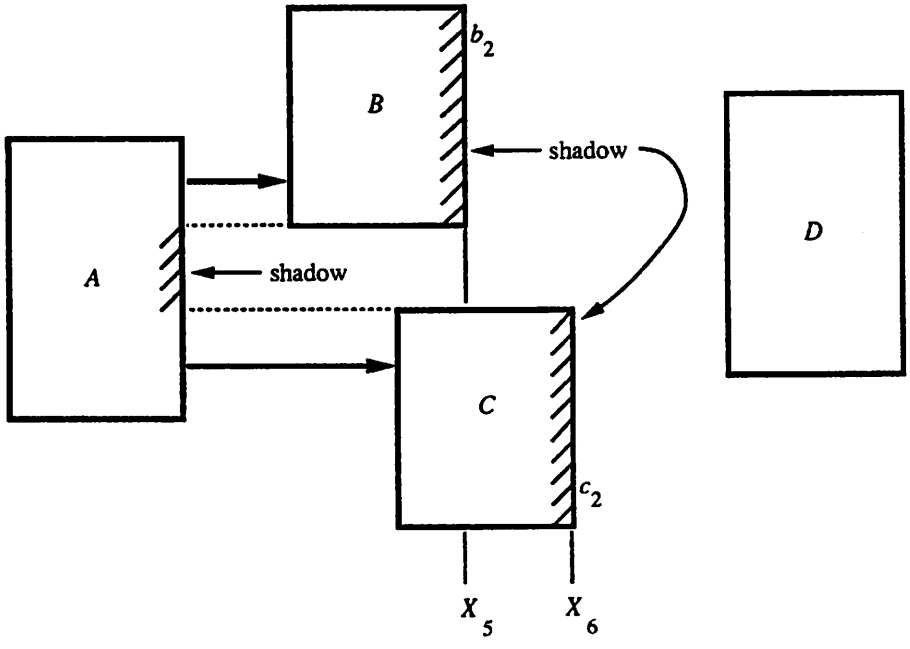


Figure 5.8: Example of shadow propagation, continued.

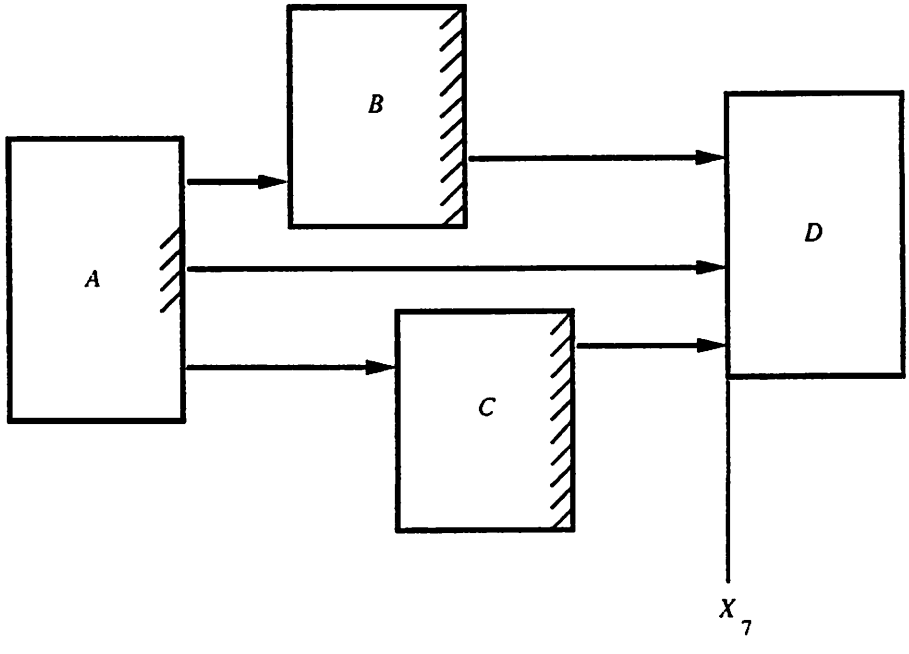


Figure 5.9: Example of shadow propagation, continued.

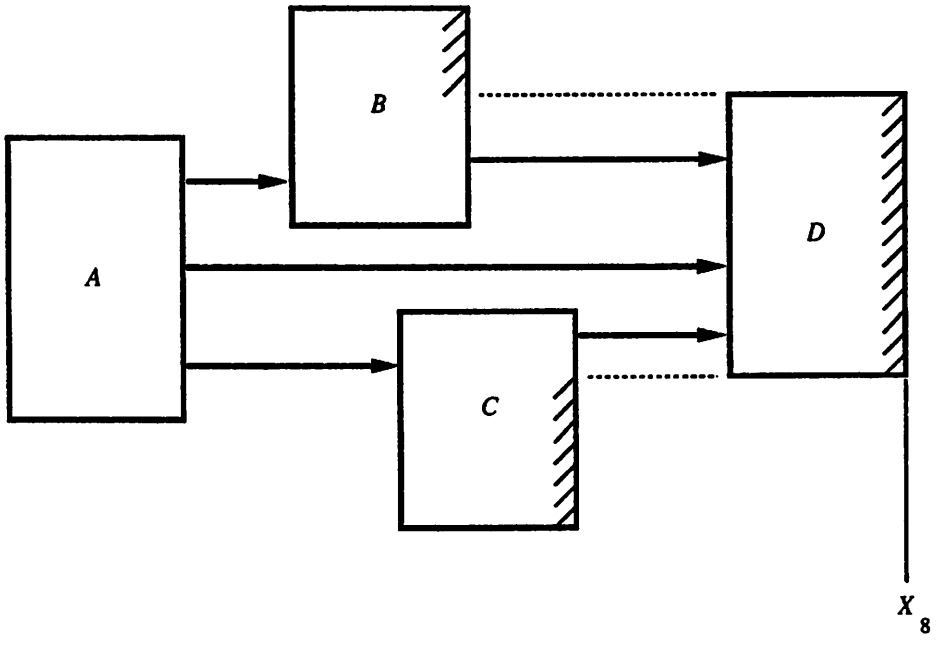


Figure 5.10: Example of shadow propagation, continued.

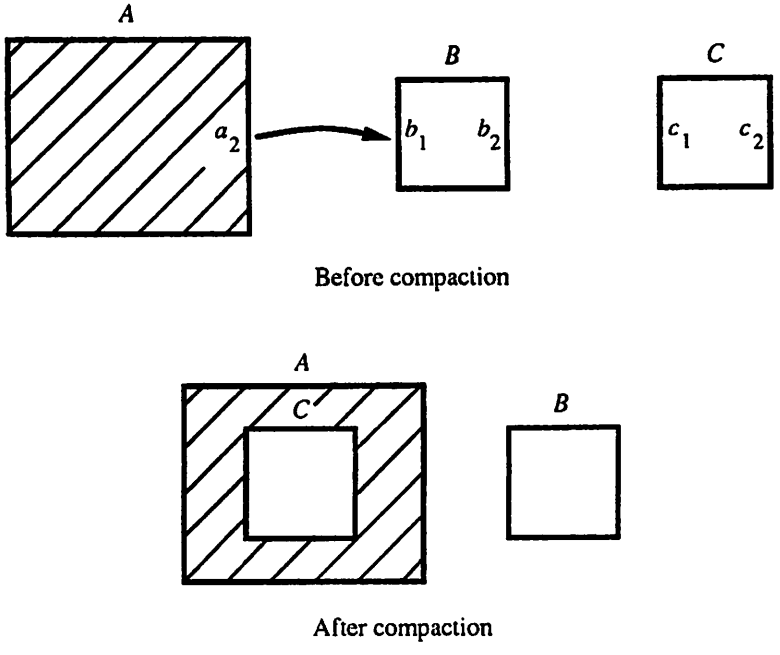


Figure 5.11: Illegal result if `tf` edge covers `pf` edge.

time, the lower bound on shadow propagation is thus $O(n \log n)$. Hsueh quotes a measured complexity of $O(n^{1.2})$ [31].

Weaknesses

Although the shadow-propagation algorithm has proven to be a workable method for constraint generation, it has several weaknesses. The weaknesses have to do with the complicated data structures that are necessary to implement it. In particular, multiple shadows are required because a separate shadow must be maintained for each mask layer in the layout.[4] The shadows should be implemented via geometric data structures that allow new edges to be efficiently compared to them as the new edges are encountered. This is needed to efficiently exploit the first type of visibility pruning noted earlier. Tree-oriented data structures, such as the segment tree [63], are good candidates. However, they must remain relatively well-balanced to achieve high efficiencies. Re-balancing is necessary even in the simplest cases, such as those where the only layout elements present are rectangular blockages.

Over a given interval each shadow must be able to represent one `pf` edge plus an arbitrary number of `tf` edges, because only `pf` edges on the same layer can perform covering. For example, the shadow contains one `pf` edge and two `tf` edges following edge c_2 in the example shown in Figure 5.11. Furthermore, the multi-edge intervals are not those that are defined by the edges themselves. Rather, artificial intervals must be created dynamically, by intersecting the real edges as shown in Figure 5.12. Also, shadows become covered in a rather arbitrary manner. For example, a shadow edge that initially spans a wide interval may fragment into many small pieces before it is completely covered. Both the artificial intervals and the edge fragmentation necessitate additional, periodic re-balancing of the shadow data structures as the algorithm executes.

5.3.3 Summary

The intervening-group and shadow-propagation methods are often compared to one another. However, they are actually complementary, since one prunes the the search space using geometric considerations (shadow propagation) while the other does so via the

[4] This statement is not strictly true; the layers can be grouped into equivalence classes such that one multi-layer shadow suffices per class. It is still necessary to have multiple shadows, however, and multi-layer shadows are themselves difficult to handle.

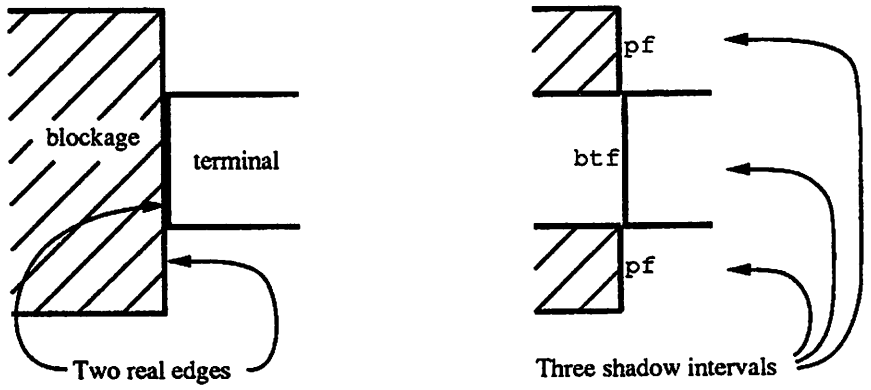


Figure 5.12: Shadow intervals defined by edge intersections.

use of longest-path information (intervening group). A detailed comparison of the two approaches has not been published.

The argument put forth in favor of the intervening-group algorithm is that it is simpler than shadow propagation and therefore faster [39,29]. This does not seem likely, especially for large examples, because of its quadratic complexity. Intervening-group methods still must perform detailed edge comparisons. If geometric pruning techniques (such as those used in shadow propagation) are not employed, then the edge-comparison component of the algorithm could be an important factor in its runtime. This factor is most significant when the elements have a large number of edges.

The shadow-propagation algorithm uses visibility analysis quite effectively, thereby pruning the search space geometrically. However, the data structure manipulation that is necessary adds a significant component to the runtime of the algorithm. This factor makes it difficult to design and implement an *efficient* constraint generator based on shadow propagation for low-level layout models like the one used in this project.

The fundamental operation in constraint generation is the comparison of element edges, whether or not longest-path pruning is employed. The research in constraint generation in the SPARCS project, which is described in the remainder of this chapter, has thus concentrated on this aspect of the problem.

5.4 Plane-Sweep Constraint Generation

The difficulties inherent in applying the shadow-propagation method to the layout model used in this work have led to the investigation of alternative methods for constraint generation. As a result a new algorithm, called **perpendicular plane-sweep** (PPS), has been designed and implemented. The basic characteristic of this algorithm is that the layout is scanned in the direction *perpendicular* to the spacing direction instead of parallel to it. For example, if the spacing direction is horizontal then the layout is scanned from bottom to top, rather than from left to right. The first constraint generator to employ perpendicular scanning is described in [50].

The PPS method has several key advantages over the shadow-propagation method. In particular, *no explicit covering operations are needed*, even though the edge-comparison search-space is pruned at least as effectively as in the shadow-propagation method. The algorithm can be implemented with simpler data structures, and the overhead due to data-structure maintenance is substantially reduced.

A brief description of the PPS algorithm appears in the following subsection. Previous work in constraint generation via perpendicular scanning is described in Section 5.5.

5.4.1 Basic Operation

The basic operation of the plane-sweep algorithm is illustrated by considering again the simple example of Figure 5.6, where all shapes are rectangular, all shapes are on the same layer, and all shapes are protection frames. The simplifications made here will be removed later in this chapter.

Since a horizontal spacing is assumed, the vertical edges of the elements participate in constraint generation. For each vertical edge, its x-coordinate will be referred to as its **loc** coordinate, its lesser y-coordinate as its **min** coordinate, and its greater y-coordinate as its **max** coordinate. The loc, min, and max coordinates will be denoted C_{loc}, C_{min}, and C_{max}, respectively. Similar definitions apply for horizontal edges. The edge definitions are illustrated in Figure 5.13.

In horizontal compaction the algorithm operates by sweeping a horizontal scanline across the layout *vertically*, from bottom to top. The scanline stops at discrete **events**, which are defined by the C_{min} and C_{max} coordinates of the edges. Processing only occurs at events. The y-coordinate of the scanline at a particular event is called the **event** coordinate.

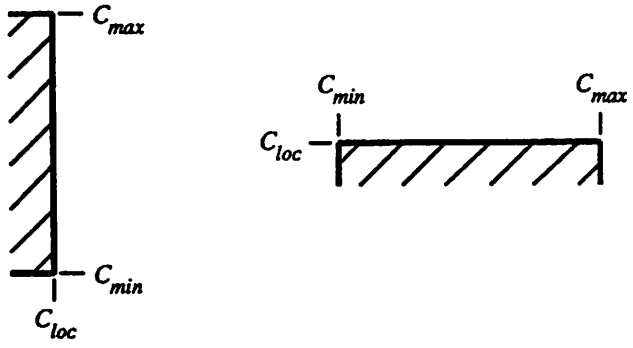


Figure 5.13: Edge definitions.

C_e. Disregarding (for now) interactions between corners of shapes, there are two *types* of events.[5] The first type corresponds to the case when the scanline reaches the lower y-coordinate of an edge; i.e., when C_e equals C_{min} of some edge. Such an event is called a **MIN** event. The second type corresponds to the case when the scanline reaches the upper y-coordinate of an edge; i.e., when C_e equals C_{max} of some edge. Such an event is called a **MAX** event.

Each edge appears in two event queues, a MIN queue and a MAX queue. The C_{min} of each edge is its key in the MIN queue, and its C_{max} is its key in the MAX queue. The event queues are sorted in increasing order; at all times, the first edge in either queue corresponds to the C_e of the next event of that type. The movement of the scanline across the layout from bottom to top (or right to left, in a vertical compaction) is realized by processing edges in the order specified by the sorted event queues. The next event coordinate is determined by the edge of lesser key from the MIN and MAX queues.

If the event is a MIN event, all edges whose C_{min} is equal to C_e are inserted into a data structure called the **edge structure**. The edge structure is sorted in increasing order of the C_{loc} coordinates of the edges. Upon insertion, each new edge is processed to generate constraints. If the event is from the MAX queue, then all edges with the same C_{max} are deleted from the edge structure. At any point in time the edges in the edge structure are those edges that are currently **active**. An edge participates in constraint generation only while it is active, and an edge is active for all event coordinates between its C_{min} and C_{max} coordinates, inclusive.

[5]There are three types when corner constraints are computed as well. Corner constraints are addressed later in this chapter.

The first event is always a MIN event as shown in Figure 5.14. In this example, edges c_1 and c_2 are inserted into the edge structure at event Y_1. Since there are no other edges in the edge structure when event Y_1 is reached, no constraints are generated. At event Y_2, edges a_1 and a_2 are activated by adding them to the edge structure (Figure 5.15); there are now four active edges. Spacing constraints arise from the comparison of opposite-polarity edges. Since edge a_2 is a **max** edge, it has constraint relationships with **min** edges to its right in the layout. Similarly, since edge a_1 is a **min** edge, it has constraint relationships with **max** edges to its left.

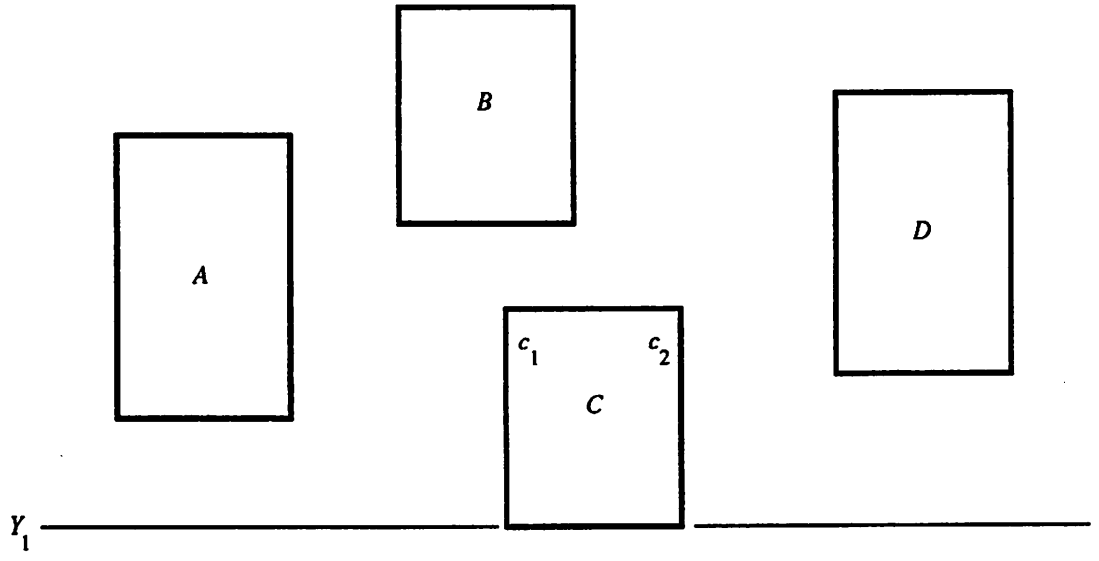


Figure 5.14: Initial scanline position - MIN event.

When a **max** edge is added to the edge structure, it is is processed by searching through the edge structure in increasing x to generate constraints. In the case of edge a_2, the opposite-polarity edge c_1 is encountered and a constraint is generated as shown in Figure 5.15. Likewise, when a **min** edge is added, constraints are generated by searching through the edge structure in decreasing x. The operation of searching the edge structure in the direction of increasing coordinates (x in this case) will be called the **walk_up** operation, since an edge is processed by "walking" through the active edges until an opposite-polarity edge is encountered. The dual operation, for searching in the direction of decreasing coordinates, will be called the **walk_down** operation. For edge E these operations will be denoted `walk_up(`E`)` and `walk_down(`E`)` (if the direction is not relevant, the operation will be denoted `walk(`E`)`). During `walk(`E`)` in this simple example, all edges from the same ele-

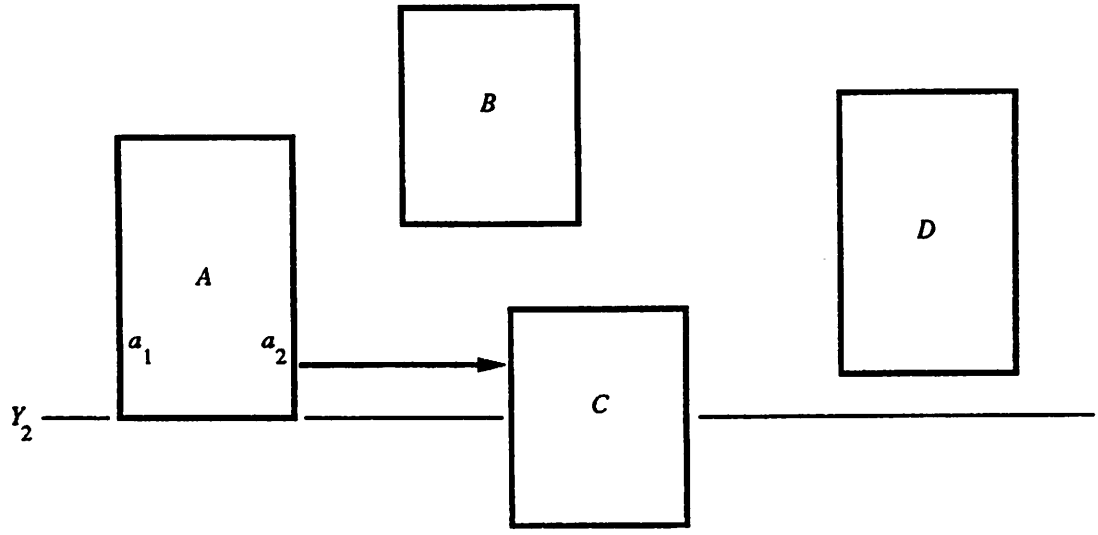


Figure 5.15: Second scanline position - MIN event.

ment, all edges of the same polarity, and all edges from unrelated layers are skipped. When **walk_down**(a_1) is performed no constraints are generated, because there are no edges further to the left in the edge structure.

It is important, for efficiency reasons, to visit as few edges as possible in the walk operation. That is, the walk operation of an edge should be terminated as soon as that particular edge is adequately constrained. In this example, the walk may terminate as soon as an opposite-polarity edge is encountered; e.g., **walk_up**(a_2) terminates at edge c_1. The actual termination criteria used in the PPS algorithm are more involved; they are described later in this chapter.

Event Y_3 in Figure 5.16 is also a MIN event. The **walk_down**(d_1) operation sees c_2 and the new constraint shown in Figure 5.16 is generated. The next event, Y_4, is a MAX event and all of the edges with $C_{max} = C_e$ are deactivated by removing them from the edge structure. As shown in Figure 5.17, edges c_1 and c_2 are removed; they will no longer participate in constraint generation.

The last event that causes constraints to be generated is event Y_5. Both b_1 and b_2 lead to the generation of constraints as depicted in Figure 5.18. All other events (Y_6, Y_7, Y_8) are MAX events, and hence no more constraints result. Compared to the shadow-propagation method, the plane-sweep algorithm creates one less constraint for this example. It is clear, because of the fact that a layer is transitive with respect to itself, that the extra

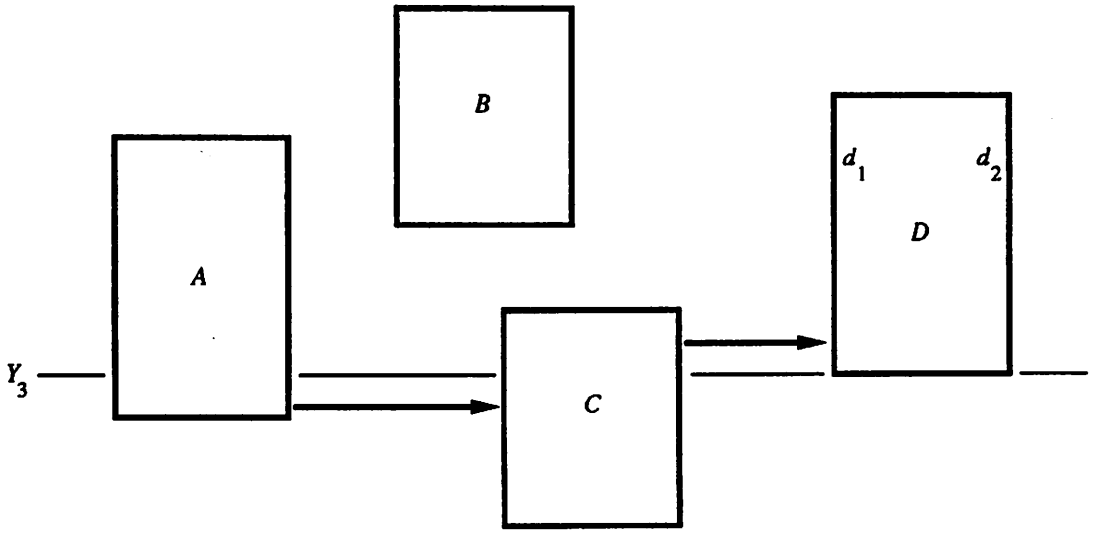


Figure 5.16: Third scanline position - MIN event.

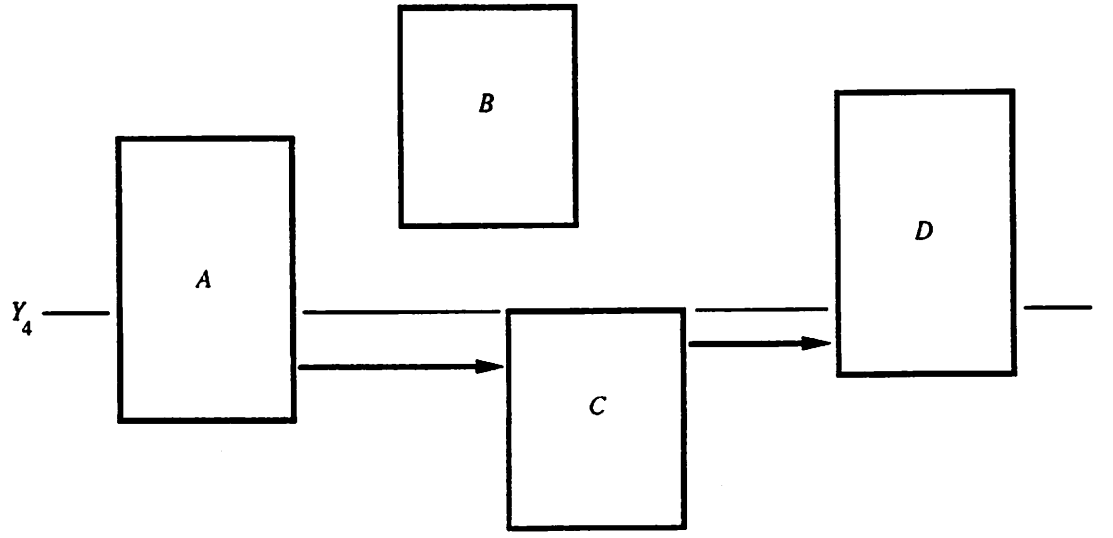


Figure 5.17: Fourth scanline position - MAX event.

constraint shown in Figure 5.10 is redundant.

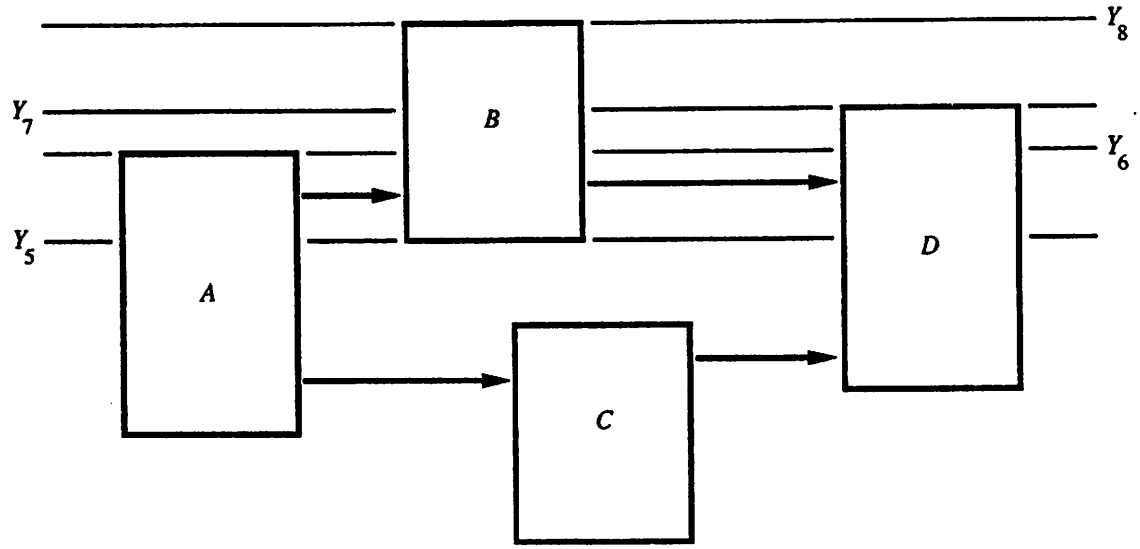


Figure 5.18: Fifth scanline position - MIN event.

5.4.2 Advantages

The major advantage of perpendicular scanning over the parallel-scanning approach used in shadow propagation is that *the edge-covering operations are performed at very low cost.* A complicated shadow data-structure and the associated updating is not required.

The perpendicular visibility analysis occurs automatically when perpendicular scanning is used, due to the nature of the scanline. Two edges have overlapping intervals *only* if both edges are in the edge structure at the same time. In the above example, the edges of B and C are not visible to one another because they are never simultaneously active. An explicit operation is necessary to determine the shadow edge(s) that are perpendicularly visible to a given edge in shadow propagation. In addition, if the edge under consideration covers part of the shadow, the shadow must be explicitly updated.

Parallel visibility is also easy to exploit, because of the sorted edge structure. The termination criteria for the walk operation can be defined to take advantage of parallel visibility at least as effectively as the shadow-propagation method does. This point will become apparent as this chapter proceeds.

5.5 Previous Work in Plane-Sweep Constraint Generation

The PPS algorithm is based on the algorithm first described by Lengauer in 1983 [50]. A more detailed description of the algorithm, which will be referred to herein as the D-L algorithm, was published in 1987 by Doenhardt and Lengauer [21]. Other contributions to the technique were described by Malik in 1987 [55]. These algorithms for plane-sweep constraint generation are described here. The PPS algorithm will be described in detail and compared with these algorithms following this section.

Doenhardt and Lengauer treat the theoretical aspects of the constraint-generation problem for layouts comprised of *rectangles* on a *single layer*. The primary emphasis of their work is on generating a minimal set of constraints. As pointed out in [21], the transitive reduction of a (non-minimal) constraint graph corresponds to the desired minimal set.

The abstraction of real layouts with many layers and polygonal shapes to single-layer, rectangle layouts is not addressed in [21]. The authors do state that the abstraction of real layouts to their layout model is "nontrivial".

Actually, Doenhardt and Lengauer present a family of algorithms, each of which provides a different degree of merging. The simplest algorithm, Gen_0, runs in $O(n \log n)$ time but does not allow any merging. Since all rectangles are on the same layer and no merging is allowed, the walk operation can be terminated as soon as the first rectangle in the edge structure[6] is encountered, as illustrated in the example in the preceding section. The walk operation thus requires $O(1)$ time per rectangle. If the active rectangles are stored in a balanced binary tree, then maintaining the active rectangles and sorting the event queues take $O(n \log n)$ time for an n-rectangle problem. This leads to the overall time-bound of $O(n \log n)$.

The algorithm outlined in Section 5.4.1 does not create a transitive reduction; under the current assumptions, as many as $2n-1$ constraints can be generated when $n-1$ are sufficient. For example, the constraints drawn as dashed lines in Figure 5.19 are redundant. The redundant constraints are avoided in Gen_0 by storing constraints on a tentative basis until MAX events, instead of generating them immediately during MIN events. This is accomplished by maintaining, for each rectangle R_i, a pointer to the rectangle currently to its left, called $cand(R_i)$. These pointers are initialized and updated at MIN events. In

[6] The D-L algorithm is rectangle-based, not edge-based. In this explanation of the D-L method, the term "edge structure" actually refers to a rectangle structure, and the walk operation processes rectangles, not edges.

the case of R_4 in Figure 5.19, $cand(R_4)$ is updated to point to R_1, R_2, and R_3 at events Y_2, Y_3, and Y_5, respectively. At the MAX event for rectangle R_i, constraints are generated between $l(R_i)$ and R_i if $l(R_i) = cand(R_i)$, and between R_i and $r(R_i)$ if $cand(r(R_i)) = R_i$. The notation $r(R_i)$ means the right neighbor of R_i in the edge structure, and $l(R_i)$ means the left neighbor. The constraint between R_3 and R_4 is generated at Y_7, e.g., because $cand(r(R_3)) = R_3$. This scheme is described more fully in [21].

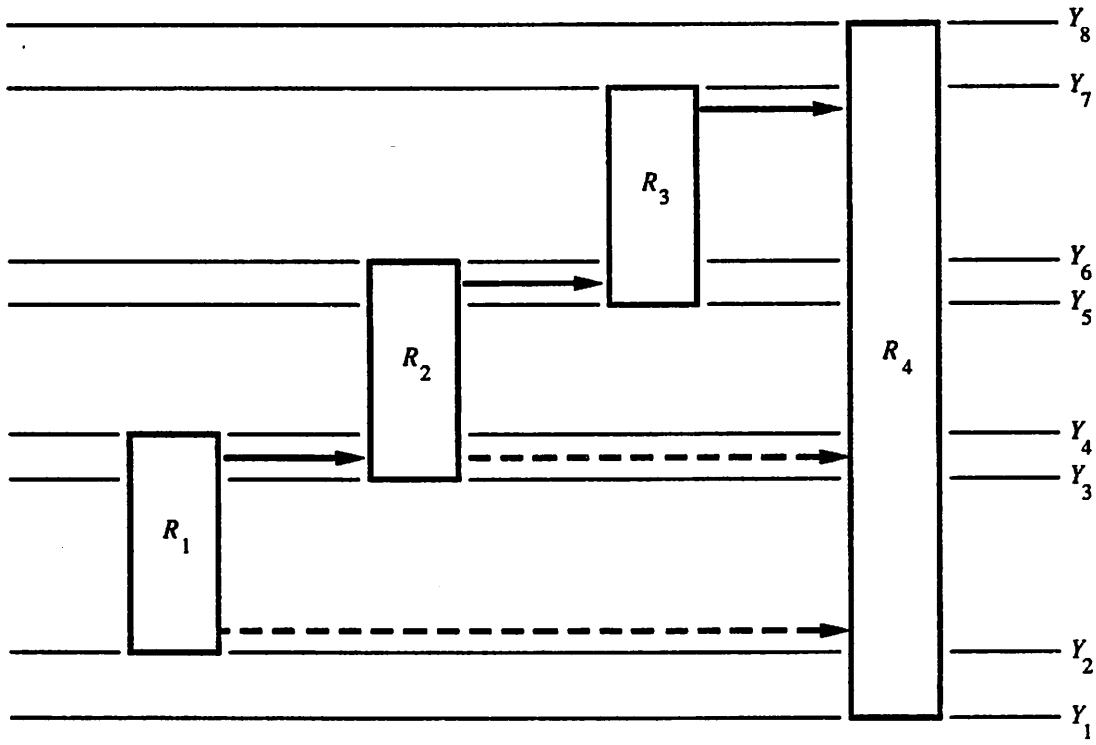


Figure 5.19: Elimination of redundant constraints.

The merging problem, within this rectangle-based, single-layer model, is addressed by several algorithms based on Gen_0. The Gen_1 algorithm allows a rectangle to merge with at most one neighboring rectangle. The asymptotic complexity of Gen_1 is the same as that of Gen_0 and an irredundant graph is produced, but at the expense of making two sweeps of the layout, one from the bottom to the top followed by one from the top to the bottom. The authors do not advocate the use of Gen_1 because of the limited degree of merging that it affords.

The parameterized algorithm $Gen(k)$ is the most flexible variant of the D-L algorithm. The parameter k controls the amount of merging, in that a rectangle is allowed to

merge with up to k neighboring rectangles that are in the same net. Since merging of up to k rectangles is possible, the walk operation is $O(k)$ rather than $O(1)$, and the overall runtime of $Gen(k)$ is $O(n(k + \log n))$. A single sweep of the layout is performed, and the graph produced is not minimal, unlike Gen_0 and Gen_1. The $Gen(k)$ algorithm is somewhat involved; hence the description given in [21] is not reproduced here.

In [55], Malik extends the Gen_0 algorithm of [50] to multiple layers. The main contribution of Malik's algorithm is in the stopping criteria for the walk operation. In the multi-layer case, the walk operation for an edge on layer L_i can always be terminated when a blockage on the same layer is reached. This strategy is conservative; sometimes the walk operation can be terminated sooner. In the example shown in Figure 5.20 (assuming horizontal compaction), R_1 and R_3 are on layer L_i, and R_2 is on layer L_j. If $C_{13} \leq C_{12}+W_2+C_{23}$ then the processing of R_1 can terminate at R_2. This criterion can be encoded as a function of L_i, L_j, and W_2. Such an encoding, which does not require transitive rules, is proposed in [55] via the use of a three-dimensional array of bit vectors.

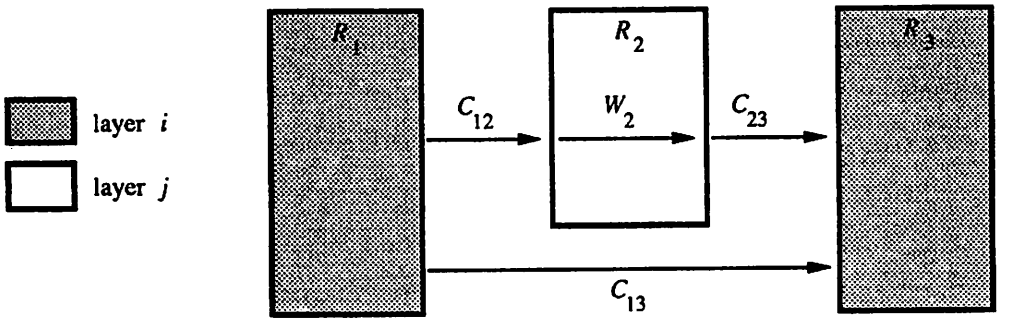


Figure 5.20: Aggressive stopping criteria.

5.5.1 Weaknesses

The D-L algorithm is interesting and innovative from a theoretical perspective. However, the algorithm does not directly apply to real layouts for the following reasons.

There does not seem to be an easy way to transform a realistic layout into a set of single-layer problems. Consider an example comprised of two identical MOSFETs from a simple technology as shown in Figure 5.21. Each MOSFET is represented by two rectangles, one on POLY and one on NDIF. This example has been further simplified by excluding terminal merging.

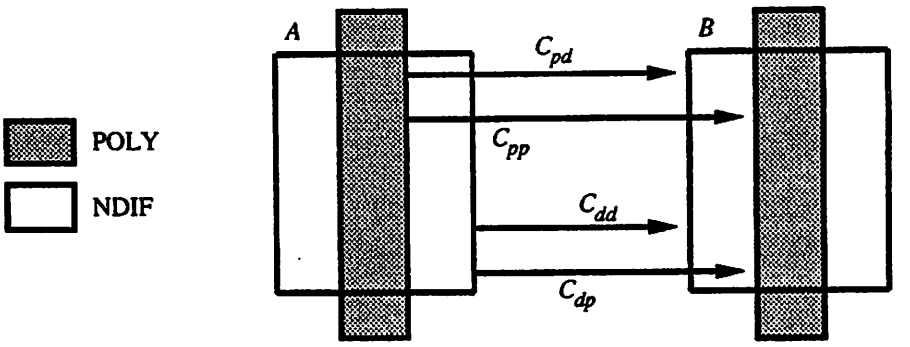


Figure 5.21: Edge comparisons for two MOSFETs.

Four edge comparisons are necessary to derive the limiting constraint between these two elements. Two of the comparisons are same-layer comparisons (C_{pp} and C_{dd}), and two are cross-layer comparisons (C_{pd} and C_{dp}). The two same-layer comparisons can be easily modeled via two independent single-layer problems.

The difficulty arises in attempting to model the cross-layer comparisons C_{pd} and C_{dp}. To fit into a single-layer framework, synthetic layers must be derived for each comparison. Consider the C_{pd} case; the POLY rectangle from element A must be represented on a synthetic layer, say layer POND. Likewise, the NDIF rectangle from B must be represented on layer POND. Creation of this third layer allows for the comparison to be made. However, note that the shape on POND differs *depending on the instance being modeled*, as shown in Figure 5.22. Similarly, in the case of the fourth comparison (C_{dp}), the synthetic shapes needed to model this rule are the complements of those used to model the C_{pd} comparison. The necessary shapes for the C_{dp} comparison are given in Figure 5.23.

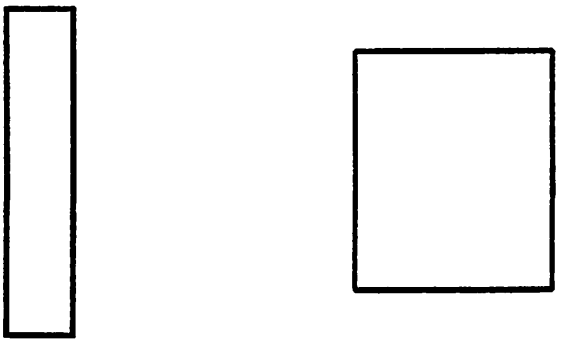


Figure 5.22: Synthetic layer for C_{pd} comparison.

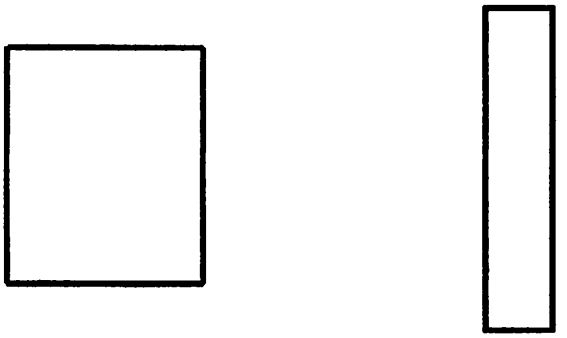


Figure 5.23: Synthetic layer for C_{dp} comparison.

The synthetic shapes needed for these two cross-layer comparisons *cannot be determined statically*, since they differ as a function of the placement of the MOSFETs. The synthetic shapes therefore cannot be stored in the MOSFET master. For this scheme to work, an algorithm would have to be invented for dynamically generating layers like POND, for all the layout elements that contain POLY or NDIF, as a function of the positions of the elements.

Addressing terminal merging in a single-layer framework presents more problems. As shown previously, terminal merging cannot be achieved unless cross-layer rules are suppressed. In the example presented in Figure 5.24, the two MOSFETs from Figure 5.21 have been connected on NDIF. Several rules must be suppressed to achieve full merging; for example, the POLY-NDIF rule between edges a_3 and b_1 must be suppressed.

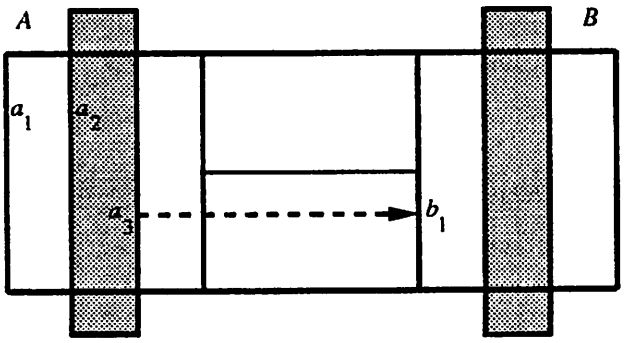


Figure 5.24: Edge suppression for two connected MOSFETs.

The conclusion that the $a_3 - b_1$ rule can be suppressed follows from the fact that A has a terminal on the same layer and in the same net as b_1. In other words, this POLY-NDIF rule is suppressed by considering the NDIF layer. A single-layer algorithm must model cross-

layer rules via synthetic layers such as POND. Hence, in this example, the edge comparisons on the POND layer would have to be conditionally suppressed as a function of connections on the NDIF layer. It is inherently cumbersome to propagate information across layers in a single-layer framework. Accordingly, the extensions necessary for merging would greatly complicate any algorithm that performs constraint generation through a set of single-layer subproblems.

Even if the above problems could be efficiently solved, a single-layer model is still undesirable, due to the amount of data required. Consider the problem of finding the limiting constraint between two elements, each with shapes on the same three layers. The number of edge comparisons is quadratic in the number of layers; i.e., nine comparisons are necessary. Mapping the cross-layer comparisons to synthetic layers still requires nine comparisons. However, the amount of data is increased in a quadratic way, which could be a severe limitation for large designs.

The rectangle-based model does not readily accommodate the polygonal shapes that are prevalent in the BLM cell abstractions; an edge-based model is more appropriate. The polygons could be sliced into rectangles, but this would have to be done twice, because the best rectangular decomposition for horizontal constraint generation is the complement of the best decomposition for vertical constraint generation. The generation of corner constraints is not addressed in [21], and transitive spacing rules are required. The extension proposed in [55] does not address these issues either.

5.6 SPARCS Plane-Sweep Algorithm

The SPARCS PPS algorithm, which was first described in [13], differs markedly from the other plane-sweep algorithms.[7] The differences are briefly mentioned here, then described in detail as the chapter progresses.

The PPS algorithm is multi-layer and edge-based, rather than single-layer and/or rectangle-based. Several of the problems described in the previous subsection are accordingly eliminated. The polygonal shapes present in the cell model are treated as such, without abstracting them to minimal enclosing rectangles, etc. Hence the full detail of the BLM cell model is exploited. The algorithm supports terminal merging to an arbitrary degree, unlike

[7] Actually, the PPS algorithm was developed with knowledge of [50], but independent of the work described in [21].

Gen(k); thus, layout density is not sacrificed by any artificial limits on mergeability. Also, the spacing rules may be non-transitive.

Corner constraints, which must be generated to insure correct results, are mentioned only in passing in [21] and not at all in [55]. The inclusion of corner constraints and multi-layer merging has a major impact on constraint generation. The combination of corner constraints and terminal merging requires the introduction of a new event type. That is, the PPS algorithm employs *three* event types instead of the two types used in the other algorithms. The need for a third event type, as well as the need for ordered processing based on edge-type values, are each described later in this chapter.

5.7 MIN Event Processing

Most of the processing in the PPS algorithm occurs when edges enter the edge structure; that is, when they become active. The processing at these MIN events depends on the types and polarities of the edges. For a given edge E, there are three factors to consider: the direction of `walk(`E`)`, the characteristics of the edges that E generates constraints to, and the criteria for terminating the walk operation. In the following subsections, the MIN-event processing of each edge type is considered in terms of these factors.

5.7.1 PF Edge Processing

As noted in the previous chapter, `pf` edges are the edges of protection frames that are not adjacent to connected terminal frames from the same element and on the same layer. External edges from terminal frames that are not connected are also of type `pf`.

For all edge types, the direction of the walk operation is away from the interior of the element. Thus, a `max pf` edge E is processed by searching the edge structure in the direction of increasing coordinates using the `walk_up()` function; the `walk_down()` function is used for a `min pf` edge.

A `pf` edge generates constraints to opposite-polarity edges only. During the walk operation, `tf` and `pf` edges are examined. There is no need to consider `btf` edges, because a `pf` or `tf` edge from the same element will always be encountered before the `btf` edge is reached. Whenever an appropriate `tf` or `pf` edge is reached, a constraint of value S_{ij} (S_{ij} is the nominal spacing rule between layers i and j) is generated between that edge and E.

Stopping Criteria

In the example given in Figures 5.14-5.18, all geometries are protection frames on the same layer. The stopping criterion is simple in this (unrealistic) case. The walk operation for any edge E can be terminated as soon as an edge E_j which leads to a constraint is reached. That is, the operation terminates as soon as an opposite-polarity edge from a different instance is found. This is true because any instances further along in the direction of the walk operation are constrained by the partner edge(s) of E_j. The **partner** of edge E, denoted $p(E)$, is the edge that spans the same interval as E, is from the same shape, and of opposite polarity. For example, if E is the left-hand edge of a rectangular shape, $p(E)$ is the right-hand edge. For polygonal shapes $p(E)$ may be a set of edges.

When protection frames are present on multiple layers, `walk(E)` must continue past the first opposite-polarity edge on the same layer as E. In Figure 5.25, instance B has protection frames on three layers, all of which are related to layer L_1, the layer of the single protection frame of instance A. The limiting constraint may be due to *any* of the frames of B; hence walk_up(a_2) cannot terminate at edge b_1; it must at least continue past all of the `min` edges of B.

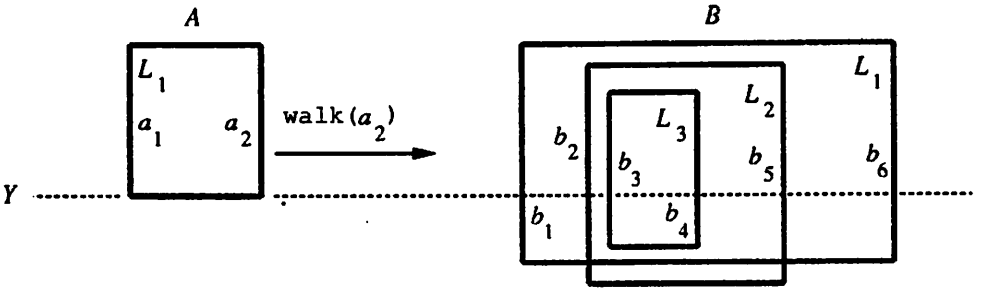


Figure 5.25: Stopping criterion for `pf` edges.

Several means exist for terminating `walk(E)` in cases like that given in Figure 5.25. The simplest method is to search until a *same-polarity* edge E_s on the same layer, but from a different element, is reached. The walk may terminate at E_s, because any edges further along in the edge structure that would generate constraints to E will be constrained by E_s. For example, in Figure 5.25, walk_up(a_2) can terminate at edge b_6: any edges related to edges on L_1 that are further to the right of b_6 will be constrained by b_6, and thus it is not necessary to constrain them with respect to a_2.

A more aggressive strategy is possible if extra information is stored, on a per-edge basis, regarding the layers that interact with layer L. Let $\mathcal{L}$ be the set of all layers that have design-rule relationships with respect to layer L. Then, the walk can be terminated according to the above criterion, or as soon as an opposite-polarity edge on each layer in $\mathcal{L}$ is encountered. Assume that $\mathcal{L}$ is comprised of the three layers present in instance B in Figure 5.25; then, `walk(`a_2`)` can terminate once edge b_3 has been reached.

Realistic situations are not restricted to `pf` edges; `tf` and `btf` edges are present as well. Fortunately, the simple termination condition applies in general, *provided that E_s is required to be a* `pf` *edge.* It is incorrect to allow a `tf` or a `btf` edge to terminate the walk operation of a `pf` edge, as exemplified in Figure 5.11. The aggressive strategy described above could be modified to work when all three edge types are present. However, the simple strategy is used instead; since some bookkeeping and storage overhead is incurred in the aggressive method, it is not clear that a net savings would be realized by it.

There is one additional circumstance under which the walk operation of a `pf` edge can be terminated; namely, the walk may terminate if an *opposite-polarity* `pf` edge, on the same layer, and from the *same* instance is found. This can occur when a protection frame has a concave region, or when an instance has more than one protection frame on a given layer. Figure 5.26 illustrates the first case. The `walk_up(`a`)` operation can stop when b is reached, since any shapes in the notch will have been caught by then and since any shape further to the right of c will be constrained by c. The second case is illustrated in Figure 5.27. Edge a_2 does not need to be processed beyond edge a_3, by the same argument as the concavity case.

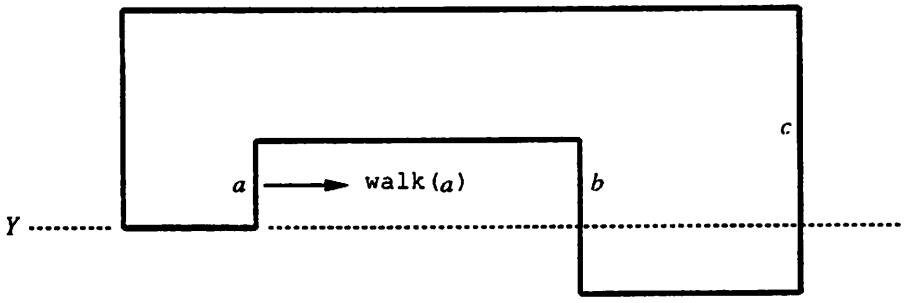


Figure 5.26: Walk termination, instance with notch.

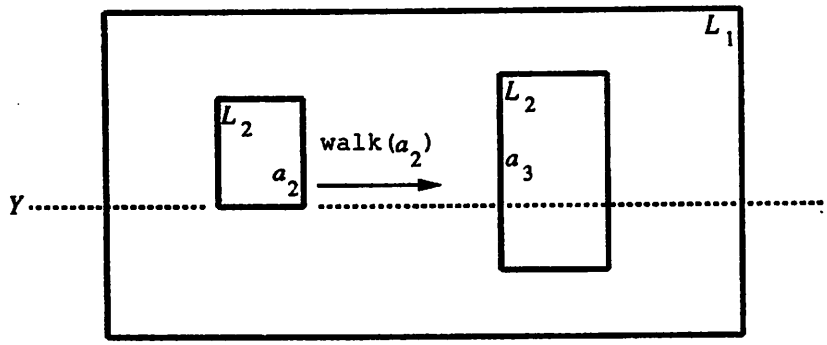


Figure 5.27: Walk termination, instance with multiple protection frames.

5.7.2 TF Edge Processing

External terminal-frame edges, i.e., those that are not adjacent to opposite-polarity protection-frame edges from the same instance and the same layer, are `tf` edges. Edges of type `tf` are processed as follows.

During the walk operation, a `tf` edge nominally sees opposite-polarity edges (with one exception). A `tf` edge behaves with respect to a `pf` edge as if it were a `pf` edge. A `tf` edge behaves with respect to a `tf` edge from a *different* net just as if both edges were `pf` edges. In these two cases, opposite-polarity edges are compared, and a constraint of value S_{ij} is generated between the two edges when their layers are related. As in the case of `pf` edges, same-polarity edges are of no consequence in these two cases. When a `tf` edge sees an opposite-polarity `tf` edge from the *same* net and on a compatible layer *no constraint is generated*, which thereby enables terminal merging. Any layer is compatible with itself, hence two same-net, same-layer terminals are always mergeable.

As described in the previous chapter, `tf` edges also generate constraints to `btf` edges on the same layer, in the same net, and with the *same* polarity. Hence the walk operation must also examine same-polarity `btf` edges when a `tf` edge is being processed. A `tf-btf` pair always generates a constraint of value zero, to limit the merge of the two elements to the legal region. For convenience, a constraint generated from this relationship will be referred to as a δ_T constraint.

Stopping Criteria

In cases where there are no `btf` edges, the stopping criteria for a `tf` edge are very similar to those of a `pf` edge. Assuming that there are no `btf` edges, the walk operation typically terminates when a same-polarity `pf` edge on the same layer is encountered. A `tf` edge from a different net *does not* suffice, because of the merging considerations illustrated in Figures 4.18 and 4.19. Opposite-polarity edges from the same instance also terminate the walk operation, as described for `pf` edges.

If `btf` edges *are* present the stopping criteria are more involved. In addition to the δ_T constraint that is generated between a `tf`-`btf` pair, the `tf` edge must be made invisible (hard-masked) with respect to all other edges of the element that the `btf` belongs to. This masking operation is necessary if full merging is to occur, as described in Chapter 4.

When the walk operation is performed and such a `btf` edge is found, the walk terminates at that point. Furthermore, no other constraints to the element that owns the `btf` can be allowed. This scheme is readily implemented. During the walk operation of the `tf` edge, the constraints that are generated are tentatively placed in a buffer. If the walk termination is due to a `btf` edge, any buffered constraints involving the element that owns the `btf` are discarded. All other constraints in the buffer, plus the δ_T constraint, are then added to the graph. All of the tentative constraints are valid if the walk terminates due to a non-`btf` edge.

5.7.3 BTF Edge Processing

As described in Chapter 4, `btf` edges are those terminal-frame edges that are adjacent to opposite-polarity protection-frame edges on the same layer and from the same instance. The function of a `btf` edge is to limit the extent of the mergeable region, and to suppress the processing of the coincident protection-frame edge. It was shown in Chapter 4 why both of these functions are needed to realize complete, and legal, terminal merging.

A `btf` edge exists only if the terminal it belongs to is connected. As a result, a terminal-frame edge cannot be recognized as a `btf` edge in the instance master. Instead, `btf` edges are recognized when they are added to the edge structure. When a connected `tf` edge is inserted into the edge structure, the other active edges with the same C_{loc} are examined. If an opposite-polarity `pf` edge from the same instance and on the same layer is found, the `tf` edge becomes a `btf` edge and the `pf` edge is soft-masked. An analogous

check is performed when pf edges are inserted. This masking is static in the sense that it does not occur during the walk operation. The parts of the pf and the tf that do not overlap are processed normally. For example, if the C_{max} coordinate of the masked pf edge is greater than that of the btf edge, the pf must be un-masked when the btf leaves the edge structure.[8] Figure 5.28 depicts these two operations, which are readily triggered by the scanline. This example illustrates another advantage of perpendicular scanning. The protection-frame edge in Figure 5.28 is effectively partitioned into three sub-edges, without ever actually dividing the edge. If shadow-propagation were used in this example, the edge would have to be explicitly partitioned into three sub-edges in the shadow data structure. Each of the three explicit sub-edges would have to be carried along in the shadow until each became covered by some other edge or edges.

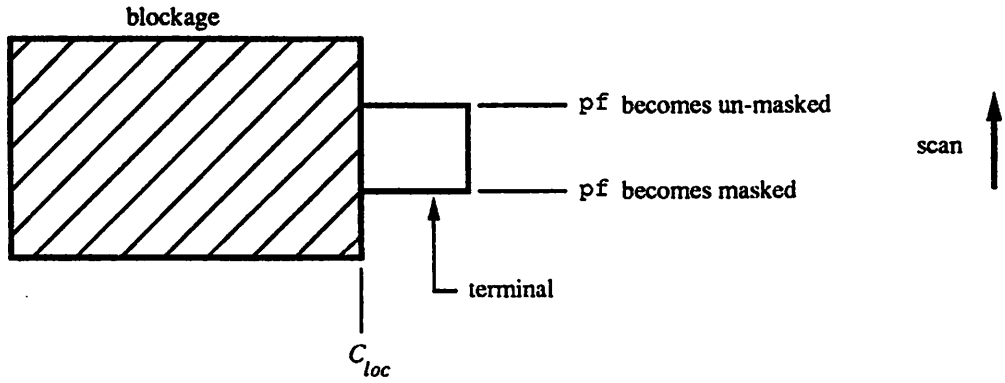


Figure 5.28: Un-masking of a pf edge when btf leaves edge structure.

Unlike tf and pf edges, btf edges generate constraints with respect to *same-polarity* edges. The direction of the walk operation for a btf edge is thus opposite to that of a pf or tf edge. That is, if btf edge E is a min edge walk_up(E) is performed; if E is a max edge walk_down(E) is performed (Figure 5.29).

A btf edge generates constraints to same-polarity, same-net, same-layer tf edges *only*; these so-called δ_T constraints always have value zero.

Stopping Criterion

Both tf and pf edges do need to be processed once a same-polarity, same-layer pf edge from a different element is reached; in other words, once the "far side" of an appropriate

[8] Un-masking is described in more detail in Section 5.8.

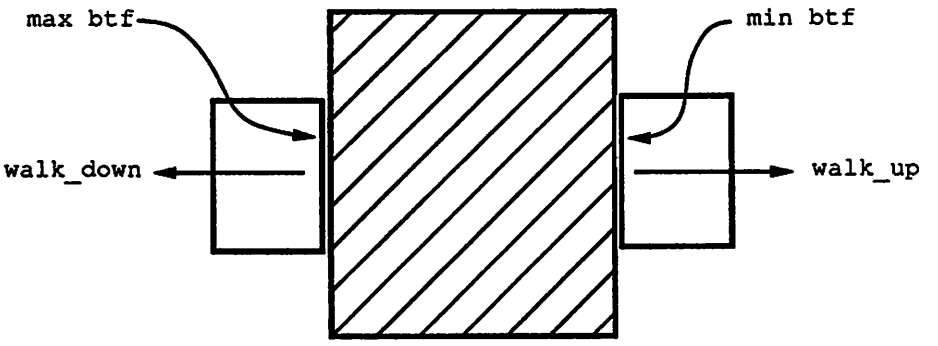


Figure 5.29: "Backwards" direction of walk operation for **btf** edges.

neighboring element is reached. This basic criterion also applies in the **btf** case. A **min btf** edge E is processed by the **walk_up()** operation; thus an edge on the far side of an element encountered during the walk must be of opposite polarity, i.e., a **max**-polarity edge. The edge E_s that terminates the walk operation is therefore the first same-layer, *opposite*-polarity **pf** edge reached during **walk**(E). Constraints to elements further along in the walk direction are captured by E_s, according to the arguments given previously. An example is presented in Figure 5.30.

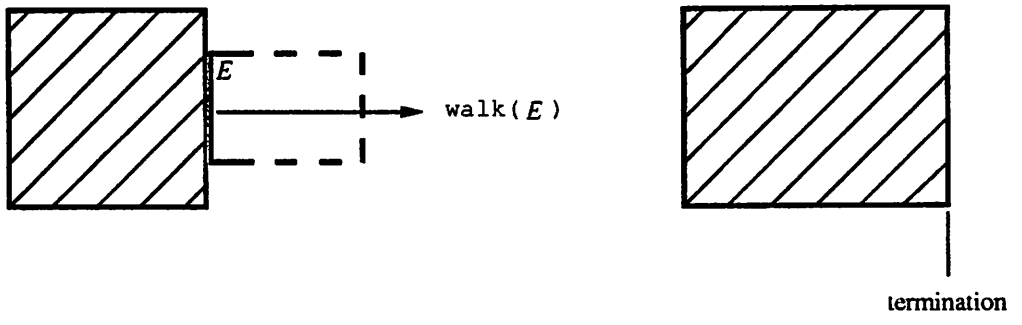


Figure 5.30: Stopping criterion for a **btf** edge.

In the previous subsection, it is stated that the walk operation for a **tf** edge can terminate once a **btf** edge is reached and the associated δ_T constraint is generated. It might be expected that the **tf**-**btf** relation is symmetric, and that the walk of a **btf** edge should likewise terminate as soon as a same-net, same-layer, same-polarity **tf** edge is reached. This *is not* the case, however.

In the example shown in Figure 5.31, assuming a horizontal compaction step, **walk_down**(c_2) reaches **tf** edge b_2 first. This leads to a δ_T constraint between wire B and

instance C. It is incorrect to terminate `walk_down`(c_2) at this point, because C and A must be constrained as well. A constraint is needed between A and C because there is no constraint between A and B, due to the fact that A and B are comprised only of `tf` edges that are in the same net. The constraint between A and C is a δ_T constraint, like the B-C constraint. Hence both a_2 and b_2 are hard-masked by c_2.

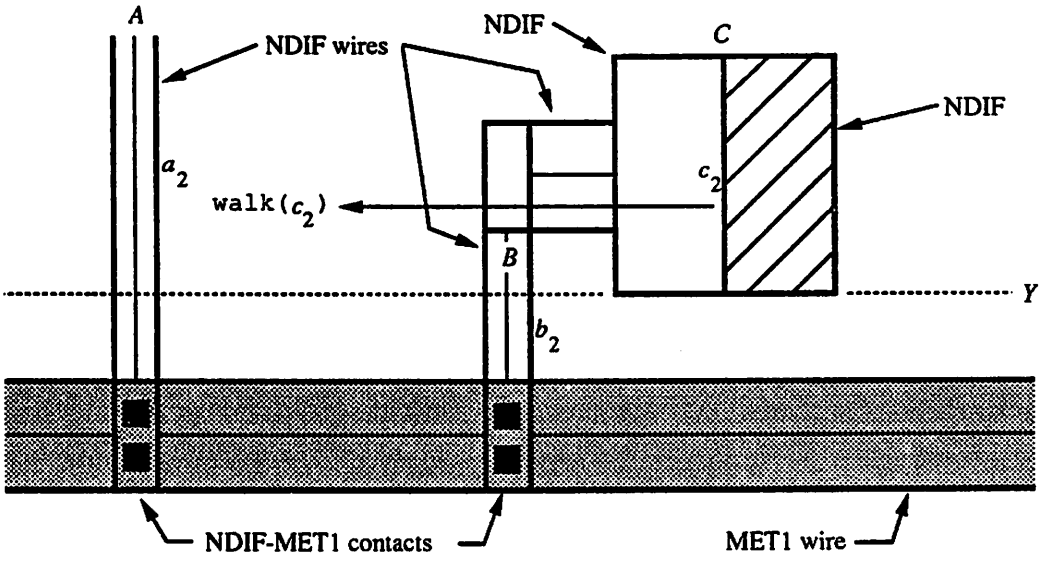


Figure 5.31: Many-to-one masking by a `btf` edge.

The example in Figure 5.31 illustrates an important difference between hard and soft masking -- that is, soft masking is one-to-one, while hard masking can be many-to-one. Soft masking occurs when a connected `tf` edge abuts a `pf` edge from the same element and layer; the `pf` edge is made invisible, and the `tf` edge becomes a `btf` edge. Exactly one edge of each type is involved; multiple edges are not possible by the definition of the BLM cell model. Hard masking is many-to-one, in that one `btf` edge can mask several `tf` edges as shown in Figure 5.31. The parts of edges a_2 and b_2 that overlap edge c_2 are constrained to C by δ_T constraints *and* are otherwise invisible to C (the non-overlapping parts do not behave in this manner as described previously). Similarly, a given `tf` edge can simultaneously be hard-masked by several `btf` edges.

5.7.4 Min Event Processing Order

At a MIN event, all edges with $C_{min} = C_e$ are activated (added to the edge structure), then processed by either `walk_up()` or `walk_down()` as just described. If the masking mechanisms were not used, then the newly-activated edges could be processed in any order. However, since masking is performed, the edges must be walked in a particular order to achieve the maximum layout density.

Hard-masking occurs between a `tf` edge and a `btf` edge. Let E_{1t} be a `tf` edge from element 1, and let E_{2b} be a `btf` edge from element 2. When E_{2b} hard-masks E_{1t}, a δ_t constraint is added between them to prevent elements 1 and 2 from over-merging, then E_{1t} is made invisible to all other edges of element 2. Element 2 could also have a `pf` edge E_{2p} with $C_{min} = C_e$. If `walk`(E_{2p}) occurs before E_{1t} is hard-masked, then a spurious constraint between E_{1t} and E_{2p} could result that artificially increases the separation of elements 1 and 2. This situation is illustrated in Figure 5.32. These undesired constraints are avoided in the PPS algorithm by processing `pf` edges last at MIN events. That is, at each C_e, all newly-activated `tf` and `btf` edges are processed first, after which the new `pf` edges are considered.

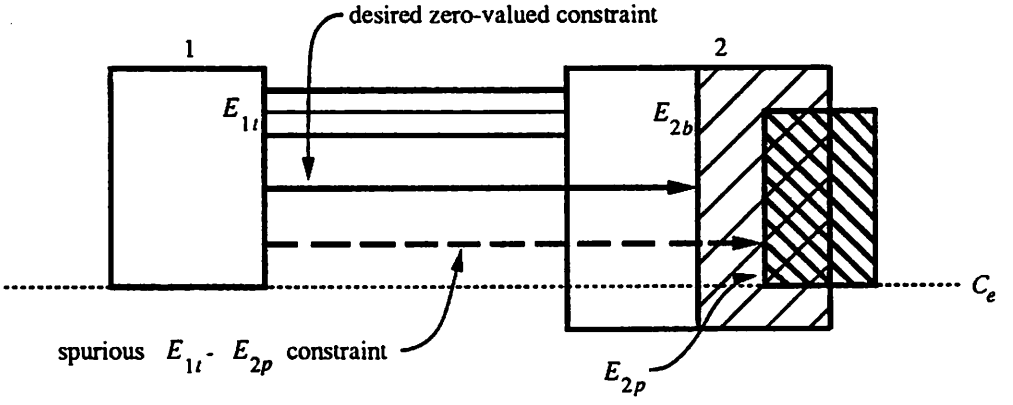


Figure 5.32: Potential spurious constraint if E_{2p} is processed first.

Figure 5.33 contains a pseudo-code description of the MIN-event segment of the PPS algorithm.

```
if (event.type == MIN) {
    edgePtr = event.firstEdge;      /* Q el't; points to an edge */
    firstEdgePtr = edgePtr;         /* first edge, this event    */
    eventCoord = edgePtr->coord;    /* current event coordinate  */

    /*  Add the new edges  */
    while (edgePtr != NIL && edgePtr->coord == eventCoord) {
        addToEdgeStruct(eventCoord, edgePtr->edge);
        edgePtr = edgePtr->next;
    }

    /*  Perform the walk operation on the edges just added  */
    /*  Process connected tfs                               */
    edgePtr = firstEdgePtr;
    while (edgePtr != NIL && edgePtr->coord == eventCoord) {
        if (edgePtr->edge->netId != UNDEF &&     /* ==> connected */
            edgePtr->edge->masked == 'N') {      /* ==> tf        */
            walkTF(eventCoord, edgePtr->edge, direction);
        }
        edgePtr = edgePtr->next;
    }
    /*  Process connected, soft-masked tfs, i.e., btfs  */
    edgePtr = firstEdgePtr;
    while (edgePtr != NIL && edgePtr->coord == eventCoord) {
        if (edgePtr->edge->netId != UNDEF &&     /* ==> connected */
            edgePtr->edge->masked == 'S') {      /* ==> btf       */
            walkBTF(eventCoord, edgePtr->edge, direction);
        }
        edgePtr = edgePtr->next;
    }
    /*  Process pfs (unless masked)  */
    edgePtr = firstEdgePtr;
    while (edgePtr != NIL && edgePtr->coord == eventCoord) {
        if (edgePtr->edge->netId == UNDEF) {     /* ==> pf */
            if (edgePtr->edge->masked == 'N') {
                walkPF(eventCoord, edgePtr->edge, direction);
            }
        }
        edgePtr = edgePtr->next;
    }
}
```

Figure 5.33: Pseudo-code for MIN event processing.

5.8 Edge Un-Masking

Both forms of masking (hard and soft) are applied to a pair of edges E_i and E_j. Masking alters the behavior of the PPS algorithm between E_i and E_j, *but only over the region that their intervals overlap.* The edges behave as dictated by their basic types otherwise. For the example shown in Figure 5.34, edge a_3 is soft-masked by a_2 over the interval (X_2, X_3). However, a_3 behaves like a pf edge for $X_1 \leq x \leq X_2$ and $X_3 \leq x \leq X_4$. One strength of the PPS algorithm is that an edge such as a_3 can be partitioned as shown

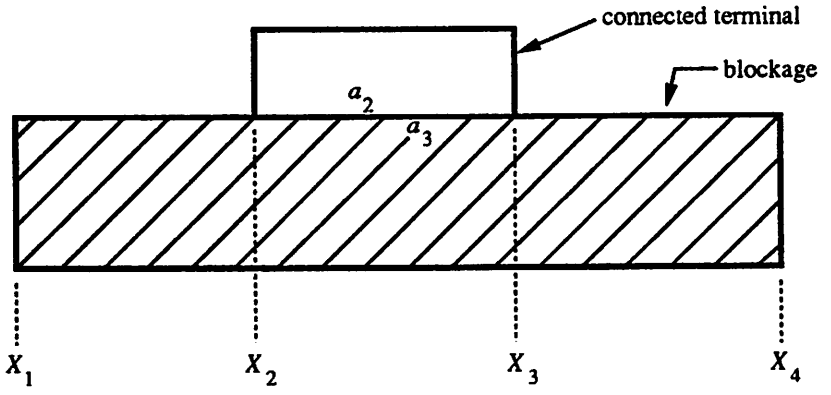


Figure 5.34: Masking as a function of intersecting intervals.

in Figure 5.34 *implicitly*, at a very low computational cost.

The PPS algorithm, as described so far, functions properly when the two edges in a masking operation leave the edge structure at the same time. In this section, the modifications to the algorithm which insure that it operates properly when the edges leave at *different* times are described. The inclusion of corner constraints alters the treatment described here. For clarity of presentation, however, corner constraints will be ignored until a later section.

If one edge in a masking pair leaves before the other, the edge that remains active must revert to its previous type. The implication of this is that the remaining edge may need to be reprocessed. Consider a horizontal compaction iteration on the example shown in Figure 5.35. Instance A is an NDIF terminal that is connected to instance B, an NMOS transistor. At event Y_1 edge a_2 is activated. When `walk_up`(a_2) executes a single constraint is generated, namely the δ_T constraint between a_2 and b_2. This constraint correctly models the spacing requirement between A and B, over the interval that a_2 and b_2 overlap. All

events following Y_1 are MAX events. If no additional processing occurs, the compacted result given in Figure 5.36 results; this layout violates the POLY-NDIF spacing rule between A and the upper extension of the gate of B.

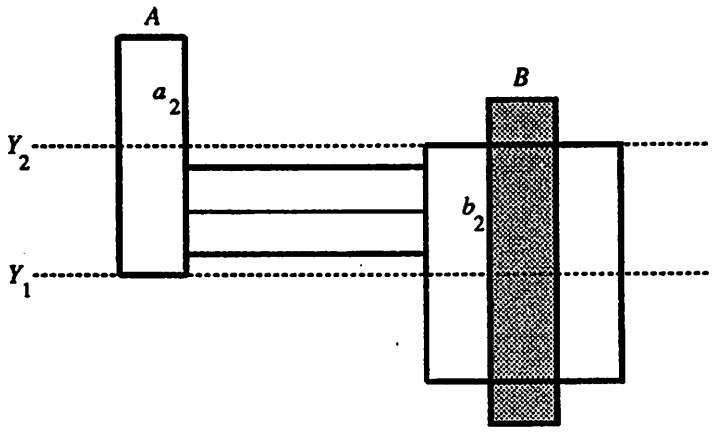


Figure 5.35: Connected terminal and transistor, before horizontal compaction.

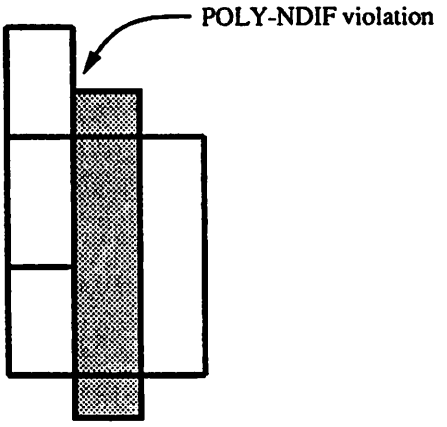


Figure 5.36: Illegal compaction of example in Figure 5.35.

At Y_2 edge b_2 leaves the edge structure: hence, from that point, edge a_2 is no longer hard-masked by b_2. As a result the sub-intervals of a_2 and b_3 above Y_2 (Figure 5.37) should be constrained according to the `tf`-edge criteria described previously. The desired constraint is pictured in Figure 5.37; with its inclusion a design-rule correct result is achieved. However, as described thus far, the PPS algorithm has no mechanism to trigger the creation of the a_2-b_3 constraint, because constraints are only generated at MIN events, and there are no MIN events following Y_1.

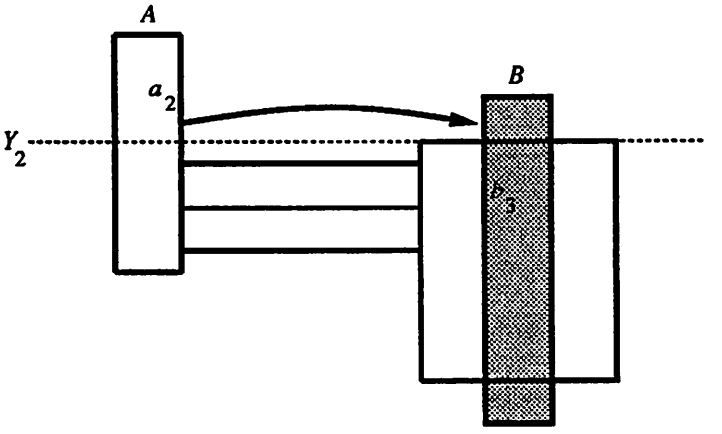


Figure 5.37: Constraint required between a_2 and b_3 above Y_2.

The solution to this problem that has been incorporated in the PPS algorithm follows from the observation that, in effect, *a new edge is created* when an edge such as a_2 is unmasked. Indeed, the constraint in Figure 5.37 is created at Y_2 if a two-step procedure is followed:

1. Unmask a_2: (a_2 becomes an ordinary `tf` edge)
2. Invoke `walk_up`(a_2). (creates the desired constraint)

This solution is correct for both types of masking. Whenever one edge in a masking pair leaves the edge structure, the remaining edge is first unmasked, then processed by the walk operation just as if it were a new edge. If both edges leave the edge structure at the same C_ϵ, no re-walking is performed.

5.9 Corner Constraints

In the examples presented to this point, constraints have been created between edges whose intervals overlap. It is also necessary to add constraints between edges whose intervals *do not* overlap, when the intervals are separated by less than a spacing rule in the direction orthogonal to the compaction direction. If these so-called **corner constraints** are not added, violations may occur in the orthogonal direction as shown in Figure 5.38. All compactors therefore compute corner constraints. Virtual-grid compactors typically compact horizontally first and vertically second, with corner constraints included only on

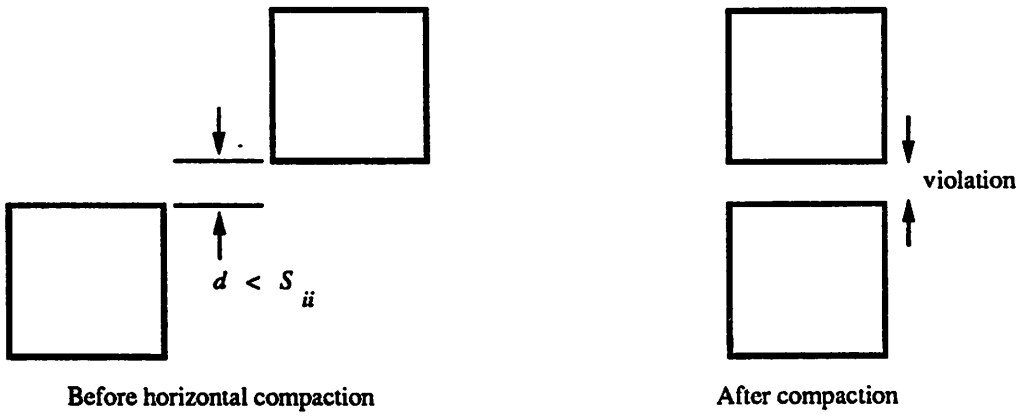


Figure 5.38: Violation due to missing corner constraint.

the second iteration [81,11,73]. This is not adequate when the layout elements are general, as they are in SPARCS. General elements may have concave regions; if corner constraints are ignored in either direction, it is possible to create overconstraints by pushing an element into a concave region that is too small for it.

Corner constraints can be caught by extending the edges of the elements appropriately. Figure 5.39 shows two simple elements that are on the same layer (L_i). In the vertical direction, A and B are separated by less than S_{ii}; hence, in horizontal compaction, a_3 and b_1 must be separated by S_{ii} to avoid a violation between a_4 and b_2. If, for example, the upper corner of a_3 were extended by S_{ii} as shown in Figure 5.39, `walk_down`(b_1) would see a_3 and the necessary constraint would be generated. In the single-layer case, corner constraints from a given edge E to all other edges can be found by either extending E or by extending all other edges, but not both.[9] If all edges were extended on both ends, then false constraints would be created between edges separated by $S_{ii} < d \leq 2S_{ii}$, whereas it is only necessary to add a corner constraint between edges separated by $d \leq S_{ii}$.

There are a number of alternative algorithms for edge extension. One dynamic mechanism is to alternatively extend or not extend edges, according to whether they are being walked or not. However, static methods, where the edge extensions do not need to be altered during the execution of the algorithm, are simpler and thus more efficient. A second possibility is to extend all edges by $S_{ii}/2$. The disadvantage of this approach is that *both* ends of each edge then require some additional processing.

A static method that extends only one end of each edge has been developed for

[9]The phrase "all edges" is used loosely here. It is only necessary to extend edges that form convex corners.

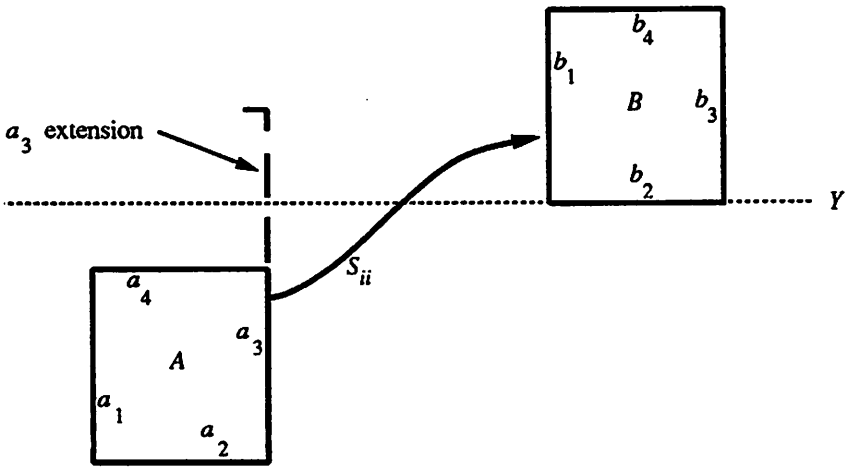


Figure 5.39: Corner visibility via edge extensions.

the PPS algorithm. The method follows from the observation that corner constraints can be captured by either extending the upper ends of all edges, or the lower ends of all edges, but not both. Figure 5.40 shows the first case. In the plane-sweep algorithm, extending the lower end of an edge corresponds to putting it into the edge structure *earlier* than normal, while extending its upper end corresponds to removing it from the edge structure *later* than normal.

In the PPS algorithm of SPARCS, the method of extending the upper end of each edge has been selected. Each appropriate edge is extended by the corresponding per-layer maximum spacing rule. The **maximium spacing rule** for layer L_i, denoted S_I, is the largest rule involving layer L_i (not the overall largest spacing rule). Extension by S_I guarantees that any edge E_i that is a candidate for a corner constraint is seen during `walk`(E_j). However, a constraint is necessary only if E_i and E_j are separated in the orthogonal direction by S_{ij} or less. As a result, a constraint results if

$$C_e - C^i_{mid} \leq S_{ij} \tag{5.1}$$

as shown in Figure 5.41.

Only those edges that form convex corners are extended: it is not necessary to extend the edges that meet at concave corners. This criterion applies to each edge-type individually. The edges that form convex and concave corners are referred to as **convex** and **concave** edges, respectively.

When an edge is extended, its C_{max} coordinate is increased by S_I. The original

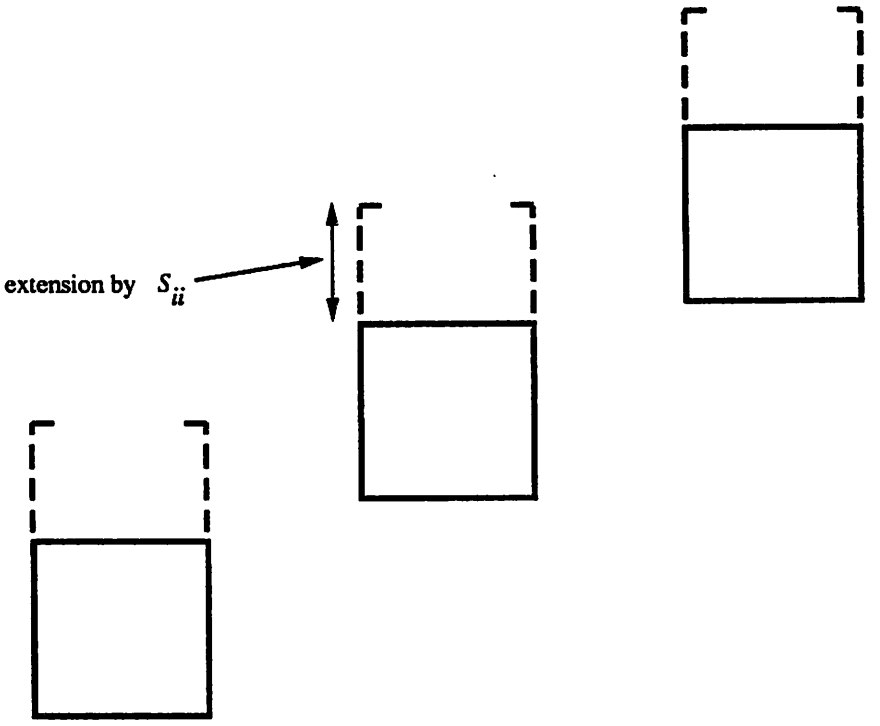


Figure 5.40: Extension of upper corners only.

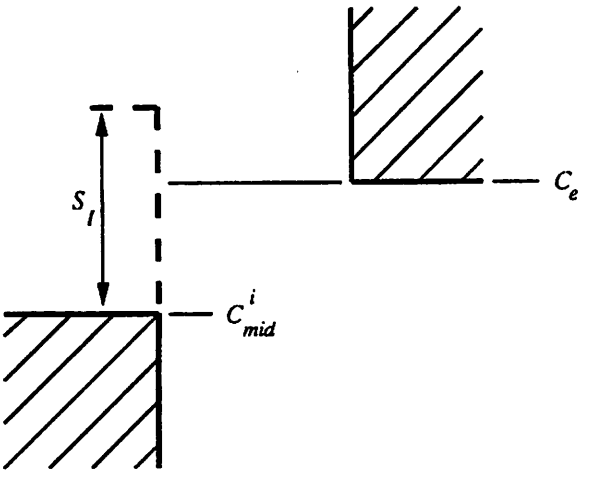


Figure 5.41: Constraint needed if $C_e - C^i_{mid} \leq S_{ij}$.

maximum coordinate, before extension, is denoted C_{mid}. The role of C_{mid} in the PPS algorithm is described in the following section.

5.10 The Need for a Third Event Type

It is stated earlier in this chapter that a third event-type is needed, due to the combination of masking and corner constraints. That statement is justified in this section.

In Section 5.8, the operation of the algorithm at MAX events is described for the case when an edge E_i, which masks a second edge E_j, leaves the edge structure before E_j. Such a case results in E_j being unmasked and then processed by the walk operation as if it were a newly-activated edge.

Corner constraints are triggered by extending the C_{max} coordinates of convex edges. As a result, it appears that any unmasking due to E_i is deferred until the scanline is S_I units beyond the actual upper end of E_i. Edge E_j thus remains masked until the end of the extension of E_i is reached, which may incorrectly suppress constraints and thereby result in design-rule errors. Figure 5.42 illustrates the problem. At $C_\epsilon = Y_2$ edge b_1 is hard-masked by edge a_2. Edge a_2 is a convex `btf` edge, and it has therefore been extended as shown. (Most of the other edges are extended as well, but their extensions are not drawn to simplify the figure.) At Y_3, edge b_1 should become unmasked. It does not, because the C_{max} coordinate of a_2 is not reached until $C_\epsilon = Y_5$. Since b_1 remains (improperly) hard-masked by a_2 between events Y_3 and Y_5, the POLY-NDIF constraint between a_1 and b_1 is never generated and the resulting layout is not legal.

The layout error occurs because b_1 remains masked beyond the interval over which it intersects a_2, the masking edge. The additional operation that is required is to unmask b_1 at Y_3, even though the C_{max} coordinate of its masking edge is not reached until Y_5. This new operation cannot be invoked unless edge a_2 causes the scanline to stop at Y_3 as well as Y_1 and Y_5; that is, a third event-type is needed.

The new event-type is termed the **MID** type. The edge coordinate that corresponds to a MID event is called the **mid** coordinate of the edge, and is denoted C_{mid}. The value of C_{mid} is the upper coordinate of the edge *before* it is extended by S_I. In Section 5.8 it was stated that the unmasking scheme described therein would be modified by the addition of corner constraints. The modification is that all unmasking occurs at MID events, rather than at MAX events. A third queue is used to schedule MID events.

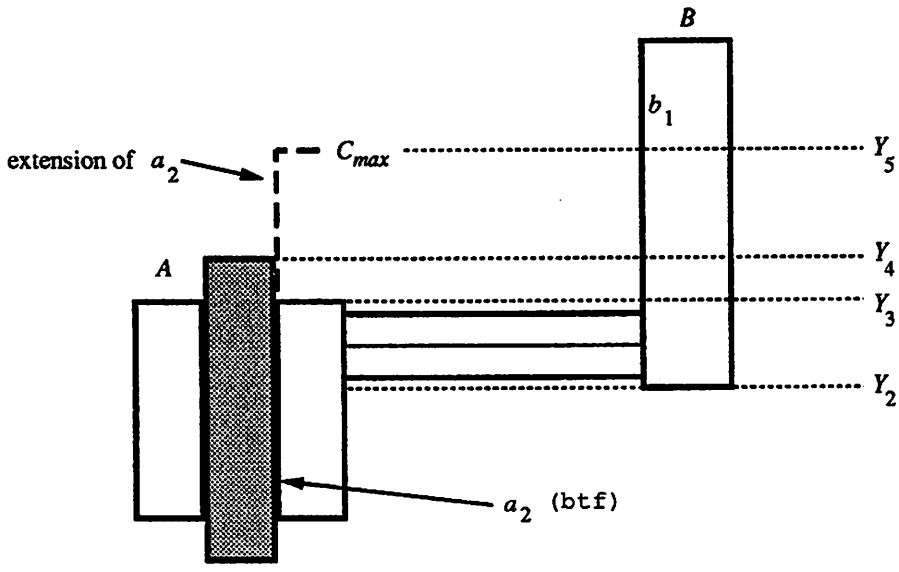


Figure 5.42: Violation due to deferred unmasking.

In principle, because MID events are used for unmasking, they are only needed for those edges that are extended and that mask other edges. Thus concave edges do not require MID events because they are not extended. A convex `pf` edge must be extended, but it does not mask other edges, unlike `tf` and `btf` edges; hence a MID event is not required. However, for implementation ease, the current version of the PPS algorithm includes a MID event for all extended edges.

5.11 Event Sequencing

All of the essential operations of the PPS algorithm have now been described. The only remaining issue is the sequencing of events, when events of more than one type must be processed at a particular C_e. The events must be sequenced such that active edges are not prematurely removed from the edge structure; if they are, necessary constraints might be omitted.

Constraints are generated during the walk operation, which occurs as a result of MIN and MID events. During MAX events edges are removed from the edge structure; no other processing occurs. If MAX events and either MID or MIN events are to be processed at the same C_e, the MAX events are thus processed last. If MIN and MID events are scheduled at the same C_e, the MIN events are processed first. In principle, MIN and MID events can

be processed in either order; this order has been chosen for minor, implementation-related reasons.

The complete PPS algorithm is summarized in the pseudo-code presented in Figure 5.43.

5.12 Performance of the PPS Algorithm

The complexity of the PPS algorithm is analyzed in this section and compared with that of the D-L algorithm. Its measured performance is presented over a range of problem sizes. In addition, it is compared with measured data from other compactors that use different constraint-generation algorithms.

5.12.1 Complexity

The complexity of the PPS algorithm can be determined by considering the complexity of each step in the following decomposition.

- Scanline traversal of the layout
- Insertion into the edge structure
- Walk operation
- Edge masking and unmasking
- Deletion from the edge structure

The appropriate measure of problem size is the number of edges in the layout, which will be denoted n. An assumption must be made regarding the arrangement of the edges in the layout. Here, it will be assumed that the model layout is rectangular with the edges evenly distributed. A horizontal or vertical line drawn across the layout is thus expected to intersect $\sqrt{n}$ edges.

Each edge has a MIN and a MAX event, and perhaps a MID event. The number of events is at most $3n$, which is $O(n)$. Once the event queues are constructed, the scanline traversal is $O(n)$ as well. The time required to construct the event queues is the time needed to sort three lists of at most n elements, which is $O(n \log n)$. The mergesort algorithm is

```
/*
 *  While (there are events)...
 *      pmin:   next Q el't in MIN event Q
 *      pmid:   next Q el't in MID event Q
 *      pmax:   next Q el't in MAX event Q
 */
while (pmin != NIL  ||  pmid != NIL  ||  pmax != NIL) {
    /*
     *  If pmin->coord == pmax->coord or pmid->coord == pmax->coord,
     *  then the deletions are done last.
     */
    if ((pmin != NIL  &&  pmin->coord <= pmax->coord)  ||
        (pmid != NIL  &&  pmid->coord <= pmax->coord)) {
        /*
         *  In a tie, do MIN events first
         */
        if ((pmin != NIL  &&  pmid != NIL  &&
             pmin->coord <= pmid->coord)  ||  pmid == NIL) {
            /*
             *  Process from the MIN list
             */
            processMinEvents(....);          /* already described */
        } else  {
            /*
             *  Process the MID events
             */
            eventCoord = pmid->coord;
            while (pmid != NIL  &&  pmid->coord == eventCoord) {
                if (pmid->edge->maskedEdge != NIL) {
                    /*  current edge does mask other edges  */
                    unmaskEdges(pmid->edge);
                }
                pmid = pmid->next;
            }
            /*
             *  Newly-visible edges are in type-specific lists.
             */
            /*  Were hard-masked tfs; now regular tfs  */
            edge = hardTFList;
            while (edge != NIL) {
                rewalkHardTF(eventCoord, edge, direction);
                edge = edge->next;
            }
```

```
            /*  Were btfs; now regular tfs  */
            edge = BTFList;
            while (edge != NIL) {
                rewalkBTF(eventCoord, edge, direction);
                edge = edge->next;
            }
            /*  Were soft-masked pfs; now regular pfs  */
            edge = PFList;
            while (edge != NIL) {
                rewalkPF(eventCoord, edge, direction);
                edge = edge->next;
            }
        }
    } else  {
        /*
         *  MAX event; delete edges from edge structure
         */
        eventCoord = pmax->coord;
        while (pmax != NIL  &&  pmax->coord == eventCoord) {
            deleteFromEdgeStruct(pmax->edge);
            pmax = pmax->next;
        }
    }
}
```

Figure 5.43: Pseudo-code for PPS algorithm.

used SPARCS, because it runs in $O(n \log n)$ time regardless of the initial ordering of the data [69]. All scanline processing can thus be accomplished in $O(n \log n)$ time.

Operations involving the edge structure are affected by the data structures used in its implementation. The edge structure of the PPS algorithm is implemented via a doubly-linked list. A doubly-linked list is not the best-performing data structure. It was selected nevertheless because it is relatively easy to implement and debug. A more efficient edge-structure implementation could be substituted if desired.

The edge structure is sorted by the C_{loc} coordinates of the edges. Insertion of a new edge is bounded by the number of edges already in the list. In the worst case, insertion is thus $O(\sqrt{n})$ for a single edge, and $O(n^{1.5})$ for all edges. Substitution of a tree for the doubly-linked list would reduce these bounds to $O(\log n)$ and $O(n \log n)$, respectively.

The walk operation occurs once for each edge at its MIN event. A given edge can also be processed again by the walk operation following a MID event. The total number of walk operations is $O(n)$, since it is bounded by the number of MIN events plus the number of MID events.

The expected complexity of a single walk operation is difficult to quantify. In a case comprised of only blockages that are all on the same layer, the walk operation is $O(1)$ because the first edge encountered satisfies the termination criterion. The inclusion of terminal merging implies that `walk(`E`)` generally visits several edges, hence the complexity is not $O(1)$. For the pathological case where all the active edges are same-layer, same-net `tf` edges, the walk operation requires $O(\sqrt{n})$ time.

In a representative case `walk(`E`)` is expected to visit a number of edges, but much less than the total number of active edges. This argument leads to the supposition that an average walk operation should require much less than $O(\sqrt{n})$ time when $\sqrt{n}$ edges are active. The total time for all walk operations should therefore be much less than $O(n^{1.5})$ in a typical case.

Edges are masked upon insertion or during the walk operation. Masking at insertion (soft masking) requires that only those edges with the same C_{loc} coordinate be examined; it is thus reasonable to regard the operation as $O(1)$. Masking during the walk operation (hard masking) does not require any additional searching over that of the walk operation itself. One `btf` edge can hard-mask several `tf` edges; this information is stored with the `btf` edge in a linear list. Only in highly unusual cases would a `btf` edge mask more than one or two `tf` edges. The number of `tf` edges masked by a particular `btf` edge is

therefore assumed to be bounded by a small constant. By these arguments, and since the masking information is explicitly stored with the edges involved, an unmasking operation requires $O(1)$ time. On a per-edge basis, it is thus reasonable to assume that all masking operations can be performed in constant time.

Deletion from the edge structure consumes constant time, since the MAX queue elements contain pointers to the edges to be removed. It follows that the total MAX-event processing is $O(n)$.

The overall time-bound of the PPS algorithm is determined by summing the time needed to execute each of these steps. However, the masking and deletion components can be ignored since they have linear complexity are thus are not significant compared to the others. As a result, the worst-case performance is given by

$$\begin{aligned} \text{Total time} &= t_{scanline} + t_{insertion} + t_{walk} \\ &= t_s + t_i + t_w \qquad (5.2) \\ &= n \log n + n^{1.5} + n^{1.5} \\ &= O(n^{1.5}). \qquad (5.3) \end{aligned}$$

Comparison with the D-L Algorithm

The complexity of the most general version of the D-L algorithm, $Gen(k)$, is given by

$$\begin{aligned} T_{DL} &= O(n(k + \log n)) \\ &= n \log n + nk. \qquad (5.4) \end{aligned}$$

The D-L algorithm's rectangle structure, which corresponds to the edge structure of the PPS algorithm, is implemented via a balanced binary tree. When this data structure is used, t_i is $O(n \log n)$. The first term in Equation 5.4 thus corresponds to the first two terms in Equation 5.2. The second term in Equation 5.4 is due to the walk operation. The value nk results because merging is limited to k neighboring rectangles in the D-L algorithm. The PPS algorithm does not limit merging. If the D-L algorithm did allow unlimited merging, then k would become $\sqrt{n}$ in the worst case.

The D-L and PPS algorithms thus exhibit the same walk-operation complexity when the amount of merging is the same. The D-L algorithm is more efficient in the

insertion-time component, but this is only due to the simpler data structure used in the PPS algorithm.

5.12.2 Measured Performance

The performance of the PPS algorithm has been measured on a set of eight similar examples that range in size from 408 elements to 1881 elements. Each example in succession is about 200 elements larger than the preceding one. The examples were constructed by replicating a 16-transistor standard cell called **c16_2** from the benchmark session of the 1986 MCNC Internation Workshop on Symbolic Layout and Compaction [1]. The smallest example has two copies of the cell, and the largest has nine copies. The examples were compacted flat, and the times were recorded for two spacing iterations, one per direction. Figure 5.44 shows these measurements; as expected, the PPS algorithm's performance is bounded below by $O(n \log n)$ and above by $O(n^{1.5})$. The times in the graph are in DEC VAX 8650 CPU-seconds.

Comparison of Measured Performance

The runtime performance of the PPS algorithm cannot be compared to the D-L algorithm, because measured data is not provided in [21].

A direct comparison with a constraint generator based on shadow propagation is not available. The shadow-propagation algorithm was implemented in an early phase of this project. The complications encountered in adapting it to the generic layout model led to the investigation of alternatives, and finally to the development of the PPS algorithm. The PPS generator was found to out-perform the shadow-propagation generator and development of the shadow-propagation generator was terminated. Since the PPS generator was subsequently refined further, a comparison is not valid.

A crude, indirect comparison can be made using results from the compaction benchmark session of the 1987 International Conference on Computer Design [9]. Four compactors were included in the session, including SPARCS. The Symbolics compactor [73] and the MACS compactor [18] also participated; MACS uses shadow propagation for constraint generation, and the Symbolics compactor uses the Most Recent Layers algorithm [11], which is an implementation of shadow propagation for virtual-grid compaction. Table 5.1 gives the total runtimes for each program on the two flat examples that all three

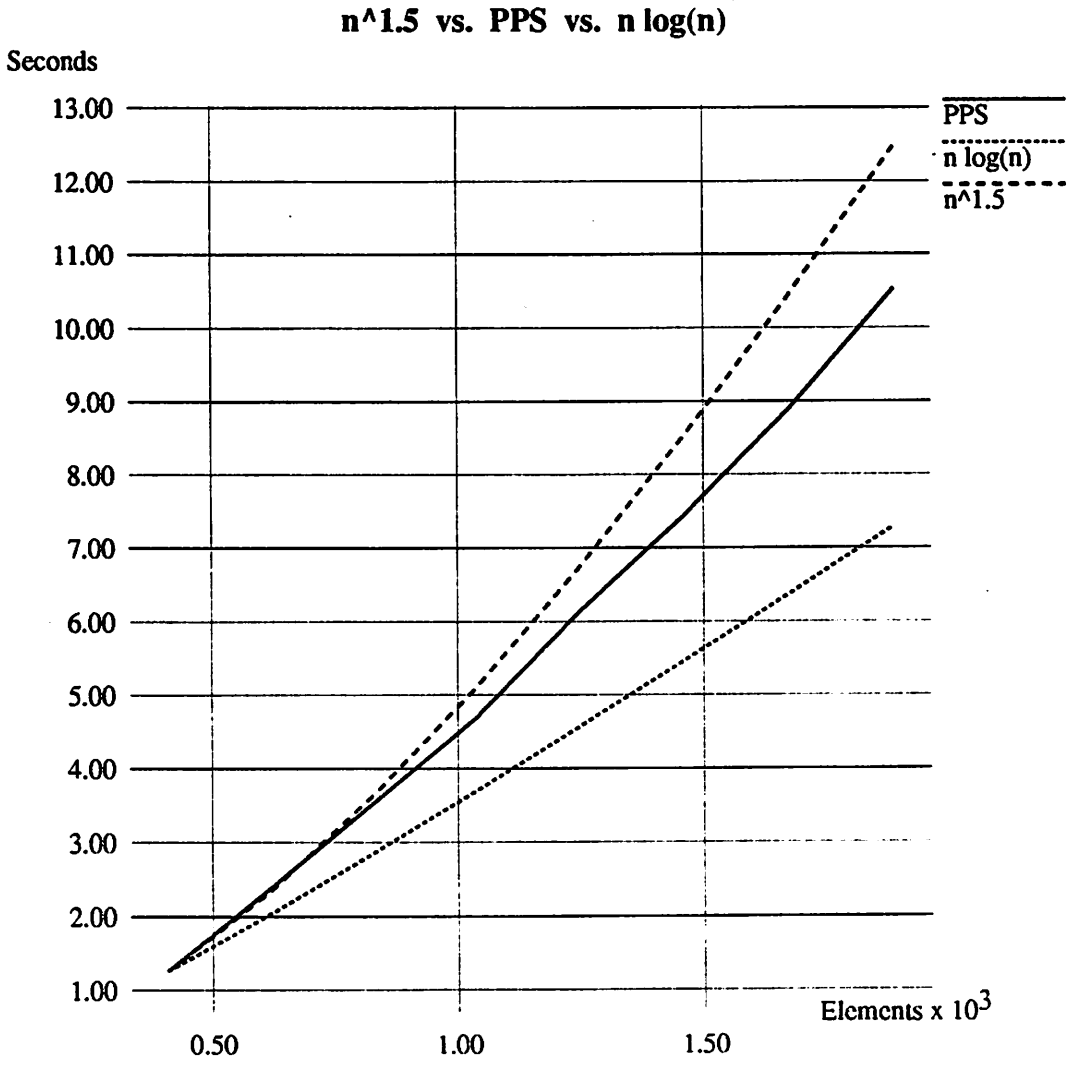


Figure 5.44: Measured performance of the PPS algorithm.

programs compacted. Times are in DEC VAX 8650 CPU-seconds.

Program	Example	Time	Example	Time
MACS	afa	9	afakr	5
SPARCS	afa	11	afakr	8
Symbolics	afa	5	afakr	5

Table 5.1: Results from [9].

The times listed in Table 5.1 are totals which include at least two compaction iterations, well generation, and CIF generation.[10] Simple programs were used in the case of SPARCS for well generation and CIF generation; these account for between 25 and 30% of the runtimes quoted above for SPARCS.

The Symbolics compactor performs two iterations [73], whereas SPARCS performed 5 iterations for the **afa** example and 3 iterations for the **afakr** example. On a per-iteration basis, SPARCS thus performs quite well compared to the Symbolics compactor. The number of iterations performed by MACS is not stated in [18]; however, it is unlikely that MACS performed as many iterations as SPARCS. It is reasonable to conclude from this data that the PPS algorithm, operating on a type-free layout model, compares well with the shadow-propagation algorithm operating on the standard layout model.

The results from this benchmark exercise are presented in more detail in Chapter 9.

5.13 Summary

Efficient constraint generation is a design goal for any layout compactor. The use of the low-level BLM layout model, and the adoption of an aggressive merging scheme, both complicate the constraint-generation problem. It is thus especially important to develop a fast constraint generator in this situation.

To be efficient for a wide variety of layouts, geometric pruning must be employed in constraint generation. The shadow-propagation algorithm does so, and it can be implemented to operate efficiently in an asymptotic sense. However, the data structures that are required are cumbersome and expensive to manage. The perpendicular-scanning approach can be implemented to perform geometric pruning at least as well as the shadowing ap-

[10]The CIF language is a textual language for describing physical layouts.

proach, but with considerably less data-structure overhead. As a result, its performance is better than shadow propagation (assuming the same layout model), even though its asymptotic behavior is similar.

The PPS algorithm has been developed to accommodate corner constraints, terminal merging, non-transitive rules, and a generic, edge-based layout model. The results presented in this chapter, as well as results which appear later in this dissertation, indicate that the PPS algorithm competes well with other approaches that use more restrictive, high-level layout models.

Chapter 6

Constraint Solution

6.1 Introduction

The constraint generator produces a weighted, directed graph, which is a global model of the layout in the direction of compaction. Graphs are typically denoted $G(V, E)$, where V is the set of vertices and E is the set of edges. The graph G is analyzed to determine the new locations for the layout elements. This processing is often called **solving** the graph. In this chapter, the terms **vertex** and **node** will be used synonymously, as will the terms **constraint**, **edge**, and **arc**.

The graph-solving algorithms used in SPARCS are described in this chapter. A novel characteristic of these algorithms is their use of **event-driven selective-trace** techniques [14]. Event-driven selective-trace algorithms minimize redundant processing via a scheduling mechanism whereby an element is processed only if a related, neighboring element changes state. In this context, element refers to a node in the graph, and a change in a neighbor's state corresponds to a change in position of a fanin or fanout node. These algorithms (as well as prototype implementations of them) were originally suggested by Newton [58]. Some improvements and enhancements for higher efficiency have been contributed in the course of this project.

Chapter 6 begins with a problem statement, followed by a description of constraint types and their effect on algorithms for graph analysis. The SPARCS longest-path, overconstraint-detection, and slack-distribution algorithms are then described in turn. The chapter concludes with a summary.

6.2 Problem Definition

The primary objective in layout compaction is to minimize the area of the layout. One-dimensional compactors minimize area indirectly, by minimizing the horizontal pitch and vertical pitch in successive spacing iterations. The primary objective in one-dimensional compaction can therefore be stated as

$$\text{minimize } t - s, \tag{6.1}$$

where s and t are the coordinates of the source and sink nodes of the constraint graph, respectively.

The primary objective can be realized by performing a single-source longest-path analysis on G [51]; this analysis will be denoted `lp(`s`)`.[1] In a horizontal (vertical) compaction iteration, the longest-path analysis from the source node gives the minimum, i.e., left-most (bottom-most) legal location for each node with respect to the source node. In general there are several paths from s to any particular node v. At least one, but usually not all of the paths limits the separation of s and v. A limiting path is a **longest** or **tight path**; the other paths are referred to as **slack paths**. The edges that form the longest paths define a so-called longest-path tree. Figure 6.1 illustrates the results of `lp(`s`)` on a simple constraint graph.

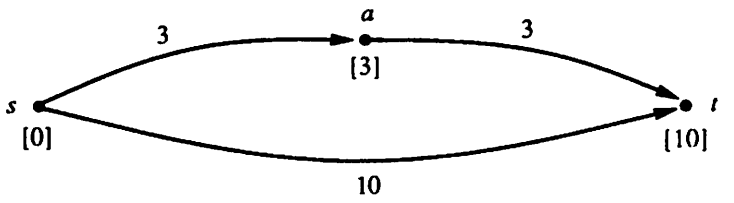


Figure 6.1: Result of `lp(`s`)`.

Since the minimum pitch is determined by the longest path(s) from s to t, any node on a longest s-t path must be located at its minimum position to realize a minimum-pitch layout. This is not true for the other nodes; they may occupy a range of locations without increasing the size of the layout. To determine the upper bound on the ranges of these **slack nodes** a second longest-path analysis is performed. This analysis is also a single-source analysis, but it starts from the sink node and is likewise denoted `lp(`t`)`. For

[1] The longest-path and shortest-path analyses on a directed graph are duals of one another [74].

lp(t), the position of the sink node is taken to be that determined from lp(s). The output of lp(t) is the *maximum*, i.e., right-most (top-most) location of each node, subject to the condition that the pitch $p = t - s$ is minimum.

Nodes which are on a longest s-t path (which is also a longest t-s path) are called **critical nodes**. The arcs that constitute the path are called **critical arcs**, and the entire path is called a **critical path**. The minimum and maximum locations of the non-critical (slack) nodes define the intervals in which they can be located without affecting the pitch of the layout. This two-longest-path analysis scheme is called the **critical-path method**, or CPM. It is widely employed in operations research [48,77,57]. Figure 6.2 depicts the results of the CPM on the graph of Figure 6.1.

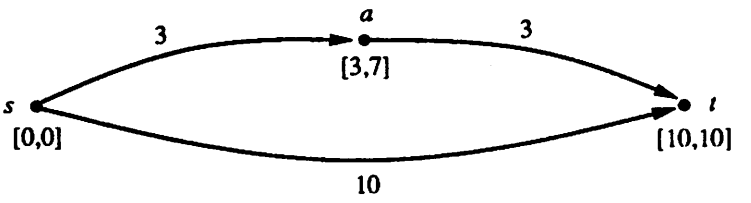


Figure 6.2: Critical path for graph of Figure 6.1.

The critical-path method provides two legal solutions for the set of all slack nodes. One solution is to locate all of them at their minimum locations, the other is to locate all of them at their maximum locations. Neither of these choices is likely to produce a desirable layout, even though the solutions are design-rule correct. As a result, the slack nodes are placed by optimizing a secondary objective. The secondary objective that is most widely employed is wire-length minimization [67,24,53,54].

The specific algorithms used in a constraint solver are a function of the constraint types allowed. However, the overall method always employs the following three-step procedure:

1. perform lp(s).
2. perform lp(t),
3. perform slack distribution.

The various types of constraint that are present in G affect the algorithms used in these analyses. Constraint types are addressed in the following section.

6.3 Constraint Types

Conventional compaction constraints are equations of the form $x_2 - x_1 \geq k$, where x_1 and x_2 are the locations of elements 1 and 2, respectively, in the direction of compaction. The quantity k is a separation requirement, which typically arises from a design rule.

In general four different constraint inequalities can be written. Considering a horizontal compaction, and assuming element 2 (x_2) is to the right of element 1 (x_1), the four inequalities are:

$$x_2 - x_1 \geq k \tag{6.2}$$

$$x_2 - x_1 \geq -k \tag{6.3}$$

$$x_2 - x_1 \leq k \quad \longrightarrow \quad x_1 - x_2 \geq -k \tag{6.4}$$

$$x_2 - x_1 \leq -k \quad \longrightarrow \quad x_1 - x_2 \geq k \tag{6.5}$$

where k is a positive integer. The above equations can be visualized by choosing x_1 as a reference and drawing the allowed interval for x_2, as shown in Figure 6.3.

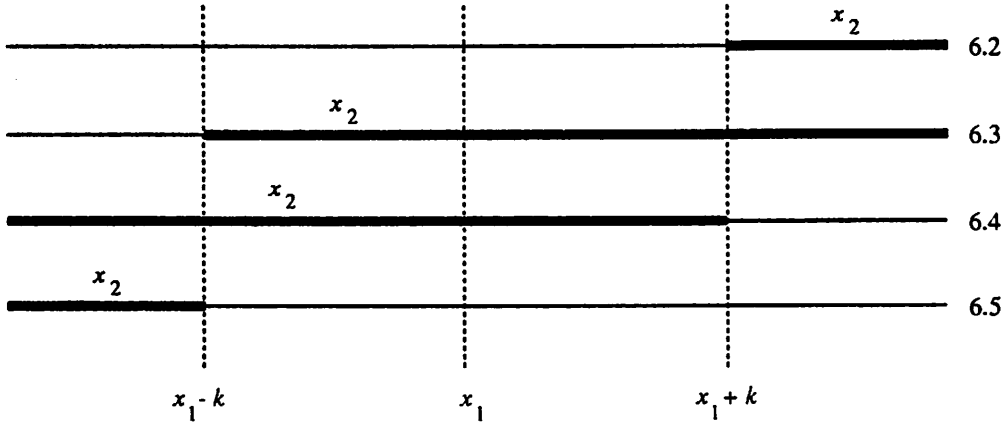


Figure 6.3: Four constraint inequalities.

Equation 6.2 is a positive **lower-bound** constraint: it limits the minimum spacing between x_2 and x_1. Equation 6.4 is a positive **upper-bound** constraint: it limits the maximum spacing between x_2 and x_1. Equation 6.3 is a negative lower-bound constraint, meaning that x_2 can move to the left of x_1, but for a distance of k units or less only. Equation 6.5 is a negative upper-bound constraint: x_2 cannot be any further to the right than $x_1 - k$. Since Equations 6.2 and 6.3 are essentially the same, the term "lower-bound

constraint" will be used to refer to them both. The term "upper-bound constraint" will likewise refer to either Equation 6.4 or Equation 6.5. Most spacing-rule requirements map to lower-bound constraints. Upper-bound constraints limit maximum separations; they are used, for example, to implement sliding terminals. A lower-bound/upper-bound pair of the same value implements an **equality** or **fixed constraint.**

6.3.1 Notation

A typical convention for drawing a constraint is to assume the greater-than-or-equal-to relation ($\geq$), and to associate the head of the arc with the first variable in the equation and the tail with the second.[2] Under this convention, a lower-bound constraint is termed a **forward** arc and an upper-bound constraint is a **backward** or **back** arc (Figure 6.4).

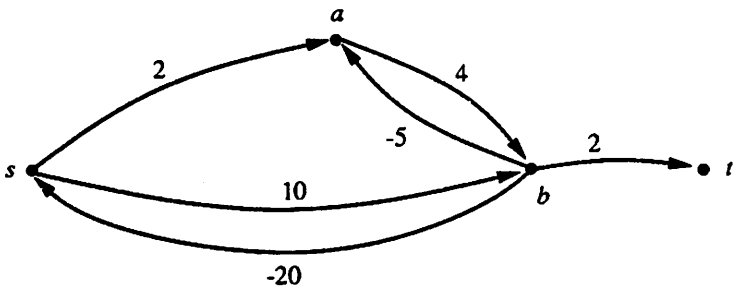


Figure 6.4: Forward and backward arcs.

For convenience, an alternate notation which is equivalent to this one will be used at times in this dissertation. In the alternate notation a single arc is used to denote both a lower-bound and an upper-bound constraint. Two weights are associated with an arc in the format L/U; L is interpreted as the lower-bound separation and U is interpreted as the upper-bound separation. If $L = -\infty$ ($U = \infty$) there is no lower-bound (upper-bound) constraint. The graph from Figure 6.4 is redrawn in Figure 6.5 to illustrate the alternate notation. The algorithms presented later in this chapter are described in terms of this alternate notation.

Regardless of the notation used, the direction of an arc has nothing to do with the compaction direction. The direction of an arc specifies the relative placement of its

[2]This convention has been used thus far in this dissertation.

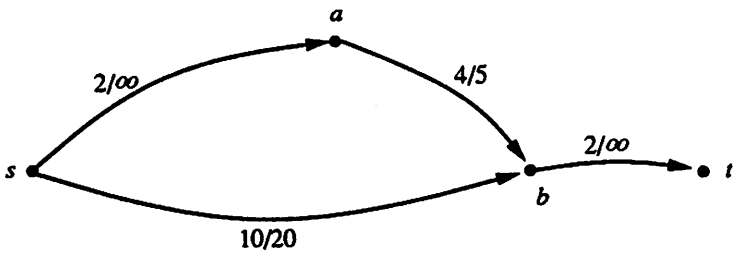


Figure 6.5: Graph of Figure 6.4 in the alternate notation.

head and tail nodes *only*. That is, for a horizontal constraint graph, the arc directions do not distinguish "compaction to the left" from "compaction to the right". In fact, a compaction direction in this sense is meaningless. The critical path is independent of whether compaction is leftward or rightward. Likewise, a well-behaved algorithm for slack distribution does not have a direction-dependent behavior.

6.4 Constraint Types and Problem Structure

The structure of a constraint graph is a function of the types of constraints that are allowed. Early compactors used lower-bound constraints only, whereas more recent programs use upper-bound as well as lower-bound constraints. The graph structures that follow from the inclusion of just lower-bound constraints, and from the inclusion of lower-bound plus upper-bound constraints, are described in this section.

6.4.1 Acyclic Problems

Constraint graphs consisting only of lower-bound constraints must be acyclic. This is easy to show; assume that such a graph does have a cycle, say from x_1 to x_2 to x_1 as depicted in Figure 6.6. This situation is clearly contradictory, because it indicates that x_2 is simultaneously to the left *and* to the right of x_1.

Assuming horizontal compaction, longest-path (LP) algorithms for acyclic problems operate by "pushing" nodes to the right; the location of each node is computed from the locations of its predecessors. In the example in Figure 6.7, s is initially at 0 and a, b, and t are initially at $-\infty$. If node a is processed first, it is pushed rightward to a new location given by $l_a = \texttt{MAX}(-\infty, 0+2) = 2$. If node b is then processed, its new location is

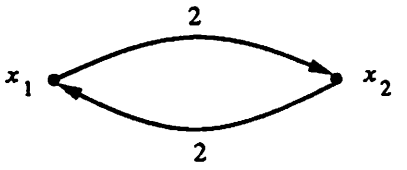


Figure 6.6: Illegal cycle of lower-bound constraints.

given by pushing it to $l_b = \texttt{MAX}(-\infty, 2+4, 0+10) = 10$. Node t is pushed to 12 in a similar manner.

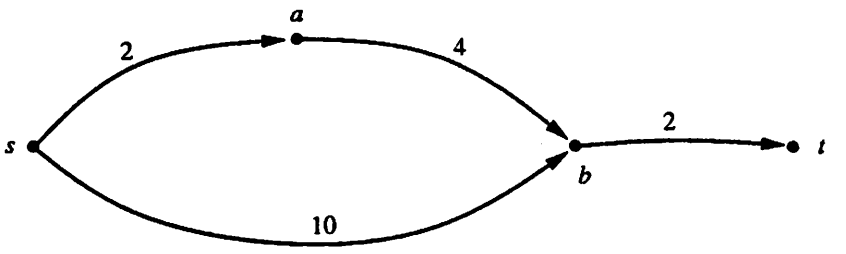


Figure 6.7: Illustration of "pushing" by lower-bound constraints.

A longest-path analysis can be performed in linear time on an acyclic constraint graph. In this case, it is possible to define processing orders such that each edge is visited only once. For example, a breadth-first traversal visits each edge one time, and each node is visited (updated) one time for each of its fanin edges. The graph in Figure 6.8, which is similar to that of Figure 6.7 but with different constraint values, provides an illustration. One breadth-first order is $(s,a), (s,b), (a,b), (b,t)$. Assuming the position of s is 0 and the initial positions of all other nodes are $-\infty$, the analysis proceeds as shown in Table 6.1.

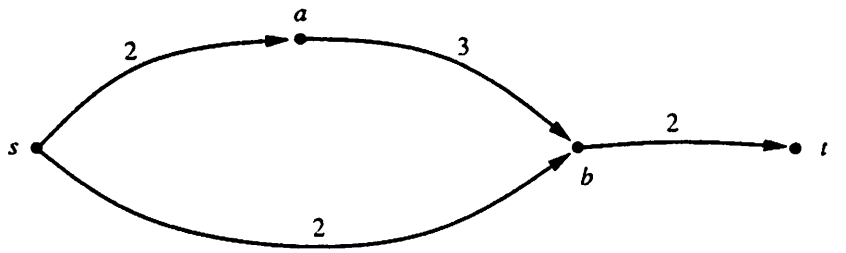


Figure 6.8: Graph comprised of lower-bound constraints.

The best possible processing order would visit each node and each edge exactly

Edge	Action
(s,a)	a moves to 2
(s,b)	b moves to 2
(a,b)	b moves to 5
(b,t)	t moves to 7

Table 6.1: Breadth-first longest-path for graph in Figure 6.8.

once. An ordering with this property does exist for acyclic graphs. This ordering will be called **level** order. The level of each node v is defined to be the length of the longest s-v path in edges.[3] Level numbers for the graph of Figure 6.8 are shown in Figure 6.9. The longest path is computed by processing the nodes in level order. When node v is processed, its fanin edges are visited to compute the new location for v. That is, for an edge (u,v), the position of u, which is already known, along with the value of the (u,v) constraint is used to compute the position of v. If v has multiple fanins, the one which maximizes v's position determines the final position of v. Table 6.2 shows the progression of this processing order for the graph of Figure 6.9. Each edge is visited once and each node's position is updated once, as desired. This method requires a linear-time algorithm for computing the level numbers. An algorithm for this task appears later in this chapter.

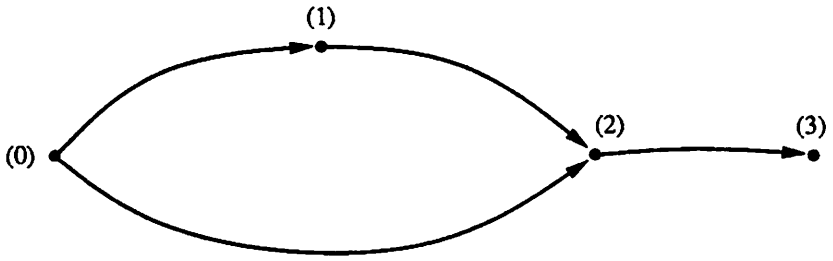


Figure 6.9: Level numbers for the graph in Figure 6.8.

Node	Action	Limited by
a	a moves to 2	s
b	b moves to 5	a
t	t moves to 7	b

Table 6.2: Level-order longest-path for graph in Figure 6.9.

[3]This is not the same as the level order defined in [74]: that ordering numbers nodes according to the shortest path, in edges, from the source.

6.4.2 Cyclic Problems

Both upper-bound and lower-bound constraints are necessary in a practical compactor. When upper-bound and lower-bound constraints are present, cyclic constraint graphs result.[4] For example, the addition of upper-bound constraint (b, a) to the graph of Figure 6.7 leads to a graph with one cycle as shown in Figure 6.10.

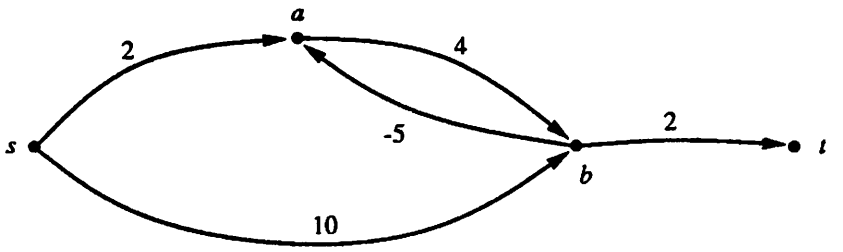


Figure 6.10: Cycle due to adding an upper-bound constraint.

When an upper-bound constraint is present, its effect is to "pull" (rather than push) the node at its head to the right. If an upper-bound constraint pulls node v to a new position further to the right, the positions of all of the descendants of v are potentially affected. Assuming that nodes a and b have been processed as described in the preceding subsection, the upper-bound constraint from b to a in Figure 6.10 must be processed next. The result is that the location of a becomes $l_a = \texttt{MAX}(2, 10 - 5) = 5$; that is, the (b, a) constraint has pulled a from 2 to 5. Since a moved, its descendants b and t must be reprocessed to insure that they still satisfy the constraints, which they do in this example. In the worst case, however, all descendants might move to new positions. Hence the back-arc processing should be scheduled to minimize reprocessing as much as possible.

The need for cyclic compaction graphs leads to two complications:

1. Linear-time longest-path analysis is not possible.

2. Unsolvable problems can occur.

A total ordering of the nodes that constitute a cycle does not exist. Hence processing a cyclic graph implies that some backtracking is inevitable, which in turn means that a linear-time LP algorithm does not exist. A graph with a cycle of net positive weight, such as cycle (s, a), (a, b), (b, s) of weight $2 + 4 - 3 = 3$ in Figure 6.11, does not have a finite longest

[4] Graphs with both types of constraints are sometimes called **mixed constraint graphs**.

path, because the length of the longest path can be made arbitrarily large by repeatedly traversing the positive cycle. As a result the longest-path problem is not solvable when one or more positive cycles exist.

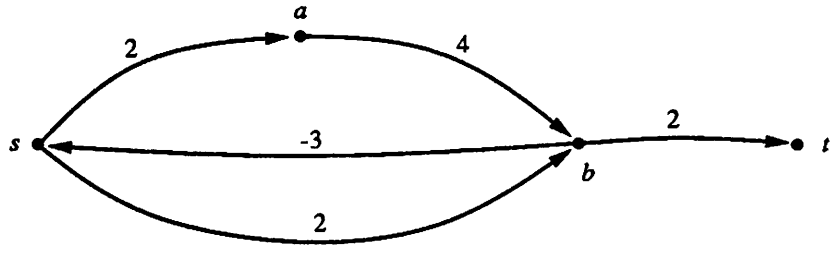


Figure 6.11: Graph with a positive cycle.

Positive cycles arise due to illegal inconsistencies in the constraints. This can be seen by examining the constraint equations for such a cycle. Referring to the graph in Figure 6.11, and observing that the longest forward path from s to b is the (s,a), (a,b) path, the constraint equations that relate s and b are

$$x_b - x_s \geq 6, \tag{6.6}$$

$$x_b - x_s \leq 3. \tag{6.7}$$

It is clear that b cannot simultaneously be *6 or more and 3 or fewer* units to the right of s. Therefore the constraints do not correspond to a realizable placement of the elements.

The important considerations in processing cyclic constraint graphs are thus to determine an efficient processing order for the longest-path calculation, i.e., an order that minimizes backtracking, and to provide a means for detecting problems with positive cycles, or **overconstraints**. Each of these considerations is addressed in this chapter.

6.5 Previous Work in Cyclic Longest-Path Analysis

The longest-path problem is well-known and well-studied. However, compaction constraint graphs tend to have a particular structure which can be exploited to develop an efficient longest-path algorithm specifically for compaction applications. Previous work in longest-path algorithms for cyclic compaction graphs is described in this section.

Bales proposed an event-driven algorithm for the longest-path problem in 1982 [5]. A queue is used as the scheduler for the vertices to be processed. Initially all vertices are

scheduled. When a vertex v is processed, its position is determined by the positions of its fanins, which can be either lower-bound or upper-bound constraints. If v moves to a new location, its fanins and fanouts are scheduled for processing. Fanin/fanout w is scheduled for the current iteration if it is not already scheduled. It is scheduled for the next iteration if it was previously scheduled in the current iteration. It is not scheduled if it is already scheduled, but not yet processed, for the current iteration. The method converges in $O(V^3)$ time in $O(V)$ iterations in the worst case, but its average time-performance is expected to be $O(V^{1.5})$ [5].

Liao and Wong proposed an algorithm for mixed-constraint longest-path analysis in 1983 [51]. The algorithm performs an alternating sequence of "push" and "pull" operations on the graph. In a push operation only lower-bound constraints are processed, and in a pull operation only upper-bound constraints are processed. The edges are partitioned into two sets, E_f and E_b, according to whether they are forward (lower-bound) or back (upper-bound) edges, respectively. Let $G_f = (V, E_f)$ be the graph induced by the forward edges. The push step processes G_f only; the push algorithm is essentially a linear-time longest-path method for acyclic graphs. Since back edges are ignored in the push step, some of the constraints in E_b might not be satisfied. The pull step corrects these violations by moving the head vertex of any unsatisfied back edge by an amount large enough to correct the violation. However, the pull step might create new violations among the constraints in E_f, which means that another push step must be performed to correct them. The push and pull steps are alternated until all constraints are satisfied. The algorithm has a worst-case complexity of $O((E_b + 1)E)$; convergence is obtained in at most $E_b + 1$ iterations, where an iteration is comprised of one push and one pull step, unless there is a positive cycle in the graph [51].

Lo et al. proposed an event-driven variant of the Liao and Wong algorithm in 1987 [54]. This method retains the same time bound as the original algorithm, but it achieves higher efficiency via the use of event-driven processing in the push operation.

6.5.1 Weaknesses

The above algorithms have desirable properties, but they nevertheless can be improved upon. The method proposed by Bales uses event-driven processing, which is beneficial. However, the processing order is not particularly efficient. The algorithm of Liao

and Wong, and likewise that of Lo et al., can also be improved in the area of the processing order employed. This point is illustrated in the following section.

6.6 Rationale for SPARCS Longest-Path Algorithm

As stated previously, the longest-path (LP) problem cannot be solved in linear time on a cyclic constraint graph, because the existence of cycles implies that the longest-path algorithm must backtrack. A key objective in formulating an LP algorithm in this situation is thus to choose an efficient backtracking scheme. The backtracking scheme, or processing order, should be selected according to the structure of the graphs that will be encountered.

6.6.1 Structure of Constraint Cycles

A cycle includes at least one lower-bound constraint (forward arc) and at least one upper-bound constraint (back arc). Lower-bound constraints arise from design-rule requirements, whereas upper-bound constraints arise from sliding terminals and from user constraints. The number of lower-bound constraints is thus much larger than the number of upper-bound constraints in most cases. The number of user constraints is usually small; hence most of the upper-bound constraints are due to sliding terminals. The upper-bound/lower-bound pair that implements a sliding terminal results in a 2-cycle.[5] The conclusion that follows from these observations is that *most of the cycles in a typical constraint graph are 2-cycles.* This conclusion has been used to choose the processing order for the longest-path algorithm used in SPARCS, which is described later in this chapter.

6.6.2 Backtracking in the Push-Pull Algorithm

The push-pull algorithm of [51] exploits the fact that the number of lower-bound constraints is typically greater than the number of upper-bound constraints. However, it does not take advantage of the observation that most cycles are short, where "short" refers to the number of edges in the cycle. The graph in Figure 6.12 contains a 2-cycle. If the push-pull algorithm is used to solve this graph, then two push passes and two pull passes are required. During the first push pass node a is moved to 2, which is not its final, legal position. However, its descendants c, d, and t are processed anyway, and they are positioned

[5] A k-cycle is a directed cycle comprised of k edges.

with respect to the current, illegal location of a. During the first pull pass, node a moves to 5, which invalidates the positions of nodes c, d, and t computed in the first push pass. The second push pass corrects the locations of c, d, and t due to the new position of a. The final pull pass does not change any node locations, which terminates the algorithm. The computation is summarized in Table 6.3. The total number of node-processing events is 12; each push operation processes 5 nodes, and each pull operation processes 1 node. The processing of c, d, and t in the first push operation is not useful, since it is based upon an illegal, intermediate position of node a. If this processing were eliminated, the number of node-processing events would be decreased by 25%.

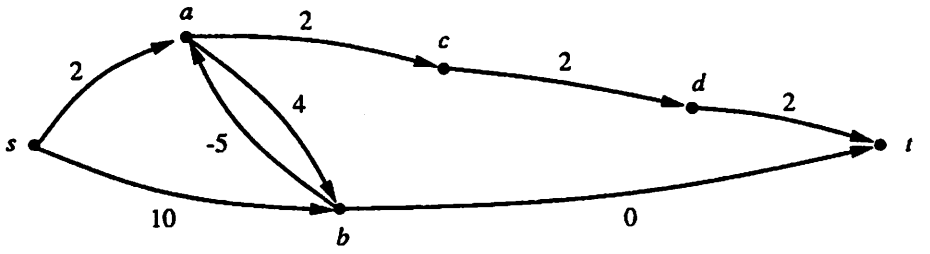


Figure 6.12: Graph with 2-cycle.

Node	Initial	Push_1	Pull_1	Push_2	Pull_2
s	0	0^*	0^*	0^*	0^*
a	$-\infty$	2	5^*	5^*	5^*
b	$-\infty$	10^*	10^*	10^*	10^*
c	$-\infty$	4	4	7^*	7^*
d	$-\infty$	6	6	9^*	9^*
t	$-\infty$	10	10	11^*	11^*

(Starred entries are legal.)

Table 6.3: Push-pull algorithm on graph of Figure 6.12.

It is apparent from this example that a given node should not be processed while any of its ancestors are in illegal positions.

6.6.3 Proposed Backtracking Order

Let v be the node at the head of a back arc that is on a longest path. The the motivation for the proposed backtracking order is to *minimize the processing of the*

descendants of v while v is in an illegal position. If the cycle induced by a back-arc is short, then an efficient backtracking scheme that satisfies this goal is to process the back-arc *immediately, as it is encountered*, rather than after all the arcs in E_f are processed. The effect of this ordering is that the influence of the back arc on v's position is accounted for before its descendants are reached.

In the proposed method, the nodes are processed in level order. The level numbers are computed by ignoring the back arcs; since the remaining arcs are forward arcs, the graph induced by them is acyclic and the level ordering exists. Furthermore, the algorithm is event-driven, meaning that a node is scheduled for processing only if one or more of the nodes adjacent to it changes position.

The graph in Figure 6.12 is redrawn in Figure 6.13, with the level numbers included, to illustrate the method. As in the preceding examples, the position of s is initialized to 0 and all other nodes are initialized to $-\infty$. The execution of the algorithm is summarized in Table 6.4.

This graph has a single node (a) with level equal to 1, and two nodes with level equal to 2, namely nodes b and c. Node a is therefore processed first, which results in its location being updated to 2 (Step 1). Since a moved its fanouts b and c are scheduled for processing. Either b or c might be processed next, because their levels are the same. In the worst case c is processed before b; assuming this ordering, c moves to 4 and causes d to be scheduled (Step 2). Of the two scheduled nodes, b and d, b has a lower level number so it is processed next. The result is that b moves to $l_b = \texttt{MAX}(-\infty, 2+4.0+10) = 10$ (Step 3).

The movement of b results in the scheduling of its fanouts. Node b thus schedules t and a; a is scheduled due to the back arc (b, a). Node a has the lowest level number of the scheduled nodes; hence it is immediately re-processed, which causes its location to be changed to 5 (Step 4). Note that a is now in its final location. Nodes b and c are re-scheduled as a result of the movement of a as indicated in Table 6.4.

The location of b does not change when it is processed in Step 5, thus the fanouts of b are not scheduled.[6] Node c then moves to 7, which would have scheduled d if it weren't already scheduled. The next node processed is d. If node d were processed before Step 7 that computation would be useless, because d's fanin node c did not reach its legal location until Step 6. Node t is processed in Step 8, terminating the algorithm. As in the case of

[6] Either b or c can be processed here, with no difference in the total number of node-processing events.

node d, earlier processing of t would be based upon illegal locations of one or more of its ancestors, and hence that processing would be of no use.

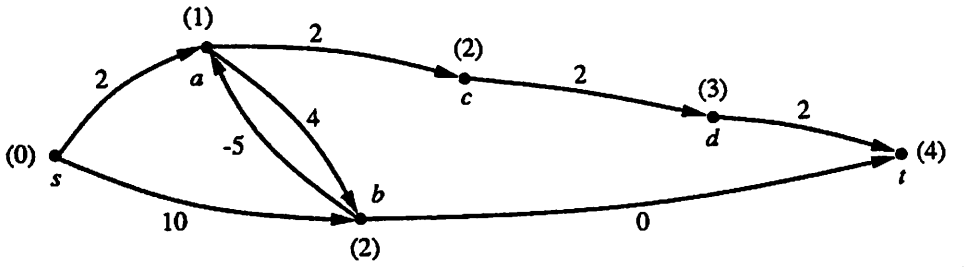


Figure 6.13: Graph of Figure 6.12 with level numbers.

Step	Node	Prev. Loc.	New Loc.	Nodes Sched.	Sched. Queue
1	a	$-\infty$	2	b, c	c, d
2	c	$-\infty$	4	d	b, d
3	b	$-\infty$	10^*	t, a	a, d, t
4	a	2	5^*	b, c	b, c, d, t
5	b	10^*	10^*	—	c, d, t
6	c	4	7^*	(d)	d, t
7	d	$-\infty$	9^*	(t)	t
8	t	$-\infty$	11^*	—	—

(Starred entries are legal.)

(Parenthetical entries are already scheduled.)

Table 6.4: Proposed algorithm on graph of Figure 6.13.

The total number of node-processing events is 8 for this example, which is 33% less than that achieved by the push-pull algorithm. If b were processed in Step 2 instead of c, the number of events would be 7 which is 42% less. This savings in computation results from choosing a backtracking order that is suited to the expected structure of the constraint graphs.

A formal description of the SPARCS LP algorithm appears in the following section.

6.7 SPARCS Longest-Path Algorithm

As described in the preceding section, a key characteristic of the SPARCS longest-path algorithm is that nodes are processed in level order. The level-number computation is the first operation performed in the SPARCS LP algorithm.

6.7.1 Leveling

The back arcs (E_b) are ignored in the level-number computation; hence the input to leveling is $G_f(V, E_f)$. By the argument given in Section 6.4.1, G_f must be acyclic. If it is not, an error is reported to the user.

The level numbers are determined via the modified breadth-first search algorithm given in Figure 6.14. Each vertex contains a field denoted "`inDegree`" that is initialized to its indegree in G_f. For a given vertex v, `inDegree` is decremented by one whenever one of its fanin edges $e \in E_f$ is visited. When `inDegree` reaches 0, v is placed at the end of the scheduling queue and its level is set to be one greater than the level of the current vertex (which is a fanin of v). This modification insures that vertices are not processed until all of their fanin vertices have been processed.

It is easy to see that `level()` executes in linear time. Each node is scheduled exactly once, namely when its `inDegree` value reaches 0. Thus there are a total of $|V|$ nodes processed. Processing node v requires one pass through its fanout edges, hence each edge is visited exactly one time. The overall runtime is therefore $O(V + E_f)$, but since $|E_f| > |V|$ in most cases, `level()` can be regarded as $O(E_f)$.

6.7.2 Longest-Path

A pseudo-code description of the event-driven longest-path algorithm used in SPARCS is presented in Figure 6.15. This algorithm is executed after the level-number computation of the preceding subsection. The input to `longestPath()` is the entire (cyclic) graph $G(V, E)$. The pseudo-code in Figure 6.15 solves the single-source problem from the source node of G (i.e., `lp(`s`)`); a similar routine is used to solve the single-source problem in the reverse direction, starting from the sink node of G. This routine computes the minimum legal location for $v \in V$; the value is stored in the `absLower` field of v. The pseudo-code assumes the alternate notation for G from Section 6.3.1.

```
level()
{
    struct node *pn, *pfoN;          /* current node, fanout node */
    struct edge *pfoE;               /* fanout edge               */
    int curLevel;                    /* level of current node     */
    extern int enQueue();            /* adds node to queue        */
    extern struct node *deQueue();   /* gets next node from queue */

    SourceNode->level = 0;           /* assign source level of 0  */
    enQueue(SourceNode);             /* schedule source to start  */

    while ((pn=deQueue()) != NIL) {
        curLevel = pn->level;
        /*
         *  Scan the fanout nodes of the current node.  Each
         *  fanout's inDegree is decremented.  When the inDegree
         *  reaches 0, all fanins of pfoN have been processed,
         *  so pfoN is assigned a level and put in the queue.
         */
        for (pfoE = pn->fanout;  pfoE != NIL;  pfoE = pfoE->next) {
            pfoN = pfoE->toNode;
            pfoN->inDegree--;
            if (pfoN->inDegree == 0)  {
                pfoN->level = curLevel + 1;
                enQueue(pfoN);
            }
        }
    }

    /*
     *  Un-leveled node ==> directed cycle(s) are present.
     */
    for (pn=NodeList; pn != NIL; pn=pn->list)  {
        if (pn->level == UNDEF)  {
            printf("level: illegal graph; cycle(s) present.\n");
            return(ERROR);
        }
    }
}
```

Figure 6.14: Algorithm for level-number computation.

```
longestPath()
{
    struct node *pn;                /* current node              */
    struct branch *pb;
    extern struct node *heapDeleteMin();
    extern void heapInsert();
    int newlb, newloc, maxSched;
    int schedCount = 0;

    maxSched = NumNodes*NumUppers;

    SourceNode->absLower = 0;
    for (pb=SourceNode->fanout; pb != NULL; pb=pb->nextfo)  {
        heapInsert(pb->toNode);     /* insert, mark as sched.    */
        schedCount++;
    }

    while ((pn=heapDeleteMin()) != NIL && schedCount < maxSched)  {
        newlb = newub = -INFINITY;
        for (pb=pn->fanin; pb != NIL; pb = pb->nextfi)  {
            if (pb->lower > -INFINITY)  {
                newlb = max(newlb, pb->fromNode->absLower+pb->lower);
            }
        }

        for (pb=pn->fanout; pb != NIL; pb=pb->nextfo)  {
            if (pb->upper < INFINITY)  {
                newub = max(newub, pb->toNode->absLower-pb->upper);
            }
        }
        newloc = max(pn->absLower, max(newlb, newub));
        if (newloc != pn->absLower)  {
            pn->absLower = newloc;
            for (pb=pn->fanin; pb != NIL; pb=pb->nextfi)  {
                if (pb->upper < INFINITY)  {
                    if (pb->fromNode->sched == NIL(struct node))  {
                        heapInsert(pb->fromNode);
                        schedCount++;
                    }
                }
            }
```

```
            for (pb=pn->fanout; pb != NIL; pb=pb->nextfo)  {
                if (pb->lower > -INFINITY)  {
                    if (pb->toNode->sched == NIL(struct node))  {
                        heapInsert(pb->toNode);
                        schedCount++;
                    }
                }
            }
        }
    }
    if (schedCount >= maxSched)  {
        printf("longestPath:  overconstraint on forward pass\n");
        printf("    current node:  %s (%d)\n", pn->name, pn->level);
        printf("    sched. nodes:  ");
        heapNodePrint();
        exit(0);
    }

    return(OK);
}
```

Figure 6.15: SPARCS longest-path algorithm.

As described in Section 6.6. `longestPath()` processes the scheduled nodes in level order. Because of this, the scheduler must return the node of lowest level from the set of scheduled nodes. A simple FIFO queue thus does not suffice as a scheduler. Instead, the appropriate data structure for the scheduler is a heap (also called a priority queue). A heap is a sorted data structure that supports, at least, the following two operations:

1. `heapInsert(`v`)` - add element v with key k to the heap.
2. `heapDelMin()` - remove and return the element from the heap with the smallest key.

An efficient heap implementation is an important component of an efficient implementation of the SPARCS longest-path algorithm.

The scheduler used in SPARCS is a d-heap, meaning that it is implemented via a complete, heap-ordered d-tree. A heap-ordered tree has the property that, for a node x and its parent $p(x)$, the key of the element in $p(x)$ is less than or equal to the key of the element

in x. This property guarantees that the element of minimum key is always stored in the root of the tree. In a complete d-tree, each node has at most d children and new nodes are added in breadth-first order. Johnson invented the d-heap in 1975 [34]. The actual implementation used in SPARCS is similar to that described in [74].

Locating the element of minimum key requires $O(1)$ time, but its deletion means that heap-order must be restored to the remaining nodes in the tree. As a result, when a complete d-tree is used, the `heapDelMin()` operation runs in $O(d \log_d n)$ time, and the `heapInsert()` operation takes $O(\log_d n)$ time [74]. If a total of m heap operations are performed, the overall runtime is thus $O(md \log_d n)$. The parameter d must be selected experimentally, because the constant factors that multiply the runtimes for the heap operations must be accounted for. The best value for d has been found to be three.

Performance

Assuming the alternate notation described in Section 6.3.1, a particular fanin vertex v_{fi} of vertex v is scheduled only if v changes and there is an upper-bound constraint between them. A fanout vertex v_{fo} of vertex v is scheduled only if v changes and there is a lower-bound constraint between them. If there are no upper-bound constraints then at most $|V|$ vertices are scheduled. Let the number of upper-bound constraints be K. Each upper-bound constraint causes the vertex at its head to to be re-processed at most once, otherwise the graph has a cycle of net positive weight. In the worst case, each re-processing of a vertex can cause V other vertices to be scheduled. The overall worst-case complexity for `longestPath()` is therefore $O(VK)$. In practical cases the worst-case behavior of this algorithm is seldom encountered, because most backtracking is local in nature.

Once the lower limit `absLower` for each vertex has been established, the algorithm is executed in the reverse direction starting at the sink node to find the upper-limit value `absUpper` for each node. If `absLower = absUpper` then v is a critical node.

Table 6.5 contains runtimes in CPU-seconds for several examples. The times include all processing, that is, leveling, the forward and reverse LP analyses, and slack distribution (Section 6.9). The measurements were performed on a DEC VAX 11/785 running Berkeley Unix.

Vertices	Constraints	Time
15	226	0.05
101	2,024	0.29
201	40,200	5.15

Table 6.5: Execution times for the longest-path algorithm.

6.8 Positive Cycles

As described in Section 6.4.2, a graph with one or more overconstraints, i.e., directed cycles of net weight greater than zero, is inconsistent and unsolvable. Because a directed cycle must contain at least one forward arc and at least one back arc, such a cycle is sometimes referred to as a **mixed** positive cycle.

The existence of positive cycles can be detected in the time required to solve a legal graph [5]. However, this information is of little use since the vertices and edges which comprise the positive cycles are not enumerated. In such situations, the spacing program should detect the cycles and report all vertices and edges involved in the limiting lower-bound path and in the limiting upper-bound path from the left-most vertex to the right-most vertex of each cycle.

The subsections below contain descriptions of previous work in overconstraint detection, and a description of the overconstraint-detection algorithm used in SPARCS. Overconstraint detection is performed if `longestPath()` is unable to converge in the maximum number of iterations.

6.8.1 Previous Work

Some compactors determine the existence of overconstraints by iteration count, but give no indication as to which particular constraints are inconsistent. The program reported in [20] is able to detect an overconstraint, but only if it contains a single upper-bound constraint. The program described in [39] copes with overconstraints by consecutively ignoring constraints until the graph can be solved. Unfortunately, this results in an illegal layout.

An efficient algorithm for solving this problem is proposed by Tsakalidis in [78]; this method detects a single overconstrained cycle in $O(V, E)$ time. An algorithm with similar properties is alluded to in [74], but the detailed description is in an internal memorandum

and is therefore not available [75].

6.8.2 SPARCS Algorithm

The algorithm used in SPARCS and described here operates on a leveled graph and uses the event-driven scheduling mechanism of `longestPath()`. As mentioned previously, `level()` detects all cycles comprised strictly of lower-bound and upper-bound constraints; thus only mixed cycles are of concern following leveling.

The routine `overcon()` in Figures 6.16 and 6.17 detects all left-most and right-most node pairs in the graph that are involved in an overconstraint. For each pair, the worst-case paths, i.e., the largest lower-bound path and smallest upper-bound path, are reported. If there are a number of overconstraints between a given node pair, the paths causing the worst overconstraint are reported.

```
overcon()
{
    struct node *pn;                    /* ptr to current node  */
    struct edge *pe;                    /* ptr to current edge  */
    struct label *pl;                   /* ptr to current label */
    int numOvers = 0;
    extern struct node *heapDeleteMin();
    extern void heapInsert();

    /* schedule nodes w/ a lower and an upper-bound fanout */
    for (pn = NodeList;  pn != NULL;  pn = pn->list)  {
        if (pn->upperOutFlag  &&  pn->lowerOutFlag)  {
            pl = createLabel(pn);
            pl->origin = pn;            /* label's origin node  */
            pl->lower = -INFINITY;      /* initial path bound   */
            pl->upper = INFINITY;       /* initial path bound   */
            pl->active = TRUE;
            pn->labels = pl;
            pn->labelCnt = 1;
            heapInsert(pn);             /* schedule this node   */
        }
    }
```

```
    while ((pn = heapDeleteMin()) != NIL)  {
        /*
         *  Process the active labels of the current node
         */
        for (pl = pn->labels;  pl != NIL;  pl = pl->next)  {
            if (pl->origin->labelCnt > 2  ||  pl->origin == pn)  {
                if (pl->active == TRUE)  {
                    /* update/propagate to fanouts */
                    for (pe=pn->fanout; pe!=NIL; pe=pe->nextfo)  {
                        if (pe->toNode != pl->origin)  {
                            if (procFanoutLabels(pe, pn, pl))  {
                                heapInsert(pe->toNode);
                            }
                        }
                    }
                    /* update/propagate to fanins */
                    for (pe=pn->fanin; pe!=NIL; pe=pe->nextfi)  {
                        if (pe->fromNode != pl->origin)  {
                            if (procFaninLabels(pe, pn, pl))  {
                                heapInsert(pe->fromNode);
                            }
                        }
                    }
                    /* deactivate unnecessary labels */
                    if (pl->origin != pn)  {
                        discardLabel(pl);
                    }
                    /* check for overconstraint at this node */
                    if (pl->lower > pl->upper)  {
                        numOvers++;
                        pn->overcon = TRUE;
                        pl->origin->overcon = TRUE;
                    }
                }
            }
        }
    }
    if (numOvers != 0)  {
        printOverconPaths();        /* trace cycles via labels */
    }
}
```

Figure 6.16: Main routine of overconstraint-detection algorithm.

```
/*
 *  subroutine to update/propagate fanout labels
 */
procFanoutLabels(pe, porigin, pl)
struct edge *pe;                        /* fanout edge to process */
struct node *porigin;                   /* origin of labels       */
struct label *pl;                       /* current label          */
{
    struct label *pfoLbl, *pnewLbl;
    int found, moved, newLower, newUpper;
    int curUpper, curLower;
    struct node *pfoN;

    moved = FALSE;
    found = FALSE;
    pfoN = pe->toNode;
    curUpper = pe->upper;
    curLower = pe->lower;
    pfoLbl = pfoN->labels;
    for (; pfoLbl != NIL; pfoLbl = pfoLbl->next)  {
        if (pfoLbl->origin == pl->origin)  {
            /* Already on list so merge constraints */
            newLower = max(pl->lower+curLower, -INFINITY);
            if (newLower > pfoLbl->lower)  {
                pfoLbl->lower = newLower;
                moved = TRUE;
            }
            newUpper = min(pl->upper+curUpper, INFINITY);
            if (newUpper < pfoLbl->upper)  {
                pfoLbl->upper = newUpper;
                moved = TRUE;
            }
            if (moved  &&  pfoLbl->active == FALSE)  {
                pfoLbl->active = TRUE;
                pfoLbl->origin->labelCnt++;
            }
            found = TRUE;
        }
    }
```

```
    if (found == FALSE)  {
        if (pl->origin->labelCnt > 2 || pl->origin == porigin)  {
            /* must add a new label to list at fanout node */
            pnewLbl = createLabel(pfoN);
            pnewLbl->origin = pl->origin;
            pnewLbl->lower= max(pl->lower + curLower, -INFINITY);
            pnewLbl->upper= min(pl->upper + curUpper, INFINITY);
            pnewLbl->active = TRUE;
            pl->origin->labelCnt++;
            moved = TRUE;
        }
    }
    return(moved);
}
```

Figure 6.17: `procFanoutLabels()` subroutine of overconstraint algorithm.

Under the alternate graph notation presented in Section 6.3.1, all potential overconstraints must contain at least one node that has at least one fanout edge with an upper bound constraint and at least one fanout edge that has a lower-bound constraint. This follows from the fact that non-mixed cycles have already been detected. The `overcon()` algorithm begins by scheduling these vertices.

The algorithm uses a labeling scheme to record positive cycles. A label is a small data structure that propagates through the graph from node to node. Each node has a set of labels containing its minimum and maximum allowable positions as determined up to the current point in time, as well as the path that has been traversed in computing those positions. When a node is processed its minimum and maximum positions are updated. If the processing of the current node v_c causes the bounds on its position to change, its fanout nodes are scheduled for later processing. The label of each fanout node v_f is augmented by appending a new label to include the new bounds on its position, and to include the fact that v_c is in the path traversed to reach v_f. Each label also has an entry that denotes the node where the label originated, which in this case is v_c.

As each node is processed its labels are scanned to check for two or more labels that originated at a common node. Say for example that two such labels are detected. Each label represents a different path from the common node to the current node. The minimum

and maximum legal positions for the current node for each of the two paths are known and are compared for consistency. If the minimum position from one path is greater than the maximum position from the other an overconstraint exists. The two relevant paths can be enumerated, since they are stored in the labels. If the paths are consistent there is no overconstraint and the two labels are discarded.

The `procFaninLabels()` routine is similar to `procFanoutLabels()` of Figure 6.17.

Performance

The algorithm presented herein has worst-case time and memory complexities of $O(V_O E)$, where V_O is the number of vertices with at least one upper-bound fanout edge and one lower-bound fanout edge. In most practical situations, this number is small relative to the total number of vertices. Unlike the other overconstraint-detection algorithms, the `overcon()` algorithm finds all worst-case overconstraints in the graph in a single pass.

Table 6.6 contains the execution times of the overconstraint-detection algorithm for three overconstrained graphs. The times are in seconds for a C-language implementation running on a DEC VAX 11/785 under Berkeley Unix.

Nodes	Lower-Bound Constraints	Upper-Bound Constraints	Overconst. Paths	Time
15	217	11	1	.10
40	315	37	7	.41
101	2024	76	4	6.20

Table 6.6: Overconstraint detection times.

6.9 Slack Distribution

As mentioned in Section 6.2, the slack nodes may be assigned locations according to a secondary objective since they do not affect the critical path. The most appropriate secondary objective is minimization of the total wire length, subject to the constraint that the critical nodes are not allowed to move. A number of algorithms for wire-length minimization have been published [67,24,53,54].

Slack distribution is driven by a set of edge weights: the weight between nodes j and k will be denoted W_{jk}. The weights are typically derived from the resistivities and widths of the layers of the fabrication technology. If j and k are connected by a polysilicon

wire and k and l are connected by a metal wire, then $W_{jk} > W_{kl}$. If m and n are connected by a narrow metal wire and n and o are connected by a wide metal wire, then $W_{mn} < W_{no}$. The weight between two nodes is interpreted as an attractive force. The net, directed force on any node v is readily determined by summing the force(s) pulling it to the left and right (or top and bottom).

6.9.1 SPARCS Slack-Distribution Algorithm

The slack-distribution scheme currently used in SPARCS is less sophisticated than true wire-length minimization. However, it does produce the same result as wire-length minimization for a restricted class of problems, and reasonable results in most other cases. This simpler method was selected because wire-length minimization was not a focus of this research.

The SPARCS algorithm is based on event-driven relaxation. A pseudo-code description of `slack()` is presented in Figure 6.18. All slack nodes are initially scheduled for processing. When node v is processed, its location is updated according to the current positions of its neighbors. If v moves, its fanins and fanouts are queued for processing. The algorithm terminates when the schedule queue becomes empty.

The behavior of `slack()` can be altered by changing the equation used to compute the new location of v. The options are controlled by the variable `slackMode` (Figure 6.18). If `MODE1` is selected the current node v is located according to the weighted average of the forces on v. If `MODE2` is selected v is moved as far as possible in the direction of the largest force. If `MODE3` is selected v is placed in the middle of its currently-allowed range of positions.

The algorithm for slack distribution that is described here differs from wire-length-minimization primarily because the current node v is positioned according to its immediate neighbors' positions only. This scheme is inadequate when the constraints between two or more adjacent slack nodes are tight. In this case, the nodes must be clustered together and treated as a unit. This clustering is a dynamic operation that should occur during the execution of the algorithm as constraints become tight. In addition, two (or more) nodes that have a fixed constraint between them should be clustered as well. This second form of clustering, which is static, *has* been included in the SPARCS implementation of `slack()`.

```
slack(slackMode)
int slackMode;
{
    struct node *pn;
    struct branch *br;
    extern int enQueue();           /* adds node to queue        */
    extern struct node *deQueue();  /* gets next node from queue */
    int leftForce, rightForce;      /* total left/right forces   */
    int newub, newlb, newloc, numer, denom;

    /*  enQueue all slack nodes to begin.  */
    for(pn = NodeList; pn != NIL; pn = pn->list) {
        if (pn->absUpper != pn->asbLower)  {
            pn->loc = pn->asbLower;            /* initial value */
            enQueue(pn);                       /* schedule      */
        }
    }
    while ((pn=deQueue()) != NIL)  {
        newlb = -INFINITY;  newub =  INFINITY;
        rightForce = leftForce = 0;
        for (br = pn->fanin; br!= NIL; br = br->nextfi)  {
            newlb = MAX(newlb, br->fromNode->loc+br->asbLower);
            newub = MIN(newub, br->fromNode->loc+br->absUpper);
            leftForce += br->force;
        }
        for (br = pn->fanout; br != NIL; br = br->nextfo)  {
            newlb = MAX(newlb, br->toNode->loc-br->absUpper);
            newub = MIN(newub, br->toNode->loc-br->asbLower);
            rightForce += br->force;
        }
        if (slackMode == MODE1)  {          /* weighted average */
            if (leftForce == 0)  {
                newloc = newub;
            } else if (rightForce == 0)  {
                newloc = newlb;
            } else  {
                numerator = (newub - newlb)*(rightForce);
                denominator = leftForce + rightForce;
                newloc = numerator/denominator + newlb;
            }
        } else if (slackMode == MODE2)  {  /* largest dir. wins */
            if (leftForce >= rightForce)
                newloc = newlb;
            else
                newloc = newub;
```

```
        } else if (slackMode == MODE3) {  /* equalize free space */
            newloc = (newlb + newub)/2;
        }
        /*  If pn moved schedule its noncritical fanins/fanouts  */
        if (newloc != pn->loc)  {
            pn->loc = newloc;
            for (br=pn->fanin; br != NIL; br=br->nextfi)  {
                if (br->fromNode->asbLower !=
                    br->fromNode->absUpper)  {
                    enQueue(br->fromNode);
                }
            }
            for (br=pn->fanout; br != NIL; br=br->nextfo)  {
                if (br->toNode->asbLower != br->toNode->absUpper) {
                    enQueue(br->toNode);
                }
            }
        }
    }
}
```

Figure 6.18: SPARCS event-driven slack-distribution algorithm.

6.10 Summary

Realistic constraint-based compaction requires that both upper-bound and lower-bound constraints be supported, which means that cyclic graphs must be processed. The need for cyclic graphs implies that an efficient processing order must be determined for graph traversals, and that a means for detecting overconstraints must be provided. These considerations have been addressed in this chapter.

The SPARCS algorithms for solving G were among the first to employ event-driven selective-trace techniques in layout compaction [14]. The backtracking order employed, namely level order, results in fewer node-processing events than other approaches, for graphs whose cycles are short. Level order is thus well-matched to the structure of most graphs that arise in layout compaction. This combination of level ordering and event-driven processing has led to efficient graph-solution algorithms for cyclic problems. A number of positive-cycle-detection algorithms exist. Unlike other methods, the `overcon()` algorithm described

in this chapter returns all worst-case positive cycles in a single pass.

Chapter 7

Active Constraints

7.1 Introduction

A compactor's flexibility and generality are determined, in part, by the constraint types supported. The combination of lower-bound and upper-bound constraints provides a level of flexibility and generality that is sufficient for a wide range of layout problems. However, these two constraint-types alone are not sufficient in a number of practical situations. For example, analog circuit layouts are often constructed in a symmetric manner, to minimize the effect of process gradients on critical, matched components [27]. Unfortunately, symmetry cannot be easily maintained during compaction with conventional constraints. This point is easy to show; conventional compaction constraints are two-variable inequalities, whereas a symmetry relationship cannot be described with less than three variables. A new constraint type is therefore required in this situation.

Such a new constraint type, called an **active constraint**, has been proposed and developed in this project [14]. An active constraint can be thought of as a mechanism for making the spacing between one pair of layout elements functionally related to the spacing between another pair. That is, when the spacing between the first pair of elements changes, the spacing between the second pair changes in response. The goal of the active-constraint investigation has been to develop a mechanism that efficiently handles realistic compaction problems, within the existing constraint-based compaction framework.

This chapter continues with several examples that motivate the need for an additional constraint type. The active-constraint problem is then defined, followed by an outline of the method of solution. The proposed solution algorithm is then derived in de-

tail. The theoretical complexity of the algorithm and its performance on several examples are presented next. The chapter concludes with a summary.

7.2 Motivation

In this section, three examples are presented to motivate the need for a more general constraint type beyond the conventional, two-variable compaction constraints presented previously.

7.2.1 Symmetric Configurations

Component matching is an important design consideration for analog circuits and for some high-performance digital circuits. In these applications, sets of elements are required to have identical electrical characteristics. For example, if the two input transistors of a differential amplifier are not identical, an undesirable input-offset voltage will be present which degrades the amplifier's performance.

One way in which component matching is addressed at the layout level is through the use of symmetry. Symmetric component placements are insensitive to certain types of fabrication-process gradients. The layout shown in Figure 7.1 is an example of a common-centroid placement, which is often used in analog designs [27]. The four bipolar transistors actually implement the two transistors of an emitter-coupled pair. The symmetry in x and y leads to better matching than an implementation comprised of two larger transistors.

Compaction with conventional constraints alone does not preserve symmetry. Figure 7.1 is redrawn in Figure 7.2 with the lines of symmetry indicated explicitly. Assuming that the reference point of each bipolar transistor is its center, symmetry is maintained about the horizontal line if and only if the following two equations are satisfied (in addition to the design-rule constraints):

$$y_a - y_h = y_b - y_h. \tag{7.1}$$

$$y_h - y_c = y_h - y_d. \tag{7.2}$$

Similar expressions apply in the horizontal direction. Equations 7.1 and 7.2 are four-variable relations that cannot be transformed into conventional, two-variable compaction constraints.

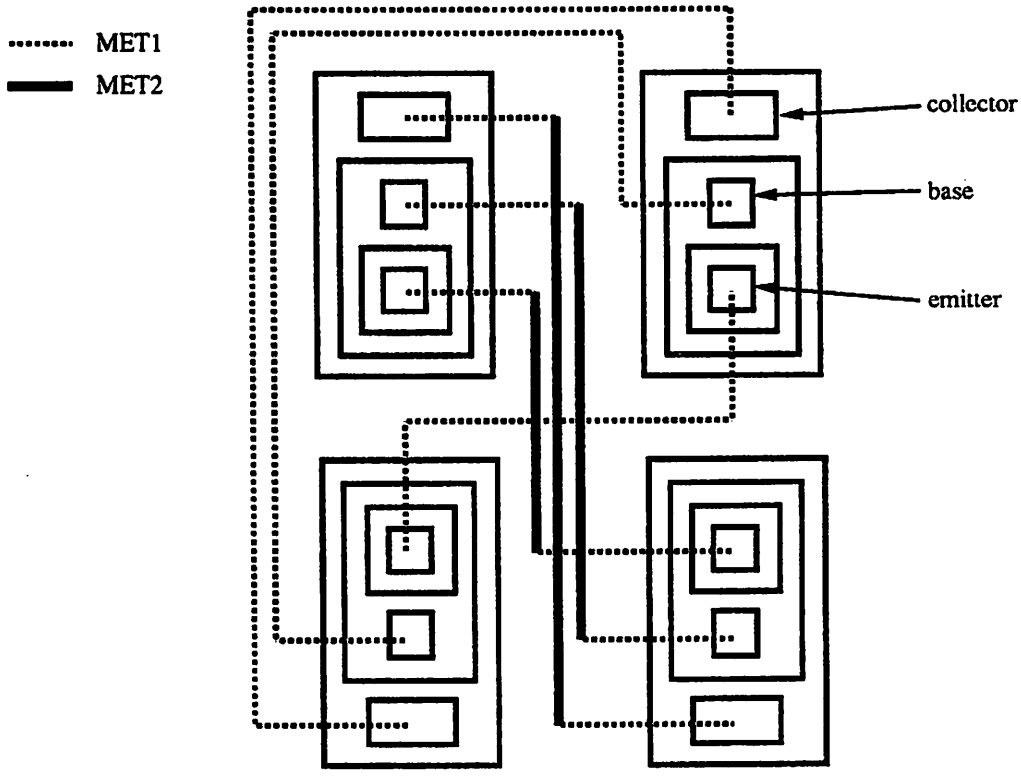


Figure 7.1: Common-centroid arrangement of four bipolar transistors.

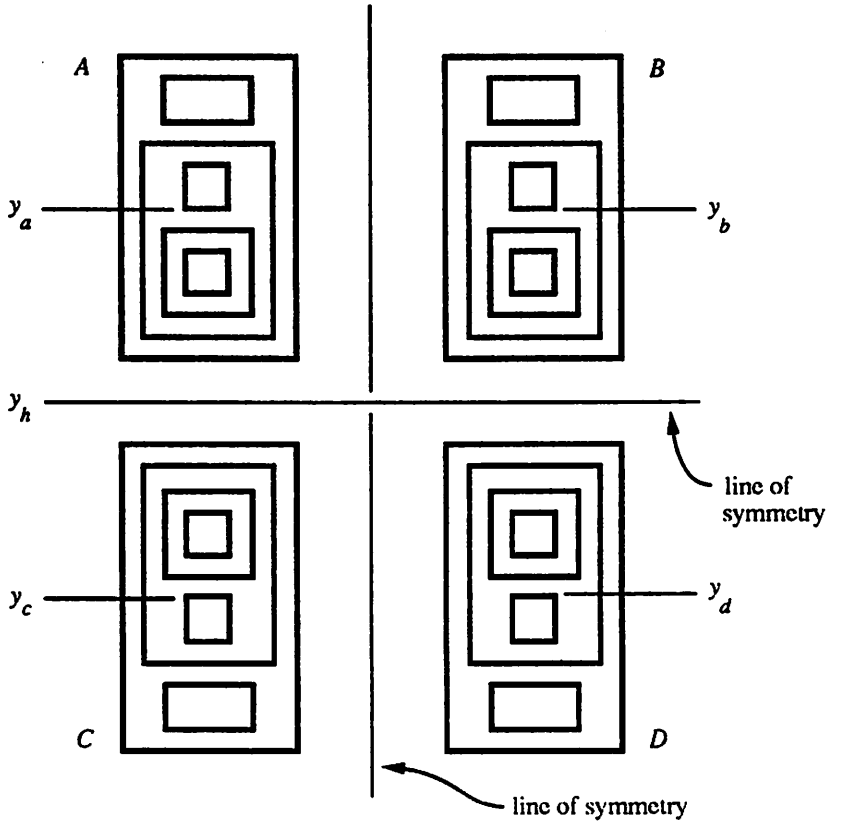


Figure 7.2: Lines of symmetry in Figure 7.1.

7.2.2 Hierarchy Preservation

The stretch-and-abut style of pitch-matching hierarchical compaction was described in Chapter 2. In this methodology, the subcells of the current cell are stretched such that connected terminals of adjacent subcells align. This guarantees that connection by abutment is achieved when the current cell is assembled by tiling together the stretched subcells.

A simple example of this layout style is presented in Figure 7.3. There are a total of four instances of three unique cells. Cell A appears twice, but in different environments in each case. When the stretching step is performed, the two instances of A do not, in general, remain the same. If the instances of A stretch differently because of their dissimilar environments, the resulting layout has four unique cells instead of three and the hierarchy is lost.

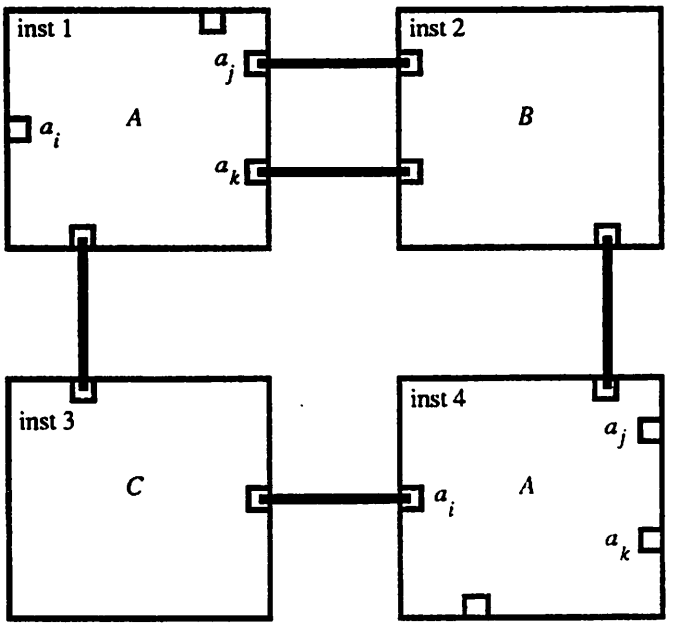


Figure 7.3: Pitch-matching example with a repeated cell.

The desired outcome is to preserve the hierarchy by forcing the two instances of A to remain the same during the stretching step. The vertical pitch-matching step, for which the parent-level constraint graph is shown in Figure 7.4, is used as an illustration. Consider the spacing between terminal a_i and the bottom edge of A; this spacing must be the same for instances 1 and 4. Similar conditions must be satisfied by the other terminals on the

left and right borders of A. In addition, the heights of the two instances of A must remain identical. Using subscripts to identify the instances, these conditions can be stated as

$$\begin{aligned} a_{i1} - s_1 &= a_{i4} - s_4, \\ a_{j1} - s_1 &= a_{j4} - s_4, \\ a_{k1} - s_1 &= a_{k4} - s_4, \\ t_1 - s_1 &= t_4 - s_4. \end{aligned}$$

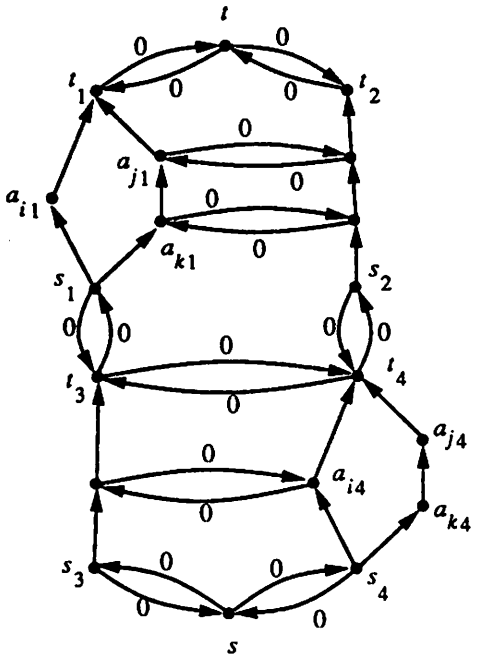


Figure 7.4: Vertical parent-level graph for example of Figure 7.3.

7.2.3 Equally-Spaced Configurations

There are a number of situations in which it is desirable to position a set of elements such that the spacing between each pair of adjacent elements is the same. In the macrocell design style, for instance, terminals often occur on the periphery of each macro. If the inter-terminal spacings are minimum, the task of routing the macros together can be difficult due to wiring congestion in the neighborhood of the terminals. The difficulty in interconnecting the macros can be eased by spacing the terminals apart, such that the

slack space is apportioned equally among them. For three terminals that are colinear in x, for example, the desired condition is

$$x_3 - x_2 = x_2 - x_1.$$

7.3 Problem Definition

The examples presented in Section 7.2 require constraints that relate the spacing between pairs of layout elements to the positions of some number of other elements in the layout. For a pair of elements i and j, this can be written as

$$x_j - x_i = F(x_a, x_b, x_c, ...) \tag{7.3}$$

where F is some arbitrary function of the locations of elements a, b, c, etc.

The problem described by Equation 7.3 is very general, and may or may not be solvable in a reasonable amount of CPU time, if at all. Fortunately, this level of generality is not necessary. All of the examples presented in Section 7.2 can be modeled if a constraint of the form

$$x_j - x_i = x_n - x_m \tag{7.4}$$

is added to the conventional compaction constraints described in Chapter 6. The four-variable constraint stated as Equation 7.4 is the basic form of the **active constraint** addressed in this research.

The term "active constraint" was selected to suggest an analogy with electric circuit theory. In an electric circuit, a voltage-controlled voltage source is an element that forces the voltage between one pair of nodes to be a prescribed function of the voltage between another pair. As a class, elements such as voltage-controlled voltage sources are called *active* elements. Active constraints are similar, in that a change in the separation between one pair of nodes forces a corresponding change in the separation between the other pair. (The meaning of the term active constraint in this dissertation is therefore not the same as its meaning in optimization theory.)

It is important to note that Equation 7.4 is not transformable into a set of conventional, two-variable constraints. All two-variable constraints have the form

$$x_j \geq x_i \pm k.$$

where k is an integer constant. The fact that k is a constant precludes the transformation of Equation 7.4 into three two-variable constraints.

The optimization objectives for compaction are unchanged with the addition of active constraints. As presented previously, the primary objective is to minimize the pitch of the layout in the spacing direction, and the secondary objective is to minimize the total wire length.

7.4 Solution Methods

It is well-known that the longest-path and slack distribution problems, subject to the conventional constraints that were presented in Chapter 6, can be formulated as linear-programming (LP) problems. Linear-programming problems can be solved via general methods such as the Simplex method. In practice, compactors use the graph-based formulation because graph-theoretic algorithms are more efficient than general LP algorithms like Simplex. This is the case because the structure of the graphs can be used to synthesize efficient processing orders for the cost functions of interest.

Incorporating active constraints results in a new problem that cannot be solved by the existing graph-theoretic algorithms. An obvious question is whether or not the compaction problem with active constraints can be formulated and solved as an LP problem. Unfortunately, it cannot be. The simple graph in Figure 7.5 will be used to illustrate this fact.

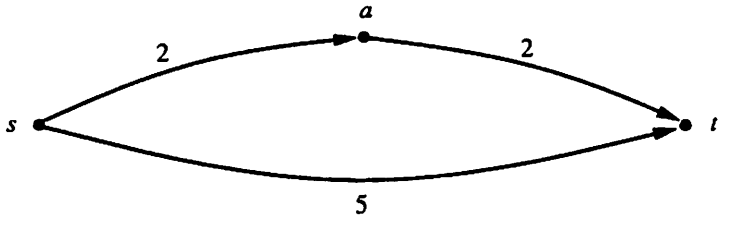


Figure 7.5: Illustration of the integer nature of the active-constraint problem.

The linear-programming formulation of the longest-path problem for the example in Figure 7.5 is written

$$\text{Minimize } y = x_t - x_s$$

subject to

$$\begin{aligned} x_a &\geq x_s + 2, \\ x_t &\geq x_a + 2, \\ x_t &\geq x_s + 5. \end{aligned}$$

Letting $x_s = 0$ for convenience, it is clear that the optimal solution is

$$\begin{aligned} x_a &= 2 \text{ or } 3, \\ x_t &= 5. \end{aligned}$$

If the active constraint

$$x_t - x_a = x_a - x_s$$

is added to the problem, then the solution becomes

$$\begin{aligned} x_a &= 2.5, \\ x_t &= 5. \end{aligned}$$

This solution is invalid, because x_a does not take on an integer value. Since the units used in compaction reflect the resolution of the fabrication process, noninteger coordinates are illegal.

To guarantee integer-valued results, the longest-path problem with active constraints must be formulated as an *integer linear programming problem*, viz.[1]

$$\text{Minimize } y = x_t - x_s$$

subject to

$$\begin{aligned} x_a &\geq x_s + 2, \\ x_t &\geq x_a + 2, \\ x_t &\geq x_s + 5, \\ x_t - x_a &= x_a - x_s, \\ x_s,\ x_a,\ x_t &\text{ integer}. \end{aligned}$$

[1] Actually, the longest-path and slack-distribution problems are ILP problems even without active constraints. However, when there are only conventional constraints, the problem is totally unimodular. When an LP problem is totally unimodular, its solution is always integer-valued [61].

The optimal solution to this simple ILP can be found by inspection; it is

$$x_a = 3,$$
$$x_t = 6.$$

It it interesting to note that *none* of the conventional constraints are tight when the pitch is at its minimum value of 6.

The integer linear programming problem (ILP) is a well-known NP-complete problem [61]. Since NP-complete problems require exponential time in general, it is not practical to formulate and solve the active-constraint problem as an ILP problem. Another possible approach might be to treat the problem as an LP problem, then round the result to obtain an integer value for each variable. Unfortunately, the problem of rounding a real-valued solution to a *feasible*, but not necessarily optimal, integer-valued solution is NP-complete as well [61]. Hence it is also impractical to use a combination of LP and rounding to solve the active-constraint problem. These concerns apply to both the longest-path and slack-distribution problems.

It has been recently proposed that the active-constraint problem be formulated as an LP problem and solved via the Simplex method [60]. However, by the reasons just presented, this approach does not guarantee a legal solution.

7.4.1 Proposed Method

The proposed solution method for compaction with active constraints is a graph-theoretic perturbation method, not an ILP-based method. The algorithm performs several single-source longest-path analyses for each active constraint. There are essentially two phases in the proposed method – a feasibility-determination phase, and an optimization phase.

The method for processing an active constraint is initialized by performing `lp(`s`)`, just as in the CPM, with the active constraint excluded. Then, in the first phase, a computation is performed to determine whether the active constraint is feasible. If it is, an optimization phase is entered in which the goal is to choose a value for the active constraint that is feasible, and that perturbs the minimum-pitch, minimum-wire-length solution by the smallest amount possible.

Although it is heuristic in nature, the proposed method has the following desirable features:

1. It is built upon the existing, efficient graph-solving algorithms.
2. Its complexity is polynomial, not exponential.
3. The algorithm performs efficiently on practical examples, in terms of layout quality and runtime.

The SPARCS active-constraint algorithm is described in detail in the remainder of this chapter.

7.5 Feasibility Analysis Overview

The first phase of the proposed active-constraint algorithm is a feasibility analysis, because some active constraints are infeasible. For example, it is impossible to make the separation between nodes a and c equal to the separation between nodes b and c for the graph in Figure 7.6; the a-c separation must always be larger than the b-c separation by at least 1.

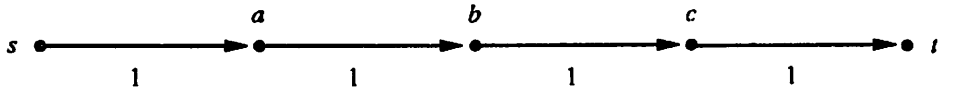


Figure 7.6: Infeasible active constraint.

For a single active constraint, e.g.,

$$x_j - x_i = x_n - x_m. \tag{7.5}$$

the feasibility analysis begins by determining upper and lower bounds on the position of each node in the active constraint. That is, bounds are found for each node i, j, m, and n for the constraint given by Equation 7.5. After the node bounds are calculated, upper and lower bounds on the *separation* between each *pair* of nodes are determined. In this case, pair-separation bounds are found for the i-j pair and the m-n pair. It will be shown in Section 7.7 that a given pair of nodes might have absolute lower and absolute upper bounds on their relative separation. For example, $x_j - x_i$ might be bounded by the interval [2, 9]. If absolute bounds exist for both pairs, and if those bounds do not intersect, then the active constraint is infeasible.

Even if the absolute bounds for both $x_j - x_i$ and $x_n - x_m$ intersect, the active constraint might be infeasible. This is because, in some cases, the two intervals are not *simultaneously* valid. The graph in Figure 7.6 has this property; increasing the *b*-*c* separation increases the lower bound on the *a*-*c* separation. The final phase of the feasibility check determines whether the separation intervals for each node pair simultaneously intersect, and if so, over what domain.

The following few sections describe the feasibility analysis in detail.

7.6 Bounds on the Position of a Node

The symbols defined in Table 7.1 will be used to describe the bounds on the position of a single node. All of the symbols take on integer values. The source node s is reference point for all nodes in G, in that all node locations are relative to the source node. For convenience, it will be assumed that s is located at zero.

Symbol	Definition
l_k	assigned position of node k
L_k	absolute minimum position of node k
U'_k	soft maximum position of node k
U_k	absolute maximum position of node k

Table 7.1: Notation for node bounds.

Constraint graphs are constructed such that all nodes are reachable from the source node. As a result, the *minimum* position of any node k is bounded by the longest path from s to k. This bound will be referred to as an **absolute minimum** position, because any node that is placed at a coordinate less than this bound violates one or more constraints. In Figure 7.7, node a, for example, must be placed such that $l_a \geq L_a = 2$; otherwise, the (s, a) constraint will be violated. Absolute minimum locations are indicated for each node in Figure 7.7 in square brackets. These locations are computed in `lp(`s`)`, a single-source longest-path analysis, which is the first phase of the critical-path method described in the preceding chapter.

The critical-path method computes a maximum position for each node as well. However, the maximum positions are not necessarily absolute. The maximum positions are determined by placing the sink node t at its minimum location, i.e., its position computed by `lp(`s`)`, then carrying out a longest-path analysis in the reverse direction, starting from

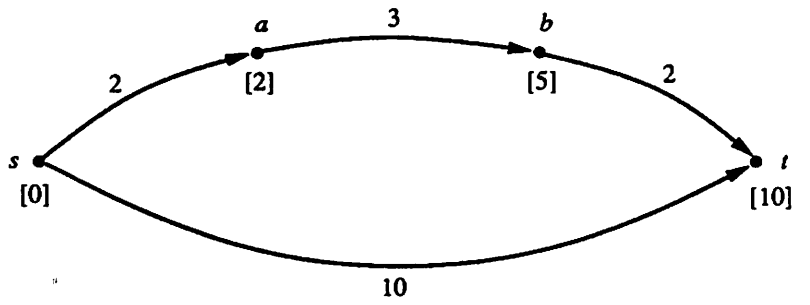


Figure 7.7: Absolute minimum locations.

t. For the graph in Figure 7.7, the sink node t is located at 10. The results of `lp(`t`)` are shown in Figure 7.8. The notation $[\alpha, \beta]$ indicates the minimum and maximum location, respectively, for each node.

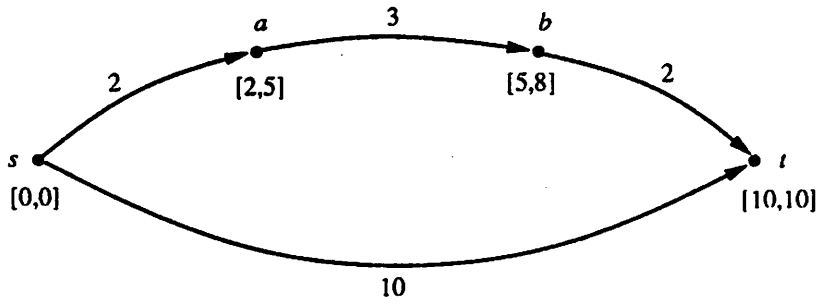


Figure 7.8: Minimum and maximum locations.

The graph in Figures 7.7 and 7.8 has no upper-bound constraints. The reverse analysis `lp(`t`)` computes a maximum location for each node, *subject to the condition that the pitch is minimum.* When upper-bound constraints are absent, these maximum locations *can* be exceeded without violating any constraints. Exceeding a maximum location in this case increases the pitch of the layout beyond its minimum value, but node locations can still be assigned such that the layout satisfies all the constraints. For example, node b can be legally placed at 9, provided t is moved to $l_t = 11$, which increases the pitch of the layout beyond its minimum value by 1.

Since they can be exceeded, these maximum positions will be termed **soft maximum** positions. Violating a soft-maximum position incurs a cost, but does not violate any constraints.

If the constraint graph contains upper-bound constraints, then **absolute maxi-**

mum positions exist for at least some of the nodes. Like absolute minimum positions and unlike soft maximum positions, absolute maximum positions *cannot* be exceeded without violating constraints. The graph from Figure 7.8 is augmented with an upper-bound constraint in Figure 7.9, and the range of legal positions for each node is indicated as before. If $l_b > 6$, then the (b, s) constraint of value -6 is not satisfied. Similarly, if $l_a > 3$ then $l_b > 6$ and the (b, s) constraint is again violated. Node t, on the other hand, is not affected by the addition of the (b, s) constraint; l_t can be increased beyond 10 without any constraint violations.

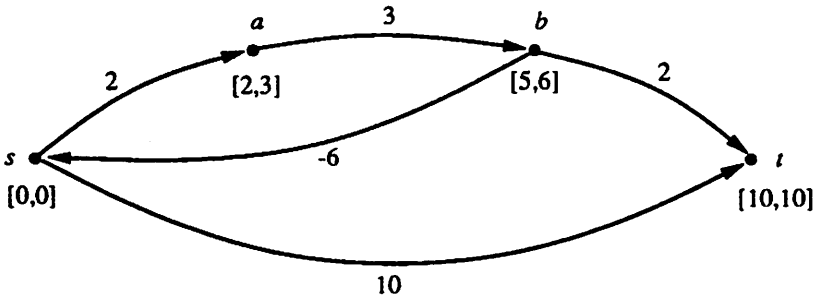


Figure 7.9: Addition of an upper-bound constraint.

The condition under which U_k exists can be deduced by comparing the graph in Figure 7.8 to that in Figure 7.9. In Figure 7.8, the source node s is not reachable from any node. In Figure 7.9, s is reachable from both a and b, but not from t. That is, there are directed paths from a to s and from b to s, but not from t to s. The condition for the existence of an absolute-maximum location follows from these observations; namely, if there is a directed path from k to s, then U_k exists and is equal to the absolute value of the length of the k-s path. If several such paths exist, U_k is the length of the longest one. If no such path exists, U_k does not exist. This condition makes sense intuitively; the source node is the reference for all nodes in G. Because of this, L_k is the longest forward path from s to k. If there is a path back to s from k, that path must constrain the maximum location of k, and that location must be absolute because s is the global reference for all nodes.

The condition for the existence of U_k can also be deduced from the constraint equations. The forward path to node b, for example, is given by

$$x_b \geq x_a + 3$$

$$x_a \geq x_s + 2$$

which reduce to

$$x_b \geq 5. \tag{7.6}$$

If no other paths involving s and b exist, then Equation 7.6 completely describes the relationship between s and b. However, if the back arc (b, s) is included, the equation

$$x_b \leq x_s + 6 \tag{7.7}$$

also applies. Equation 7.7 clearly represents an upper bound on the position of b with respect to s.

The existence of U_k can be determined by performing a single-pair longest-path analysis from k to s. This analysis will be denoted $\texttt{lp}(k, s)$. The single-pair analysis is essentially the same as the single-source analysis described previously.

The locations computed by $\texttt{lp}(t)$ may or may not be absolute maximums, as exemplified by Figure 7.9. Therefore, the results of $\texttt{lp}(t)$ will always be taken to be soft maximums, even though absolute maximums of the same value might exist as they do for nodes a and b in Figure 7.9. It is not generally true that both maximum bounds are equal. For the graph in Figure 7.10, which differs from that of Figure 7.9 in the value of the (b, s) constraint, $U'_a = 5$ and $U_a = 17$, and $U'_b = 8$ and $U_b = 20$. Node t is unchanged: $U'_t = 10$, but $\texttt{lp}(t, s)$ is undefined so t has no absolute maximum. This is indicated by the notation $U_t = \infty$. When node bounds are being addressed, the bound triples will be written in the form $[L_k, U'_k, U_k]$ (Figure 7.10).

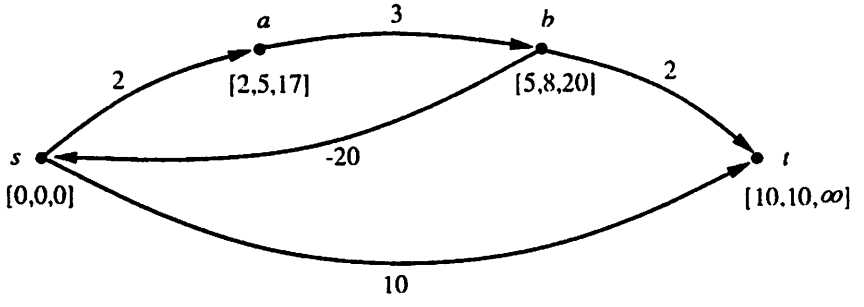


Figure 7.10: Graph showing node bounds $[L_k, U'_k, U_k]$.

The node bounds can be summarized as follows.

- L_k exists for all nodes, and is computed by $\texttt{lp}(s)$.

- U'_k exists for all nodes, and is computed by $\texttt{lp}(t)$.
- U_k exists for those nodes for which a k-s path exists, and is given by $|\texttt{lp}(k,s)|$.

7.7 Bounds on the Separation Between Two Nodes

The achievable distance (separation) between any two nodes i and j, i.e., $x_j - x_i$, is generally an interval that is bounded both above and below. Table 7.2 contains the notation that will be used to describe these bounds.

Symbol	Value	Definition
$\texttt{dist}(i,j)$	interval	synonym for $x_j - x_i$
$\texttt{d}(i,j)$	integer	current value of $x_j - x_i$
L'_{ij}	integer	soft lower-bound on $\texttt{dist}(i,j)$
L_{ij}	integer	abs. lower-bound on $\texttt{dist}(i,j)$
U'_{ij}	integer	soft upper-bound on $\texttt{dist}(i,j)$
U_{ij}	integer	abs. upper-bound on $\texttt{dist}(i,j)$
D'_{ij}	interval	$[L'_{ij}, U'_{ij}]$
D_{ij}	interval	$[L_{ij}, U_{ij}]$

Table 7.2: Notation for bounds on the separation between a pair of nodes.

The computation of the bounds on $\texttt{dist}(i,j)$ depends on whether the nodes have absolute upper-bounds on their positions, and on whether there are directed paths between them in the graph. If there are no i-j paths and no j-i paths, the nodes will be referred to as **independent**; otherwise they are **dependent**.

There are four cases to consider in computing the bounds, each of which is addressed in this section.

1. $U_i = \infty$ and $U_j = \infty$, i and j independent.
2. $U_i < \infty$ and/or $U_j < \infty$. i and j independent.
3. $U_i = \infty$ and $U_j = \infty$. i and j dependent.
4. $U_i < \infty$ and/or $U_j < \infty$. i and j dependent.

To clarify the description, it will be assumed that the direction of compaction is horizontal, and that the element corresponding to node j is to the right of the element corresponding to node i in the layout.

7.7.1 Case 1

The first case is the simplest. It is required that there be no directed paths from i to j or from j to i. As defined above, the two nodes are said to be independent under this condition. In addition, U_i and U_j must be undefined. A graph G that fulfills these restrictions is shown in Figure 7.11. Since only lower-bound constraints are present, $U_k = \infty$ for all nodes k in G. The node bounds are shown in the figure.

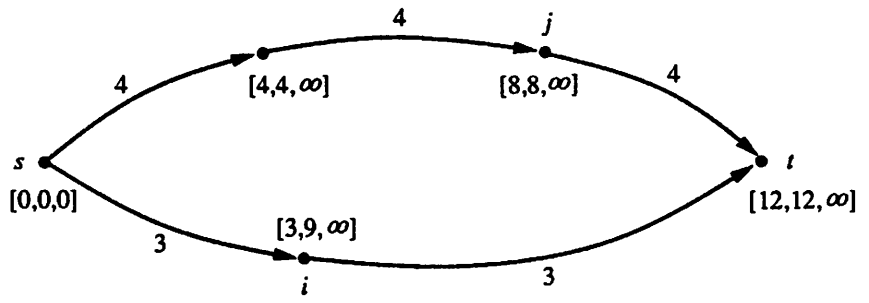


Figure 7.11: Graph with no upper-bound constraints, i-j, or j-i paths.

The distance between i and j can be reduced by moving j to the left (decreasing x_j), or by moving i to the right (increasing x_i). The minimum i-j separation is obtained by maximizing x_i and minimizing x_j. The absolute minimum position for node j is $L_j = 8$, and the soft maximum position for i is $U'_i = 9$. Hence the lower bound on the separation between i and j, subject to the condition that the layout pitch is minimum, is

$$\begin{aligned} L'_{ij} &= L_j - U'_i \\ &= 8 - 9 \\ &= -1. \end{aligned} \tag{7.8}$$

An upper bound on `dist`(i, j) can be similarly defined. The maximum separation is achieved by maximizing the position of j (moving it as far as possible to the right), and by minimizing the position of i (moving it as far as possible to the left). The limits on these movements are U'_j and L_i. The resulting upper bound on the separation is thus

$$\begin{aligned} U'_{ij} &= U'_j - L_i \\ &= 8 - 3 \\ &= 5. \end{aligned} \tag{7.9}$$

The bounds given by Equations 7.8 and 7.9 will be called **soft minimum** and **soft maximum** bounds, respectively. This is due to the fact that the interval $D'_{ij} = \left[L'_{ij}, U'_{ij}\right]$ can be expanded both above and below, since it is determined by node bounds that are soft. The interval can be expanded below by increasing U'_i, which decreases $L'_{ij} = L_j - U'_i$. The interval can be expanded above by increasing U'_j, which increases $U'_{ij} = U'_j - L_i$. However, in both cases the size of the layout increases beyond its minimum pitch.

Since U' is defined for all nodes, soft bounds are defined on the minimum and maximum separations between all node pairs.

7.7.2 Case 2

If U_i and/or U_j is defined, then `dist(`i,j`)` has an absolute minimum and/or an absolute maximum bound in addition to the soft bounds described above. A graph with this property, obtained by adding an upper-bound constraint to the graph in Figure 7.11, is shown in Figure 7.12. It will be shown shortly that the presence of U_i or U_j is one of two conditions under which absolute bounds on `dist(`i,j`)` exist. For the moment, i and j are required to be independent, as in Case 1.

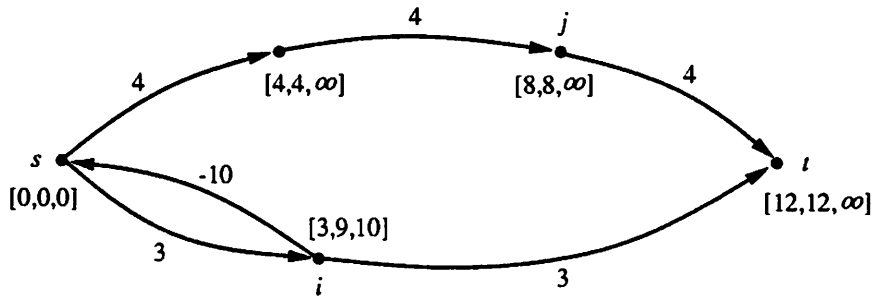


Figure 7.12: Graph with finite U_i, and with i and j independent.

The effect of adding the (i,s) constraint is that i's absolute maximum location is now defined, with value $U_i = |\mathtt{lp}(i,s)| = 10$. Since U_i is finite, there is an absolute limit on the rightward movement of i, which in turn limits the minimum i-j separation. This can be written as

$$\begin{aligned} L_{ij} &= L_j - U_i \\ &= -2. \end{aligned} \tag{7.10}$$

Compared to the soft minimum on $\texttt{dist}(i,j)$ (Equation 7.8), the absolute minimum is less restrictive in this example. However, L_{ij} is an **absolute minimum**; the only way to decrease the i-j separation further is to violate L_j, U_i, or both. If either $x_j < L_j$ or $x_i > U_i$, the constraints are not satisfied and the layout is illegal.

If U_j is finite, then a similar **absolute maximum** bound exists, which is given by

$$U_{ij} = U_j - L_i. \tag{7.11}$$

7.7.3 Case 3

The analyses in Cases 1 and 2 are valid when i and j are independent. Bounds on $\texttt{dist}(i,j)$ are derived in this subsection for the first of two cases wherein they are **dependent**; that is, when there *are* directed paths between i and j. Here, U_i and U_j are undefined.

The CPM, which is comprised of $\texttt{lp}(s)$ followed by $\texttt{lp}(t)$, computes L_k and U'_k for all nodes k. Placing all nodes such that $l_k = L_k$ constitutes a feasible solution, as does placing all nodes such that $l_k = U'_k$. However, other node-location assignments are not necessarily feasible. For the graph in Figure 7.13, it is not legal to place node i at U'_i and node j at L_j *simultaneously*, because this placement violates the (i,j) constraint. In Cases 1 and 2, the mixed assignments that were considered *are* feasible, due to the assumed independence of i and j.

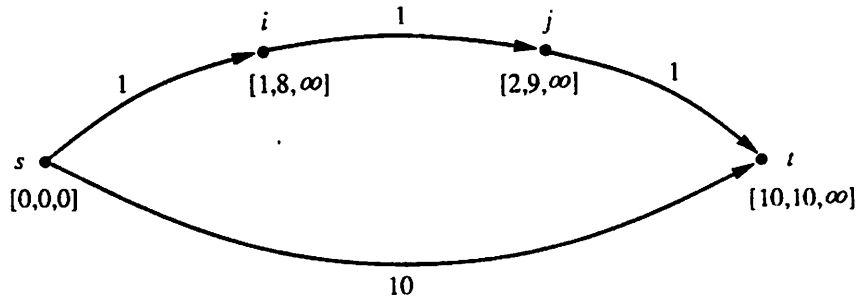


Figure 7.13: Graph with an i-j path.

The graph in Figure 7.13 contains a directed path from i to j. The existence of an i-j path implies that there is an **absolute lower bound** on $\texttt{dist}(i,j)$, regardless of whether U_i exists or not. The value of the absolute minimum separation is the length of the longest path from i to j, which is denoted $\texttt{lp}(i,j)$. The absolute and soft lower bounds

are thus

$$\begin{aligned} L_{ij} &= \mathtt{lp}(i,j) \\ &= 1 \\ L'_{ij} &= L_j - U'_i \\ &= -6. \end{aligned} \tag{7.12}$$

In this example, the absolute bound L_{ij} is more restrictive than the soft bound L'_{ij}.

The soft upper-bound U'_{ij} can be increased at will for the graph in Figure 7.13, by locating i at L_i and then increasing x_j as much as desired. Any location for j is legal, provided $x_j \geq x_i + 1$. This is true because $U_j = \infty$, *and* because there are no directed paths from j to i in G.

A modified version of the graph of Figure 7.13 is presented in Figure 7.14. The graph in Figure 7.14 has two paths from j to i, namely (j,i) of length -12 and (j,t), (t,i) of length -9. If $\mathtt{lp}(j,i)$ is finite, as it is in Figure 7.14, then $\mathtt{dist}(i,j)$ has an absolute maximum value, given by

$$U_{ij} = |\mathtt{lp}(j,i)|. \tag{7.13}$$

The absolute maximum bound U_{ij} is 9 in this case. If $x_j - x_i > 9$, then the constraint equation $x_t \leq x_i + 10$ is not satisfied. If $x_j - x_i > 12$, then the constraint equation $x_j \leq x_i + 12$ is not satisfied either. For the graph in Figure 7.14, the absolute interval is thus

$$\begin{aligned} D_{ij} &= [L_{ij}, U_{ij}] \\ &= [1,9]. \end{aligned}$$

7.7.4 Case 4

The final case allows for finite U_i and U_j, and for dependence (directed paths) between i and j. This is the worst case, in the sense that the movements of the nodes are more constrained than in the preceding three cases.

Absolute bounds on $\mathtt{dist}(i,j)$ arise from finite upper bounds on the positions of nodes i and j (Case 2), and from directed paths between them (Case 3). The graph in

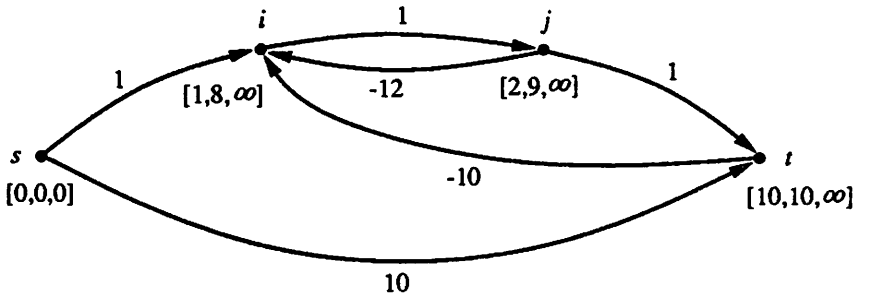


Figure 7.14: Graph of Figure 7.13 with j-i paths added.

Figure 7.12 has $U_i = 10$. The graph in Figure 7.15 has been obtained by adding a constraint to the graph from Figure 7.12. The finite value of U_i leads to

$$L_{ij} = L_j - U_i = -2 \tag{7.14}$$

according to Equation 7.10. The new graph also has an i-j path, hence

$$\mathtt{lp}(i,j) < \infty = 2. \tag{7.15}$$

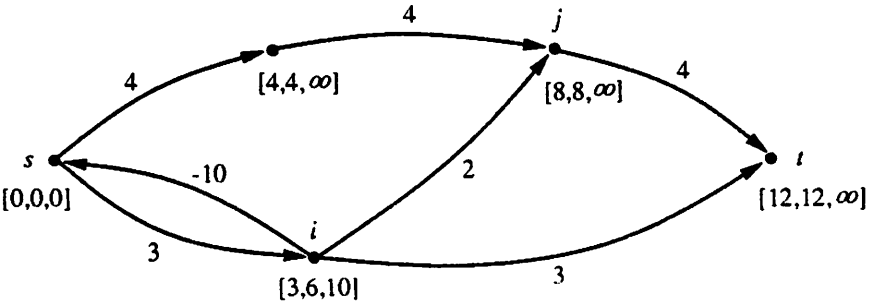


Figure 7.15: Graph with U_i and an i-j path.

It is clear that Equations 7.14 and 7.15 must both be satisfied. This is guaranteed by simply taking the more restrictive value, or

$$\begin{aligned} L_{ij} &= \mathtt{MAX}((L_j - U_i),\ \mathtt{lp}(i,j)) \\ &= \mathtt{MAX}(-2,\ 2) \\ &= 2. \end{aligned} \tag{7.16}$$

A dual argument leads to the following expression for U_{ij}, when U_j and one or more j-i paths exist:

$$U_{ij} = \texttt{MIN}(U_j - L_i,\ |\texttt{lp}(j,i)|) \tag{7.17}$$

7.7.5 Summary

The bounds on the achievable separation between a pair of nodes in G are

$$\begin{aligned} L'_{ij} &= L_j - U'_i, \\ U'_{ij} &= U'_j - L_i, \\ L_{ij} &= \texttt{MAX}((L_j - U_i),\ \texttt{lp}(i,j)), \\ U_{ij} &= \texttt{MIN}((U_j - L_i),\ |\texttt{lp}(j,i)|). \end{aligned}$$

The absolute bounds, which may or may not exist, arise from two sources – absolute node-position bounds, and directed paths between the nodes. In effect, the $\texttt{lp}(i,j)$ and $\texttt{lp}(j,i)$ terms model the direct influence of node i upon node j and vice versa. The $L_j - U_i$ and $U_j - L_i$ terms model the effect of all other constraints in the system on the i-j separation. These bounds are termed absolute because an illegal layout results if they are not obeyed.

The L_k and U'_k values are defined for all nodes k, therefore $D'_{ij} = [L'_{ij}, U'_{ij}]$ is defined for all node pairs. These soft bounds can be exceeded (subject to D_{ij}), but the extent of the resulting layout will be larger than its minimum value.

7.8 Dependent and Independent Pairs

In Section 7.6, two nodes were defined as independent if their positions did not affect one another. An analogous definition will be given here, regarding the distance intervals of *pairs* of nodes.

Let $D_{ij} = [L_{ij}, U_{ij}]$ denote the absolute distance interval for one pair of nodes i and j, and let $D_{mn} = [L_{mn}, U_{mn}]$ denote the absolute distance interval for the second pair m and n. In some cases, $x_j - x_i$ can be assigned any value in D_{ij} *without* affecting D_{mn}. When this is true, the pairs will be referred to as **independent**. For the graph in Figure 7.16, the pairs i-j and m-n are independent. In this case, the absolute bounds are

$$\begin{aligned} D_{i,j} &= [3.6], \\ D_{m,n} &= [2.4]. \end{aligned}$$

It is clear by inspection that the i-j separation does not affect the m-n separation and vice versa.

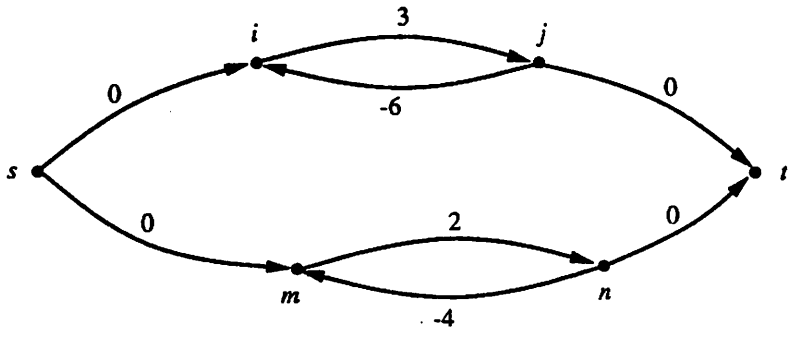


Figure 7.16: Independent pairs i-j and m-n.

If $x_j - x_i$ *does* affect D_{mn}, the pairs will be termed **dependent.** For the graph in Figure 7.17, the pairs are dependent. In this case, the absolute bounds are

$$\begin{aligned} D_{i,j} &= [2, \infty], \\ D_{m,n} &= [4, \infty]. \end{aligned}$$

Unfortunately, for the graph in Figure 7.17, *any* value assigned to `dist(`i, j`)` alters the bounds D_{mn} on the other pair, due to the (j, n) path. When `d(`i, j`)` $= 3$, for instance, the absolute bounds on D_{mn} become

$$D_{m,n} = [5, \infty].$$

This graph in fact has *no* feasible solution for the active constraint, due to the nature of the dependency between the pairs. The spacings cannot be made equal: the m-n spacing is always at least 2 units larger than the i-j spacing. Active constraints involving dependent pairs are not necessarily infeasible, however. Feasibility calculations, for both independent and dependent pairs, are presented later in this chapter.

7.8.1 Detecting Dependent Pairs

Fortunately, it is easy to detect whether or not two node pairs are dependent. It was shown in Section 7.6 that the position of some node a can affect the position of some other node b *if and only if* there is a directed path between them in the graph. This fact generalizes readily to the influence of one pair upon another.

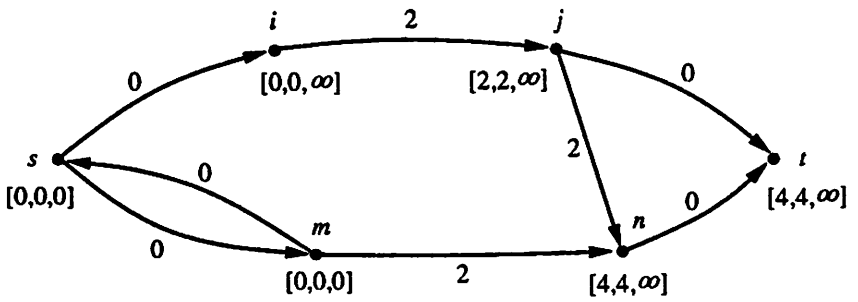


Figure 7.17: Dependent pairs i-j and m-n.

Consider two nodes, one from each pair of the active constraint, say i and m. Their positions are independent if there are no paths from i to m and no paths from m to i. When this is true, i can be placed anywhere within its interval $[L_i, U_i', U_i]$ without affecting the position of m, and vice-versa. If there *are* paths, then the positions of the two nodes are affected by one another.

If nodes i and m are independent, and if nodes j and m are also independent, then $\texttt{dist}(i,j)$ can be set to any value within its legal interval without affecting node m. In addition, if node n is also independent of nodes i and j, then n is unaffected by the i-j spacing as well. Under these conditions of no inter-pair paths, there are no inter-pair dependencies, and thus the **pairs** are **independent**. Each pair can be assigned any separation within its legal interval without affecting the bounds on the separation between the other pair.

The **independence condition** is thus that there be no i-m, i-n, j-m, j-n, m-i, m-j, n-i, or n-j paths in G. If one or more of these paths exists, the pairs are dependent and changing the spacing between one pair in general requires changing the spacing between the other.

It might appear that eight single-pair analyses are required to determine the independence/dependence of the two node pairs. However, this is not the case; all of the eight paths are checked for existence during the absolute pair-bounds calculation. In particular, the paths are discovered as follows:

i-m and i-n — in $\mathtt{lp}(i,j)$,
j-m and j-n — in $\mathtt{lp}(j,i)$,
m-i and m-j — in $\mathtt{lp}(m,n)$,
n-i and n-j — in $\mathtt{lp}(n,m)$.

Therefore the cost of resolving whether or not two pairs are independent is *zero*, because it is determined during the (mandatory) absolute-bounds calculation. During the absolute-bounds phase of the analysis, a flag is set if any of these paths exists, which thereby signals the need for the dependent-pair analysis described later in this chapter.

7.9 Feasibility of Independent Pairs

If the two pairs i-j and m-n are independent, then the feasibility of the active constraint

$$x_j - x_i = x_n - x_m$$

is determined solely by the absolute pair bounds D_{ij} and D_{mn}. If the absolute bounds exist for both pairs, then they must intersect and the domain of feasibility it given by

$$\mathbf{D} = D_{ij} \bigcap D_{mn} = [\mathbf{L}, \mathbf{U}]. \tag{7.18}$$

If D_{ij} and/or D_{mn} are not defined, then the active constraint is unconditionally feasible.

For the graph in Figure 7.16, the pairs i-j and m-n are independent, and the interval over which the active constraint is feasible is

$$\begin{aligned} \mathbf{D} &= D_{ij} \bigcap D_{mn} \\ &= [3,6] \bigcap [2,4] \\ &= [3,4]. \end{aligned}$$

7.10 Solution for Independent Pairs

When the two node pairs are independent, the feasibility requirement is that

$$\mathbf{D} = [\mathbf{L}, \mathbf{U}] \neq \emptyset.$$

In this section, it is assumed that the active constraint is feasible. Given a feasible active constraint, the problem is to select a value X such that $X = x_j - x_i = x_n - x_m$ lies in the

feasible region, and such that the compaction objectives are optimized. Once a value for X is selected, it is enforced by augmenting the graph with fixed constraints of value X between the i-j and m-n pairs. When all active constraints are thus processed, final positions are computed for all nodes via the usual graph-solving algorithms.

As described in Section 7.6, a separation value within the soft-bounds interval of a node pair, e.g., $X \in [L'_{ij}, U'_{ij}]$, can be achieved *without* affecting the pitch of the layout. That is, if P_{min} is the minimum layout pitch that is obtained when the active constraint is excluded, then i-j separation values within D'_{ij} will not lead to an increase in the pitch of the layout beyond P_{min}. On the other hand, if the soft interval is exceeded by k units either above or below, then the pitch grows by k units over P_{min}. Exceeding either D'_{ij} or D'_{mn} should thus be avoided if possible. If a soft interval must be exceeded by some amount k, the goal is to minimize k.

Assuming temporarily that $\mathbf{D} = [-\infty, \infty]$, the solution process begins with the computation of the intersection of the soft bounds of each pair. The intersection is denoted

$$\mathbf{D'} = D'_{ij} \bigcap D'_{mn} = [\mathbf{L'}.\mathbf{U'}].$$

Soft-bounds intervals always have finite lower and upper limits, therefore $\mathbf{D'}$ can be empty or finite, but not unbounded.

7.10.1 Non-intersecting Soft Bounds

If $\mathbf{D'} = \emptyset$, then there is a gap between the soft intervals as shown in Figure 7.18. Conceptually, the active constraint can be satisfied by extending the the lower interval upward, extending the upper interval downward, or both. *Because the pairs are independent,*

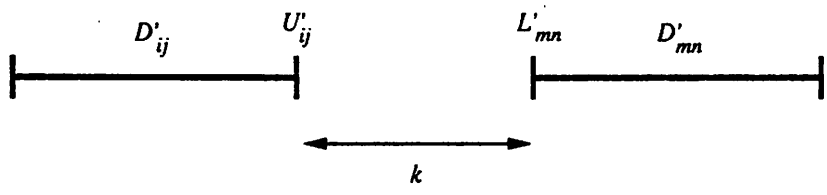


Figure 7.18: $\mathbf{D'} = \emptyset$.

the layout pitch is optimized by the assignment

$$X = \frac{\left(U'_{ij} + L'_{mn}\right)}{2}. \qquad (7.19)$$

which extends each interval by half the distance between them.

The reason for this is as follows. Consider the case wherein $L'_{mn} - U'_{ij} = k$, as shown in Figure 7.18. Selecting $X = U'_{ij}$ increases the pitch by k units, because the interval D_{mn} must then be extended *downward* by k units. Selecting $X = L'_{mn}$ increases the pitch by k units as well, because D'_{ij} must be extended *upward* by k units.

Choosing $X = U'_{ij} + k/2$ extends the lower interval upward by $k/2$ units. As noted above, enforcing the new i-j spacing is equivalent to adding a fixed constraint of value X between i and j. Since D'_{ij} has been exceeded, the longest path in G becomes $k/2$ units larger than P_{min}; furthermore, *the longest path must pass through the new (i, j) constraint.* The upshot is that some of the paths in G have an additional $k/2$ units of slack once the (i, j) constraint is added. In particular, any paths that limit the minimum m-n separation *must* fall into this category, because the pairs are independent. Therefore the m-n spacing can be increased by $k/2$ without any further increase in the pitch of the layout beyond $P_{min} + k/2$.

Thus, when $\mathbf{D'} = \emptyset$ and the pairs are independent, the optimum value for X is half of the separation k between the soft intervals of the pairs. This causes each interval to grow by $k/2$ units. But, since the pairs are independent, growing both by $k/2$ grows the pitch by only $k/2$. This is not necessarily the case if the pairs are dependent, as will be shown later.

7.10.2 Intersecting Soft Bounds

If the soft intervals *do* intersect, i.e.,

$$\mathbf{D'} = [\mathbf{L'}, \mathbf{U'}] \neq \emptyset.$$

then the active constraint can be satisfied without increasing the pitch of the layout beyond P_{min}. Since any value $X \in [\mathbf{L'}, \mathbf{U'}]$ does not affect the pitch, the optimization criterion is to satisfy the active constraint within $\mathbf{D'}$ such that the wire length is minimized. A representative case is shown in Figure 7.19.

The method used in SPARCS in this circumstance is an extension of the slack-distribution strategy described in the preceding chapter. As described before in Section 6.9, the attractive force between two nodes g and h for slack distribution is a positive integer, denoted W_{gh}.

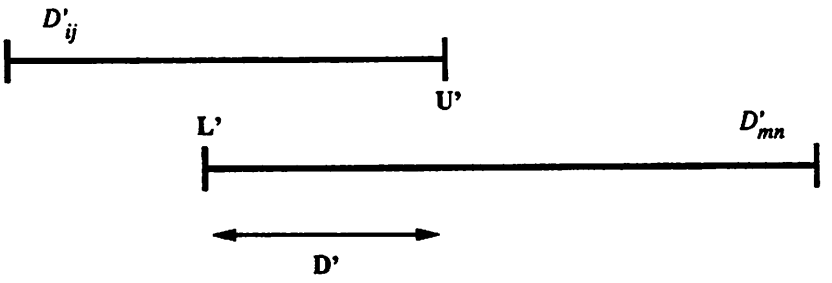


Figure 7.19: $\mathbf{D'} = [\mathbf{L'}, \mathbf{U'}]$.

For a node pair i-j of an active constraint, W_{ij} may not exist. However, other slack weights involving i and j do exist, e.g., between i and other nodes in G. As a result, a slack weight is computed for each of the four nodes individually. The node weights are signed integers; a positive value means that there is a net force to the right on the node, and a negative value corresponds to a net force to the left. The weight for node g is denoted w_g.

The node weights are used to compute a weight for each pair. For the i-j pair, and assuming i is to the left of j, the weight for the pair is given by

$$w_{ij} = w_i - w_j.$$

The pair's weight can be positive, negative, or zero, which correspond to the desired effects on the i-j spacing listed in the following table.

w_{ij}	i-j Spacing
positive	shrinks
negative	grows
zero	no preference

If w_{ij} and w_{mn} are both positive (negative), then both spacings should grow (shrink). If the signs are different, then the value with the larger magnitude determines X. The result is calculated by taking the sum

$$\omega = w_{ij} + w_{mn}$$

and assigning $X = x_j - x_i = x_n - x_m$ as indicated in the table below.

ω	X
positive	$\mathbf{L'}$
negative	$\mathbf{U'}$
zero	$(\mathbf{L'} + \mathbf{U'})/2$

This strategy of taking the extreme value of the slack interval is consistent with minimization of the L^1 norm of the constraint system [24].

7.10.3 Effect of Finite Absolute Bounds

Thus far it has been assumed that the region of absolute feasibility is the set of all integers. The modification needed when the region is finite, i.e., when $\mathbf{D} = D_{ij} \bigcap D_{mn} = [\mathbf{L}, \mathbf{U}]$, is described in this subsection.

It is possible to account for the effect of finite absolute bounds during the computation of X in each of the situations described above. For example, if $\mathbf{D}' = \emptyset$ and $\mathbf{D}$ is finite, there are several possible configurations, including those shown in Figure 7.20. Referring to Figure 7.20, if $\mathbf{D_1}$ is the absolute interval the optimum X is $\mathbf{U_1}$. This case could be recognized initially and the averaging computation (Equation 7.19) could be avoided. Also, cases like $\mathbf{D_2}$ could be recognized as having no effect on the computation of X. Similar remarks apply when $\mathbf{D}' \neq \emptyset$.

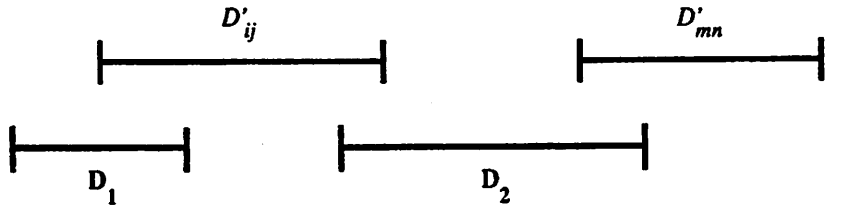


Figure 7.20: Two possible absolute intervals.

An alternative approach is to ignore $\mathbf{D}$ initially and compute X as already described. Then, if

$$X \bigcap \mathbf{D} = \emptyset,$$

X is set to $\mathbf{L}$ or $\mathbf{U}$, whichever is closer. This mechanism leads to the same result, but it enables a simpler and more structured implementation; hence it has been employed in SPARCS.

7.10.4 Summary

The flow of the computing $X = x_j - x_i = x_n - x_m$, when the pairs are independent, is as follows.

1. Compute D_{ij} and D_{mn};
2. If $D_{ij} \cap D_{mn} = \emptyset$ stop (infeasible);
3. Otherwise compute D'_{ij} and D'_{mn};
4. If $D'_{ij} \cap D'_{mn} = \emptyset$
   (a) If $U'_{ij} < L'_{mn}$ then $X = (U'_{ij} + L'_{mn})/2$;
   (b) Else $X = (U'_{mn} + L'_{ij})/2$;
5. Otherwise
   (a) Set X according to slack calculation;
6. If $X \cap \mathbf{D} \neq \emptyset$ then
   (a) X is legal;
7. Otherwise
   (a) Move X to nearest value in $\mathbf{D}$;
8. Add fixed constraints of value X between i and j, m and n.

7.11 The Dependent-Pairs Problem

When the two node pairs of an active constraint are dependent, more computation is required to determine feasibility than in the independent-pairs case. Fortunately, the cost of detecting that two pairs are dependent is zero, thus the additional checking is avoided when the pairs are independent.

The absolute bounds on `dist`(i, j) and `dist`(m, n) give the *independently achievable limits* on the separation of pairs i-j and m-n. When two pairs are dependent, it is not generally true that their absolute bounds are *simultaneously* valid. Another way of viewing the situation is that the absolute bounds on the second pair may be altered when the spacing for the first pair is set to a particular value. It was mentioned previously that setting `d`(i, j) is mathematically equivalent to putting a fixed constraint between i and j in G. When the new fixed constraint is added, the absolute bounds on the other pair may be altered if the pairs are dependent.

7.11.1 Examples

The example presented in Figure 7.5 is reproduced in Figure 7.21. The active constraint of interest is

$$x_a - x_s = x_t - x_a,$$

and the absolute bounds are

$$\begin{aligned} D_{sa} &= [2, \infty], \\ D_{at} &= [2, \infty]. \end{aligned}$$

The pairs are dependent in this case, because there are paths from both nodes in the first pair (s and a) to both nodes in the second pair (a and t).

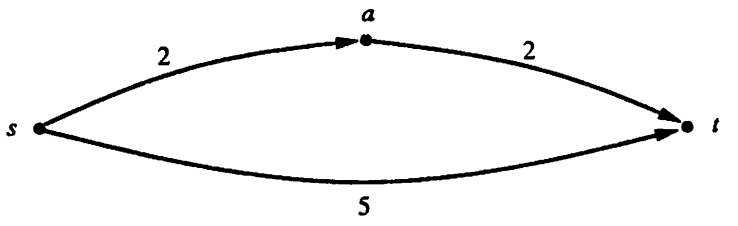


Figure 7.21: Desired active constraint is $x_a - x_s = x_t - x_a$.

Suppose an attempt is made to satisfy the active constraint by choosing $X = 2$, which is within both D_{sa} and D_{at}. Choosing the s-a pair as the "independent" variable and setting its spacing to $2 = x_a - x_s$ leads to the augmented graph shown in Figure 7.22. In the augmented graph, the absolute bounds on the a-t separation are different than before; i.e.,

$$\begin{aligned} D_{sa} &= [2.2]. \\ D_{at} &= [3.\infty]. \end{aligned}$$

Hence the value $X = 2$ is not a feasible active-constraint value, because both pairs cannot simultaneously take on this value.

This active constraint is indeed feasible; the question is, at what value(s) of X is it satisfied. It is clear that X must be at least 3, by the results of trying $X = 2$ on D_{at}. It is thus reasonable to check whether $X = 3$ is a feasible value. Setting $X = 3$ produces the

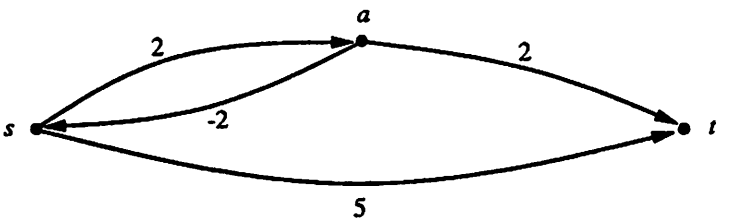


Figure 7.22: $X = x_a - x_s = 2$ is infeasible.

graph is shown in Figure 7.23, for which the bounds are

$$\begin{aligned} D_{sa} &= [3, 3], \\ D_{at} &= [2, \infty]. \end{aligned}$$

Since $D_{sa} \cap D_{at} = 3$, $X = 3$ is a valid solution for the active constraint $x_a - x_s = x_t - x_a$. In fact, $X = 3$ is the optimal solution for this graph, because any larger value increases the pitch. In this example the inter-pair dependencies do change the absolute bounds, but the constraint is still satisfiable.

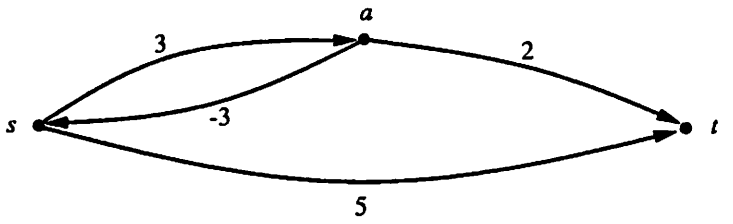


Figure 7.23: $X = x_a - x_s = 3$ is a valid solution.

The active constraint

$$x_b - x_a = x_c - x_a$$

applied to the example shown in Figure 7.24 (from Figure 7.6) is *never* satisfiable. Again there are inter-pair paths, hence the pairs are dependent. The absolute bounds are

$$\begin{aligned} D_{ab} &= [1, \infty], \\ D_{ac} &= [2, \infty]. \end{aligned}$$

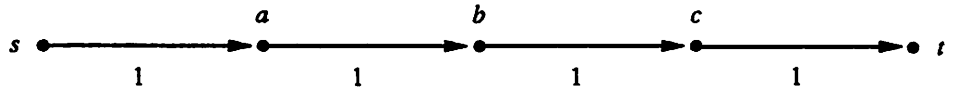


Figure 7.24: Desired active constraint is $x_b - x_a = x_c - x_a$.

Obviously, the minimum solution that might be feasible is $X = 2$. Setting $X = 2 = x_b - x_a$ produces the augmented graph in Figure 7.25. As in the previous example, the lower bound on the second pair, L_{ac}, increases as a result. The new bounds are

$$\begin{aligned} D_{ab} &= [2, 2], \\ D_{ac} &= [3, \infty]. \end{aligned}$$

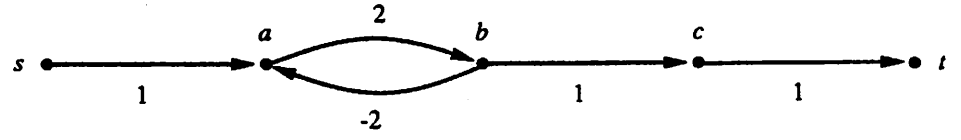


Figure 7.25: $X = x_b - x_a = 2$ is infeasible.

Trying $X = 3$ as in the previous example does not lead to a satisfiable constraint. Rather than decreasing or staying the same, L_{ac} increases again when X is increased from 2 to 3, i.e.,

$$\begin{aligned} D_{ab} &= [3, 3]. \\ D_{ac} &= [4, \infty] \end{aligned}$$

for $X = 3$. Indeed, by inspecting the graph, it can be seen that for any $X = \alpha$,

$$\begin{aligned} D_{ab} &= [\alpha, \alpha]. \\ D_{ac} &= [\alpha + 1, \infty]. \end{aligned}$$

In contrast with the preceding example, the inter-pair dependencies in this case change the region of feasibility such that the constraint is *never* satisfiable.

Unlike the independent-pairs case, the absolute bounds do not provide enough information to determine active-constraint feasibility when the pairs are dependent. As

shown by the above examples, some or even all values $X \in \mathbf{D}$ may not be satisfiable. For an active constraint

$$x_j - x_i = x_n - x_m,$$

what is desired is an indication of whether the i-j spacing affects the m-n spacing, and if so, in what manner. The following section address this question.

7.12 Properties of Dependent Pairs

To determine feasibility when two node pairs are dependent, it is necessary to compute the effect of changing the spacing between one pair on the absolute bounds on the separation of the second pair. The change in D_{mn} as $\texttt{dist}(i,j)$ is swept over its legal range of values is thus considered in this section. It is convenient to begin by determining the effects, due to changes in $\texttt{dist}(i,j)$, on the bounds of a single dependent node q.

7.12.1 Bounds of a Dependent Node

Without loss of generality, it will be assumed that node i is bound to the source node s with a fixed constraint of value 0. Therefore, sweeping $\texttt{dist}(i,j)$ over $[L_{ij}, U_{ij}]$ becomes one-to-one with sweeping x_j over the interval $[L_j, U_j]$, and derivatives with respect to x_j are equivalent to derivatives with respect to $x_j - x_i = \texttt{dist}(i,j)$. As previously stated, setting $\texttt{dist}(i,j)$ to some value k means that a fixed constraint of value k must be added between i and j in G. Hence setting $\texttt{dist}(i,j) = k = \texttt{d}(i,j)$ creates a bidirectional path between j and i, *and* a bidirectional path between j and s due to the assumed s-i constraint.

The following lemmas state some properties regarding the potential changes in L_q and U_q that follow from increasing $\texttt{d}(i,j)$ from $L_{ij} \equiv L_j$ to $U_{ij} \equiv U_j$. The prototype graph in Figure 7.26 is useful in visualizing the lemmas.

Lemma 7.1 (tight paths) *Increasing* $\texttt{dist}(i,j)$ *causes* L_q *to change if and only if there is a tight path from* s *to* j *to* q. *Likewise, increasing* $\texttt{dist}(i,j)$ *causes* U_q *to change if and only if there is a tight path from* q *to* j *to* s.

Proof: L_j and L_q are determined by tight s-j and s-q paths, respectively. Increasing x_j therefore increases L_j. An increase in L_j causes a corresponding increase in L_q if and only if there is an j-q path which is tight as well. Similarly, U_q increases only if there is a q-j-s path that dominates all q-s paths. If this is the case, the q-j constraint must be tight.

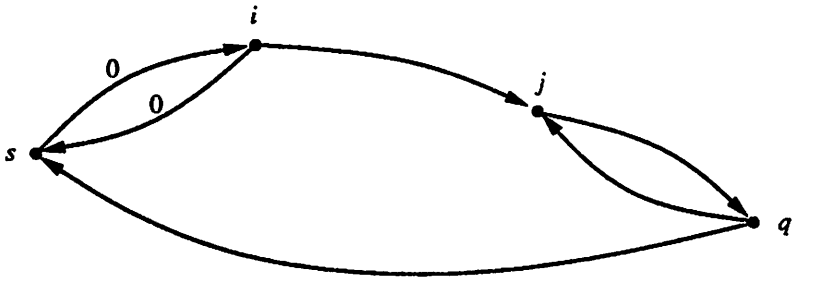


Figure 7.26: Bounds of node q are dependent on $\mathtt{d}(i,j)$.

Lemma 7.2 (monotonicity) *The bounds on q are monotonic with respect to* $\mathtt{dist}(i,j)$*: i.e., regardless of the direction of the path between j and q, an increase in* $\mathtt{dist}(i,j)$ *cannot cause a decrease in L_q or U_q. Furthermore, dL_q/dx_j and dU_q/dx_j equal 0 or 1.*

Proof: In the constraint equations, e.g., $x_q \geq x_j + k_1$ and $x_q \leq x_j + k_2$, all variables have the same sign, thus changing one leads to either no change or a change in the same direction in the other. By Lemma 7.1, $\mathtt{dist}(i,j)$ affects L_q and/or U_q if and only if there exist paths between j and q that are tight. Hence changing x_j leads to either a change in the same direction in L_q and/or U_q (if the corresponding paths are tight), or to no change (if the paths are slack or nonexistent). The values of the derivatives follow directly from the constraint equations and whether or not the paths are tight.

Lemma 7.3 (path tightening) *Suppose there is a j-q path in G which is loose at $x_j = L_j$. As* $\mathtt{dist}(i,j)$ *is increased, the path can become tight, after which it never becomes loose again. That is, a j-q path can go from loose to tight, but not vice-versa, as x_j sweeps from L_j to U_j.*

Proof: Say the j-q path is tight for $x_j = x_1$. To become loose at $x_j = x_1 + 1$, the longest path to q must increase by 2 or more over its length at $x_j = x_1$. However, the coefficients of the variables in the constraint equations are 1, so an increase of one unit in any variable can increase another by at most one. Thus, when x_j increases from x_1 to $x_1 + 1$, x_q increases by 1 and the j-q path remains tight.

Lemma 7.4 (path loosening) *As* $\mathtt{dist}(i,j)$ *sweeps from L_j to U_j, a q-j-s path can go from tight to loose but not vice-versa.*

Proof: Assume there are 2 paths, one from q to s and another from q to j to s, and that the q-j-s path is initially tight. That is, the length of the q-j-s path is longer than the q-s

path. Increasing x_j by 1 *decreases* the length of the j-s path by 1, hence the q-j-s path length decreases by 1. Eventually, the q-j-s path becomes shorter than the q-s path, after which the q-j path is no longer tight.

These lemmas lead to several conclusions regarding the functional relationships between $\texttt{dist}(i,j)$ and L_q, and between $\texttt{dist}(i,j)$ and U_q. In particular, as $\texttt{dist}(i,j)$ is swept over its range of legal values from L_{ij} to U_{ij}, the limits on node q behave as follows:

Theorem 7.1 (slope transitions)
The rate of change of L_q either remains 0, remains 1, or goes from 0 to 1 over $\texttt{dist}(i,j) \in [L_{ij}, U_{ij}]$. The transition from 1 to 0 cannot occur. The rate of change of U_q either remains 0, remains 1, or goes from 1 to 0 over $\texttt{dist}(i,j) \in [L_{ij}, U_{ij}]$. The transition from 0 to 1 cannot occur.

Proof: From Lemmas 7.2, 7.3, and 7.4.

The behavior of the limits as specified in Theorem 7.1 is shown graphically in Figure 7.27. It is important to note that *functional relationships other than these are not possible.*

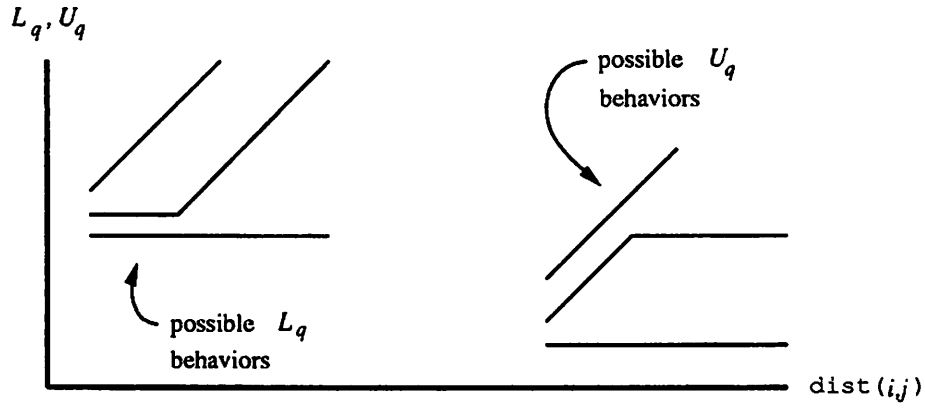


Figure 7.27: All possible behaviors of L_q and U_q, by Theorem 7.1.

7.12.2 Lower Bound of a Dependent Pair

The lower bound on D_{mn} is given by

$$L_{mn} = \texttt{MAX}((L_n - U_m), \texttt{lp}(n,m)).$$

Since the choice of $\mathtt{d}(i,j)$ does not affect $\mathtt{lp}(n,m)$ when all four nodes are distinct, the situation of interest in this section is that wherein $L_n - U_m$ dominates L_{mn}, i.e.,

$$L_{mn} = L_n - U_m. \tag{7.20}$$

Here, the behavior of L_{mn} as a function of $\mathtt{dist}(i,j)$ is deduced from the results of the preceding subsection. The (non-restrictive) assumptions are retained, thus $x_j \equiv \mathtt{dist}(i,j)$, $L_j \equiv L_{ij}$, $U_j \equiv U_{ij}$, etc. as before.

Table 7.3 summarizes the behavior of the slope of L_q, denoted dL_q/dx_j, due to a change in $\mathtt{dist}(i,j)$. The first column gives the slope at the lower limit of $\mathtt{dist}(i,j)$, the second column gives the slope at the upper limit, and the third column indicates any change that occurs in the path from j to q. A corresponding summary for dU_q/dx_j is presented in Table 7.4.

dL_q/dx_j @ L_{ij}	dL_q/dx_j @ U_{ij}	j-q path
1	1	tight — tight
0	0	loose — loose
0	1	loose — tight

Table 7.3: Summary of dL_q/dx_j.

dL_q/dx_j @ L_{ij}	dL_q/dx_j @ U_{ij}	q-j path
1	1	tight — tight
0	0	loose — loose
1	0	tight — loose

Table 7.4: Summary of dU_q/dx_j.

The slope of L_{mn} can be determined by differentiating Equation 7.20 to yield

$$\frac{dL_{mn}}{dx_j} = \frac{dL_n}{dx_j} - \frac{dU_m}{dx_j} \tag{7.21}$$

(which is the same as differentiating with respect to $x_j - x_i$ under the current assumptions). The slope of L_{mn}, and thus the shape of $L_{mn} = F(\mathtt{dist}(i,j))$, can now be constructed from the information in Tables 7.3 and 7.4. It is apparent that the slope of L_{mn} can only take on the three values presented in Table 7.5. Furthermore, several important conclusions regarding the transitions that dL_{mn}/dx can make are presented as Theorem 7.2.

dL_q/dx_j	dU_q/dx_j	dL_{mn}/dx
0	1	−1
0	0	0
1	1	0
1	0	1

Table 7.5: All possible values of of dL_{mn}/dx.

Theorem 7.2 (slope monotonicity) *As* $\texttt{dist}(i,j)$ *increases from* L_{ij} *to* U_{ij}, *the slope of* $L_{mn} = F(\texttt{dist}(i,j))$ *either remains constant or increases. The slope cannot decrease.*

Proof: The slope can take on the three values in Table 7.5 only. There are four cases. The statements below are supported by Theorem 7.1.

Case 1: Initially $dL_{mn} = -1$, $dL_n = 0$, $dU_m = 1$. dL_n can change from 0 to 1 only; dU_m can change from 1 to 0 only. Each change increases the slope by 1.

Case 2: Initially $dL_{mn} = 0$, $dL_n = 1$, $dU_m = 1$. dL_n must remain 1; if dU_m changes from 1 to 0, the slope increases to 1.

Case 3: Initially $dL_{mn} = 0$, $dL_n = 0$, $dU_m = 0$. dU_m must remain 0; if dL_n changes from 0 to 1, the slope increases to 1.

Case 4: Initially $dL_{mn} = 1$, $dL_n = 1$, $dU_m = 0$. dU_m must remain 0 and dL_n must remain 1, thus the slope cannot change.

Theorem 7.2 is summarized schematically in Figure 7.28. The solid segments denote the initial behavior of each of the cases enumerated in the theorem. The dashed segments indicate the effects of the possible changes in dL_n and dU_m in each instance.

An active constraint that is feasible with respect to L_{mn} satisfies $L_{mn} \leq \texttt{d}(i,j)$ for at least one value in the interval $\texttt{dist}(i,j) \in [L_{mn}, U_{mn}]$. The region of feasibility is thus the set of integer-valued points on or below the line $\texttt{dist}(i,j) = L_{mn}$ on a plot such as Figure 7.28. This fact, along with Theorem 7.2, leads to a number of conclusions regarding active-constraint feasibility. Two statements can be made immediately.

Theorem 7.3 *If the active constraint is feasible with respect to* L_{mn} *at some value* $\texttt{d}(i,j)$, *say* d_1, *then it is feasible for any larger value of* $\texttt{dist}(i,j)$.

Proof: Since d_1 is feasible, $L_{mn}(d_1) \leq d_1$. To be infeasible at a larger separation d_2 requires $L_{mn}(d_2) > d_2$. This can only occur if $dL_{mn}/dx > 1$, which is impossible.

Theorem 7.4 *If the active constraint is infeasible at some separation* d_1, *then it is infeasible for any smaller value of* $\texttt{dist}(i,j)$. *In particular, if it is infeasible at* U_{ij}, *then it is*

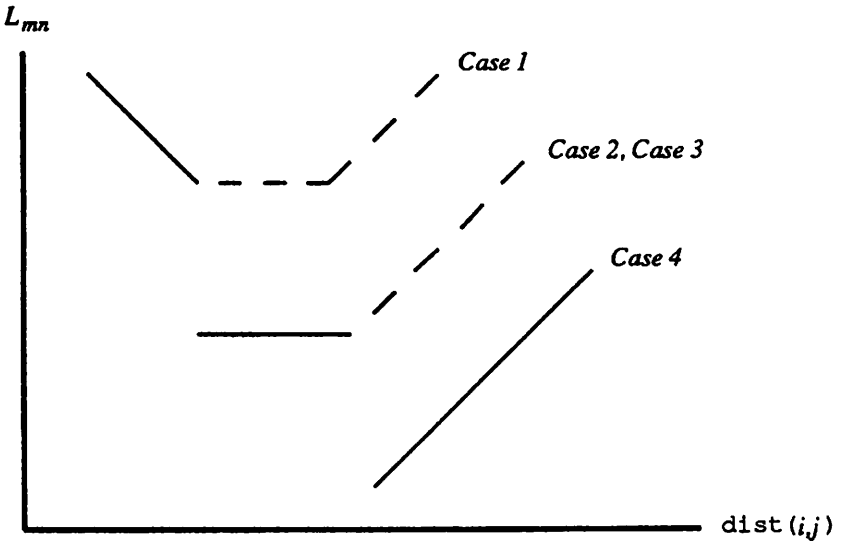


Figure 7.28: Possible behavior of $L_{mn} = F(\texttt{dist}(i,j))$, from Theorem 7.2.

infeasible for all legal values of $\texttt{dist}(i,j)$.

Proof: Similar to that of Theorem 7.3.

Theorems 7.3 and 7.4 can be used to begin developing a strategy for solving the dependent-pairs problem. If the active constraint is found to be infeasible with respect to L_{mn} at $\texttt{d}(i,j) = d_1$, there is no point in trying to enter the feasible region by decreasing $\texttt{dist}(i,j)$. However, the constraint *may* become feasible if $\texttt{dist}(i,j)$ is increased to some value $d_2 > d_1$. If it does become feasible at d_2, there is no concern about it becoming infeasible again at some $\texttt{d}(i,j) > d_2$.

One remaining question is, if $x_j - x_i = x_n - x_m$ is infeasible at d_1, at what value of $\texttt{d}(i,j) > d_1$ might it become feasible? Obviously, if $dL_{mn}/dx_j = 1$ at d_1 the constraint never becomes feasible, according to Theorem 7.2. However, if the slope at d_1 is -1 or 0 then L_{mn} might enter the feasible region at a larger $\texttt{d}(i,j)$ value. Theorem 7.5 gives a bound on the increase in $\texttt{d}(i,j)$ that is necessary to reach the feasible region. Note that $z = L_{mn} - d_1$ is the amount that L_{mn} is outside of the feasible region when $\texttt{d}(i,j) = d_1$ as shown in Figure 7.29.

Theorem 7.5 *Let* $z = L_{mn} - d_1$ *be the amount by which* $L_{mn} = F(\texttt{dist}(i,j))$ *is infeasible at* $\texttt{d}(i,j) = d_1$. *The constraint must become feasible at or before* $d_2 = d_1 + z$; *otherwise it is always infeasible.*

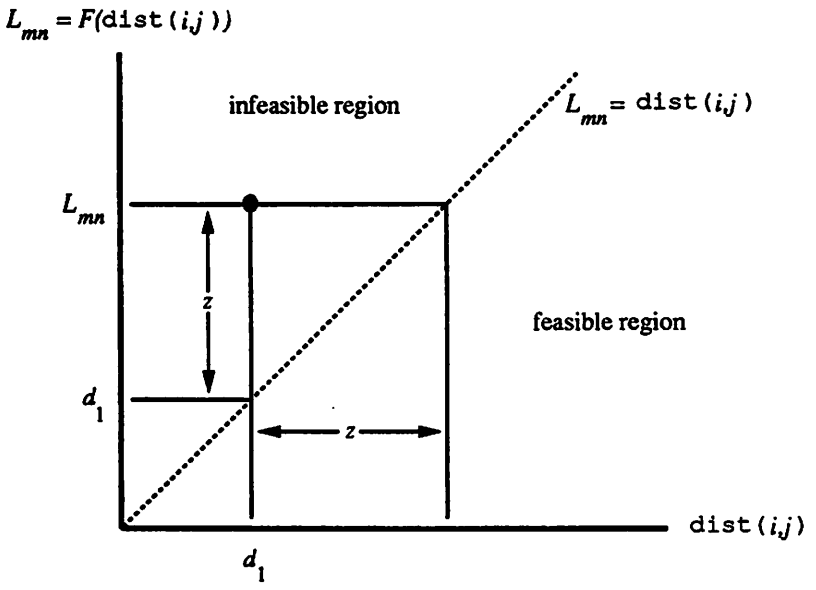


Figure 7.29: Distance from infeasible value to feasible region.

Proof: By the geometry of the feasible region (Figure 7.30) and the possible slope values of L_{mn}, L_{mn} reaches the feasible region at $\texttt{d}(i,j) = d_1 + z/2$ if the slope remains -1 from d_1 to $d_1 + z/2$. L_{mn} reaches the feasible region at $d_2 = d_1 + z$ if the slope of L_{mn} remains 1. If L_{mn} is also infeasible at d_2, then the slope must change to 1 outside of the feasible region and the constraint cannot be satisfied at any $\texttt{d}(i,j)$.

7.12.3 Upper Bound of a Dependent Pair

The behavior of U_{mn} as a function of $\texttt{dist}(i,j)$ is the dual of the behavior of L_{mn}. Since the analysis for U_{mn} is essentially the same as that presented in the last subsection, only the results are given here. The upper bound on D_{mn} is given by

$$U_{mn} = U_n - L_m$$

when the length of the n-m path does not dominate; the corresponding slope of U_{mn} is

$$\frac{dU_{mn}}{dx_j} = \frac{dU_n}{dx_j} - \frac{dL_m}{dx_j}.$$

As in the case of L_{mn}, the slope of U_{mn} takes on the values -1, 0, or 1 only. The slope of U_{mn} is a *decreasing*, rather than increasing function of $\texttt{dist}(i,j)$, as stated in Theorem 7.6.

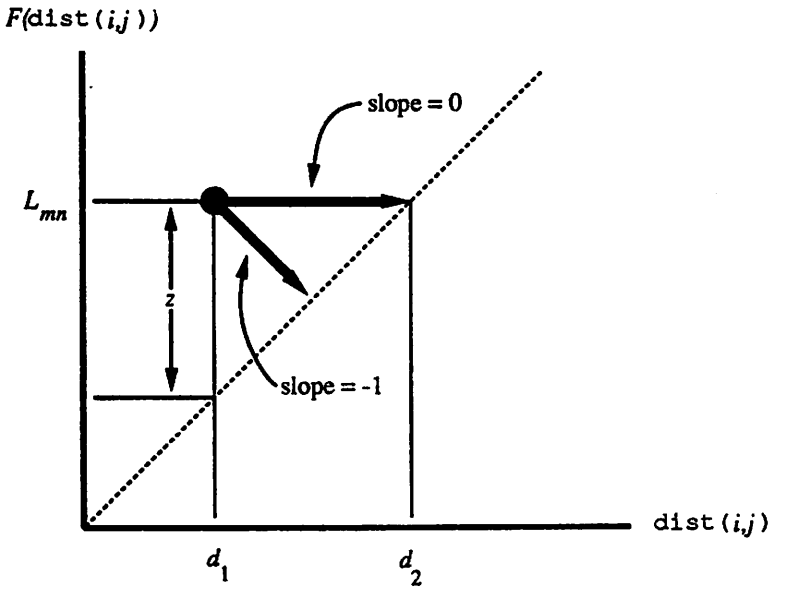


Figure 7.30: Possible paths from infeasible value into feasible region.

Theorem 7.6 is sketched in Figure 7.31. The region of feasibility with respect to the upper bound U_{mn} is the set of integer-valued points *on* or *above* the line $\texttt{dist}(i,j) = U_{mn}$.

Theorem 7.6 *As* $\texttt{dist}(i,j)$ *increases from* L_{ij} *to* U_{ij}, *the slope of* $U_{mn} = F(\texttt{dist}(i,j))$ *either remains constant or decreases. The slope cannot increase.*
Proof: Similar to Theorem 7.2.

Theorems 7.7 and 7.8 are the duals of Theorems 7.3 and 7.4, respectively. If the active constraint is infeasible with respect to U_{mn} at $\texttt{d}(i,j) = d_1$, it might become feasible if $\texttt{dist}(i,j)$ is *reduced* (rather than increased) to some value $d_0 < d_1$. Theorem 7.9 is equivalent to Theorem 7.5; it gives a bound on the maximum decrease in $\texttt{dist}(i,j)$ that is necessary to reach the feasible region, if it is reachable.

Theorem 7.7 *If the active constraint is feasible with respect to* U_{mn} *at some value of* $\texttt{d}(i,j)$, *say* d_1, *then it is feasible for any smaller value of* $\texttt{dist}(i,j)$.
Proof: Similar to Theorem 7.3.

Theorem 7.8 *If the active constraint is infeasible at some separation* d_1, *then it is infeasible for any larger value of* $\texttt{dist}(i,j)$. *If it is infeasible at* L_{ij}, *then it is infeasible for all*

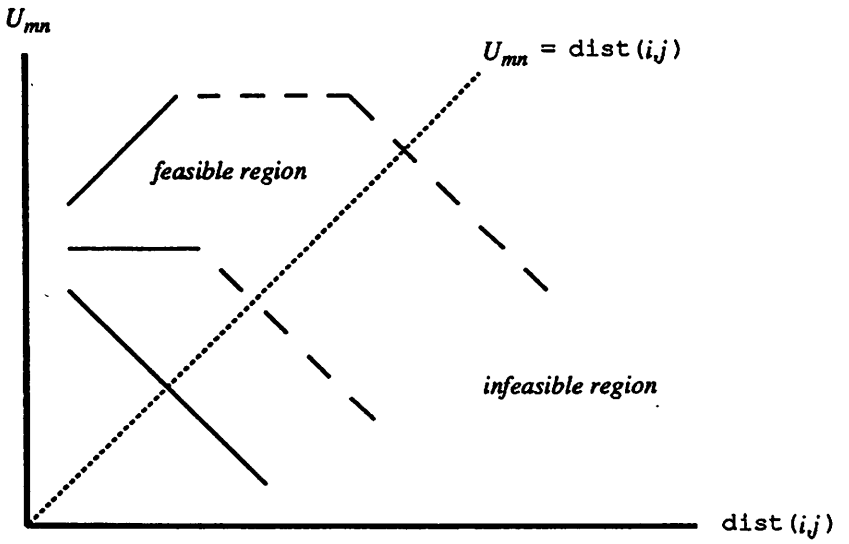


Figure 7.31: Possible behavior of $U_{mn} = F(\texttt{dist}(i,j))$, from Theorem 7.6.

legal values of $\texttt{dist}(i,j)$.

Proof: Similar to that of Theorem 7.4.

Theorem 7.9 *Let* $z = d_1 - U_{mn}$ *be the amount by which* $U_{mn} = F(\texttt{dist}(i,j))$ *is infeasible at* $\texttt{d}(i,j) = d_1$. *The constraint must become feasible within* $d_0 = d_1 - z$; *otherwise it is always infeasible.*

Proof: Similar to Theorem 7.5.

These properties lead directly to the proposed algorithm for solving the dependent-pairs problem which is described in the next section.

7.13 Dependent-Pairs Solution

One possible approach to solving the active-constraint problem, when the node pairs are dependent, is to compute an explicit relationship between the pairs. That is, one could designate one of the pairs, say i-j, as the independent pair, then compute a function for the dependent pair's separation

$$\texttt{d}(m,n) = F(\texttt{dist}(i,j)).$$

Note that $\texttt{dist}(m,n)$ is in general an *interval*, not a single value, for each value of $\texttt{dist}(i,j)$.

The function F could be tabulated by computing the D_{mn} bounds for every legal value $\texttt{dist}(i,j) \in D_{ij}$. This approach would be quite expensive, unless $\texttt{dist}(i,j)$ is only legal over a narrow interval. Also, the absolute intervals D_{ij} and D_{mn} are not necessarily bounded, in which case it is unclear how to choose the domain for the independent pair.

Another approach to computing F is to construct an explicit mapping between $\texttt{dist}(m,n)$ and $\texttt{dist}(i,j)$. In principle, this can be done by analyzing the various dependencies between the nodes in each pair, as was done earlier in this chapter to compute the absolute and soft bounds on node-pair separations. However, the large number of possible inter-pair relationships means that the number of cases to be investigated is large, thus this approach would also be very expensive.

7.13.1 Proposed Solution

The proposed solution method for dependent pairs makes use of the properties just derived regarding the behavior of dependent pairs. Using these properties, a method has been developed wherein the function F *is sampled twice, at most*, for each active constraint. This method is generally much more efficient than either of the above approaches.

The first few steps are identical to the independent-pairs algorithm, namely:

1. Compute D_{ij} and D_{mn};
2. If $D_{ij} \cap D_{mn} = \emptyset$ stop (infeasible) Otherwise continue;
3. Compute D'_{ij} and D'_{mn};
4. Compute X according to $D'_{ij} \cap D'_{mn}$;
5. Adjust X according to $X \cap \mathbf{D}$ if necessary.

At this point, the value X has been optimized and it it guaranteed to be realizable by either pair individually. It remains to be determined whether X can be simultaneously realized by both pairs. If not, it is necessary to determine an adjusted value for X that *is* simultaneously realizable, if such a value indeed exists.

The next step is therefore to set

$$x_j - x_i = X$$

by adding a fixed constraint to G between i and j. The modified graph is used to compute a new absolute-bounds interval for the m-n spacing. If the new interval satisfies

$$L_{mn} \leq X \leq U_{mn},$$

X is the simultaneously-feasible value that optimizes the objectives; a fixed constraint of value X is then added between m and n, completing the processing.

On the other hand, the new bounds on $\texttt{dist}(m, n)$ may not intersect X. In this case, X is increased or decreased according to Theorem 7.5 or 7.9. The adjusted value for X, denoted X^*, may or may not lie within the bounds for the *independent* pair. If it does, i.e., if $L_{ij} \leq X^* \leq U_{ij}$, the processing continues; otherwise the constraint is infeasible. Assuming the new value X^* is within D_{ij}, the i-j fixed constraint is updated and D_{mn} is recomputed. If X^* is still outside D_{mn} the constraint is infeasible over the entire domain by Theorems 7.5 and 7.9. If this value for X^* does lie within D_{mn} the constraint is feasible, X^* is the optimized value, so a fixed constraint is then added between m and n to complete the processing.

7.13.2 Summary

When the pairs are dependent, the additional steps following the calculation of the initial value X are as follows:

1. Choose one pair, say i-j, to be the independent pair and set its value to X by adding a fixed constraint to G;
2. Recompute D_{mn} using the augmented graph with $\texttt{d}(i, j) = X$;
3. If $L_{mn} \leq X \leq U_{mn}$ the constraint is satisfied; add a fixed constraint of value X between m and n and return;
4. Otherwise, compute z;
5. If z is outside D_{ij}, the constraint is infeasible; remove the i-j fixed constraint and return;
6. Otherwise, compute $X^* = X \pm z$;
7. Modify the i-j fixed constraint to set its value to X^* and recompute D_{mn};

8. If $L_{mn} \leq X^* \leq U_{mn}$ the constraint is satisfied; add a fixed constraint of value X^* between m and n and return;

9. Otherwise, the constraint cannot be satisfied; remove the i-j fixed constraint and return.

7.14 Multiple Active Constraints

For a problem with a single active constraint, the proposed algorithm is optimal. Multiple active constraints are handled by processing each constraint sequentially. The algorithm does not synthesize a processing order for the constraints. Rather, they are processed in the order specified by the user when the constraints are entered.

The sequential processing of multiple constraints does not guarantee a globally-optimal result. However, it represents a reasonable compromise between optimality and runtime efficiency. The problem of processing multiple active constraints simultaneously is vastly more complex than the problem that has been addressed. It does not appear likely that an efficient algorithm can be developed for simultaneously processing multiple constraints.

7.15 Final Graph Solution

As the values are determined for each active constraint, fixed constraints are added to the graph to guarantee that the selected values will be realized. When all the active constraints have been processed, the perturbed graph is solved by the CPM algorithms to produce the final coordinates for all of the layout elements.

7.16 Complexity Analysis

In addition to forming the basis for the proposed active-constraint algorithm, the feasibility analysis dominates the computation time for an active constraint.

To compute the feasibility of

$$x_j - x_i = x_n - x_m,$$

the absolute bounds for each pair D_{ij} and D_{mn} are required. As shown earlier, the absolute lower and upper bounds are given by equations of the form

$$\begin{aligned} L_{ij} &= \texttt{MAX}((L_j - U_i),\ \texttt{lp}(i,j)), \\ U_{ij} &= \texttt{MIN}((U_j - L_i),\ |\texttt{lp}(j,i)|). \end{aligned}$$

The following path analyses therefore appear to be needed for the initial computation of D_{ij} and D_{mn}.

$\texttt{lp}(s)$:	yields L_i, L_j, L_m, L_n
$\texttt{lp}(i,s)$:	yields U_i
$\texttt{lp}(j,s)$:	yields U_j
$\texttt{lp}(m,s)$:	yields U_m
$\texttt{lp}(n,s)$:	yields U_n
$\texttt{lp}(i,j)$:	yields a component of L_{ij}
$\texttt{lp}(j,i)$:	yields a component of U_{ij}
$\texttt{lp}(m,n)$:	yields a component of L_{mn}
$\texttt{lp}(n,m)$:	yields a component of U_{mn}

It seems that one single-source analysis plus eight single-pair analyses are required for the initial feasibility calculation. However, this is not the case. The algorithm has been described in terms of single-pair analyses to help clarify its presentation. In its actual implementation in SPARCS, all graph searches are single-source analyses. For example, an analysis such as $\texttt{lp}(i,s)$ is never performed; instead, a single-source analysis from i (denoted $\texttt{lp}(i)$) is used. The single-source analysis produces the longest paths from i to *all nodes reachable* from node i, including both s and j, if they are indeed reachable. As a result, the following four single-source calculations are used to produce the same information as the eight single-pair analyses listed above.

$\texttt{lp}(i)$:	replaces $\texttt{lp}(i,s)$, $\texttt{lp}(i,j)$
$\texttt{lp}(j)$:	replaces $\texttt{lp}(j,s)$, $\texttt{lp}(j,i)$
$\texttt{lp}(m)$:	replaces $\texttt{lp}(m,s)$, $\texttt{lp}(m,n)$
$\texttt{lp}(n)$:	replaces $\texttt{lp}(n,s)$, $\texttt{lp}(n,m)$

If $D_{ij} \cap D_{mn}$, the constraint might be feasible and the next step is to determine a value X for the constraint via the soft bounds on each pair. The soft bounds are given by equations of the form

$$\begin{aligned} L'_{ij} &= L_j - U'_i, \\ U'_{ij} &= U'_j - L_i. \end{aligned}$$

The absolute lower-bounds are known for all nodes from $\texttt{lp}(s)$; the soft upper-bounds are not. As a result $\texttt{lp}(t)$ must be computed to enable the calculation of D'_{ij} and D'_{mn}. Given D_{ij}, D'_{ij}, D_{mn}, and D'_{mn}, the cost of computing $X = x_j - x_i = x_n - x_m$ is very low and can be ignored for the present purposes.

If the pairs are independent the active constraint is fully processed once X has been selected. The cost of evaluating an active constraint when the pairs are independent is therefore equal to the cost of six single-source longest-path analyses.

When the pairs are dependent, X may not be realizable by both pairs simultaneously. As described in Section 7.13, the pairs are checked to determine if X is simultaneously realizable by setting $\texttt{dist}(i,j)$ to X and recomputing D_{mn}. If X is not realizable by both pairs at the same time, $\texttt{dist}(i,j)$ is set to $X^* = X \pm z$ and D_{mn} is recomputed once more. It was proven earlier in this chapter that these two samples of the function $\texttt{dist}(m,n) = F(\texttt{dist}(i,j))$ are sufficient to determine a feasible value for the active constraint, if such a value exists. The graph searches $\texttt{lp}(s)$ (to update L_m and L_n), $\texttt{lp}(m)$ (to update U_m and $\texttt{lp}(m,n)$), and $\texttt{lp}(n)$ (to update U_n and $\texttt{lp}(n,m)$) are repeated each time $\texttt{dist}(m,n) = F(\texttt{dist}(i,j))$ is sampled. The computational cost for a dependent pair, above that of an independent pair, is thus the cost of either three or six single-source analyses of G.

The performance that is achievable in practice for the $\texttt{lp}(s)$ computation, which visits all nodes and edges in G, is nearly linear in the size of G. Single-source longest-path analyses from nodes other than s are faster than $\texttt{lp}(s)$ because only a subgraph of G is visited. This is due to the fact that, for nodes other than s, only a subset of the nodes is typically reachable.

Each active constraint requires at least six but no more than twelve longest-path analyses of G. At worst four of them visit the entire graph; the remaining analyses only visit subgraphs. Strictly speaking, setting the value of an active constraint increases the size of G, but by at most two fixed constraints. This increase in the number of constraints is insignificant in practical cases, as the number of conventional constraints is usually much larger than the number of active constraints. The size of G can thus be regarded as constant.

In summary, each active constraint requires between six and twelve path analyses on G. The time required for each analysis is bounded above by the time required to perform $\texttt{lp}(s)$. Thus the overhead for processing a single active constraint is constant, under the (reasonable) assumption that the increase in the size of G due to the two additional

fixed constraints is negligible. For n active constraints, the added complexity beyond a conventional compaction without active constraints is therefore $O(n)$.

This analysis indicates that the complexity of the proposed algorithm, in theoretical terms, is very good. The examples presented in the next section show that the actual performance achieved by the proposed algorithm, on practical examples, is also quite acceptable.

7.17 Examples

Two practical examples are presented in this section to demonstrate the implementation of the SPARCS active-constraint algorithm. In the first example, active constraints are used to maintain hierarchy during pitch-matching. In the second example, active constraints are used to maintain symmetry during the compaction of an analog amplifier circuit.

7.17.1 Hierarchy Preservation

The example used here is taken from one of the benchmarks used in the 1987 ICCD compaction session [8]. The circuit is from a CMOS technology with two levels of metal. Counting the wiring channel as a cell, there are a total of four cells, three of which are unique, arranged as shown in Figure 7.32. The symbolic layout of the example, before compaction, is shown in Figure 7.33. The goal in this case is to perform a pitch-matching hierarchical compaction such that the two instances of **c16_2**, which are identical before compaction, are identical after compaction as well.

The result of a conventional compaction of this example by SPARCS is given in Figure 7.34. It is obvious that the two instances of **c16_2** are no longer identical, and the hierarchy has been lost.

During the horizontal pitch-matching step, if the corresponding pins of the two instances of **c16_2** are related by active constraints as outlined in Section 7.2.2, the result shown in Figure 7.35 is obtained by SPARCS. There are seven terminals on the top and bottom of **c16_2**. Each adjacent terminal pair of the top instance is constrained to the corresponding pair of the bottom instance via an active constraint. In addition, the source and the first terminal of each instance are likewise constrained together. There are thus a total of seven active constraints employed. Since the active constraints force the pin

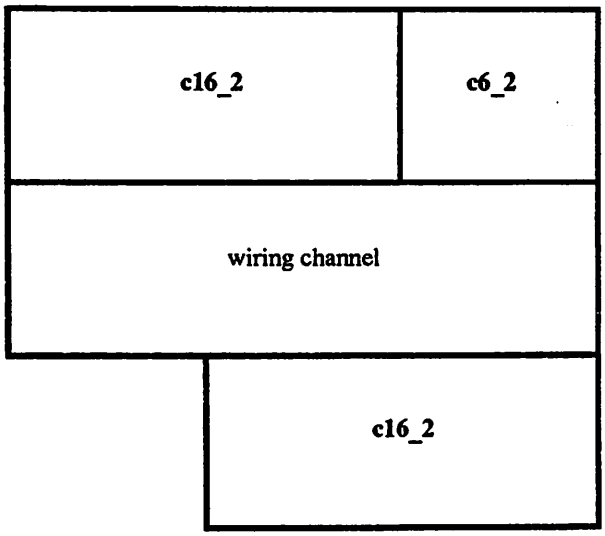


Figure 7.32: Floorplan for the pitch-matching example.

separations of both copies of **c16_2** to change such that they remain the same, the hierarchy has been maintained in the final result as shown in Figure 7.35.

The quantity of interest in runtime performance is the graph solution time for the horizontal pitch-matching step, both with and without active constraints. Without active constraints (Figure 7.34), the graph solution time is .50 seconds.[2] With all seven active constraints included, the graph solution time is 7.36 seconds. Including a single active constraint results in a graph solution time of 1.33 seconds. Hence the runtime for graph solution grows by a factor of 5.5 when the number of active constraints grows by a factor of 7, which agrees with the theoretical $O(n)$ complexity. At present, SPARCS does not produce port-abstraction graphs. Each instance is thus represented by its full constraint graph in this example. The use of port-abstraction graphs for the instances would substantially decrease the graph-solution times, both with and without active constraints.

7.17.2 Symmetry

The second example is a bipolar differential amplifier layout taken from an actual industrial design. The design methodology used to create the original circuit is similar to the gate-array methodology. A standard template cell is employed, which is comprised of a number of pre-fabricated resistors, bipolar transistors, and capacitors. A particular circuit

[2]The runtimes in this section were measured on an IBM RTPC model 125 running AIX version 2.2.1.

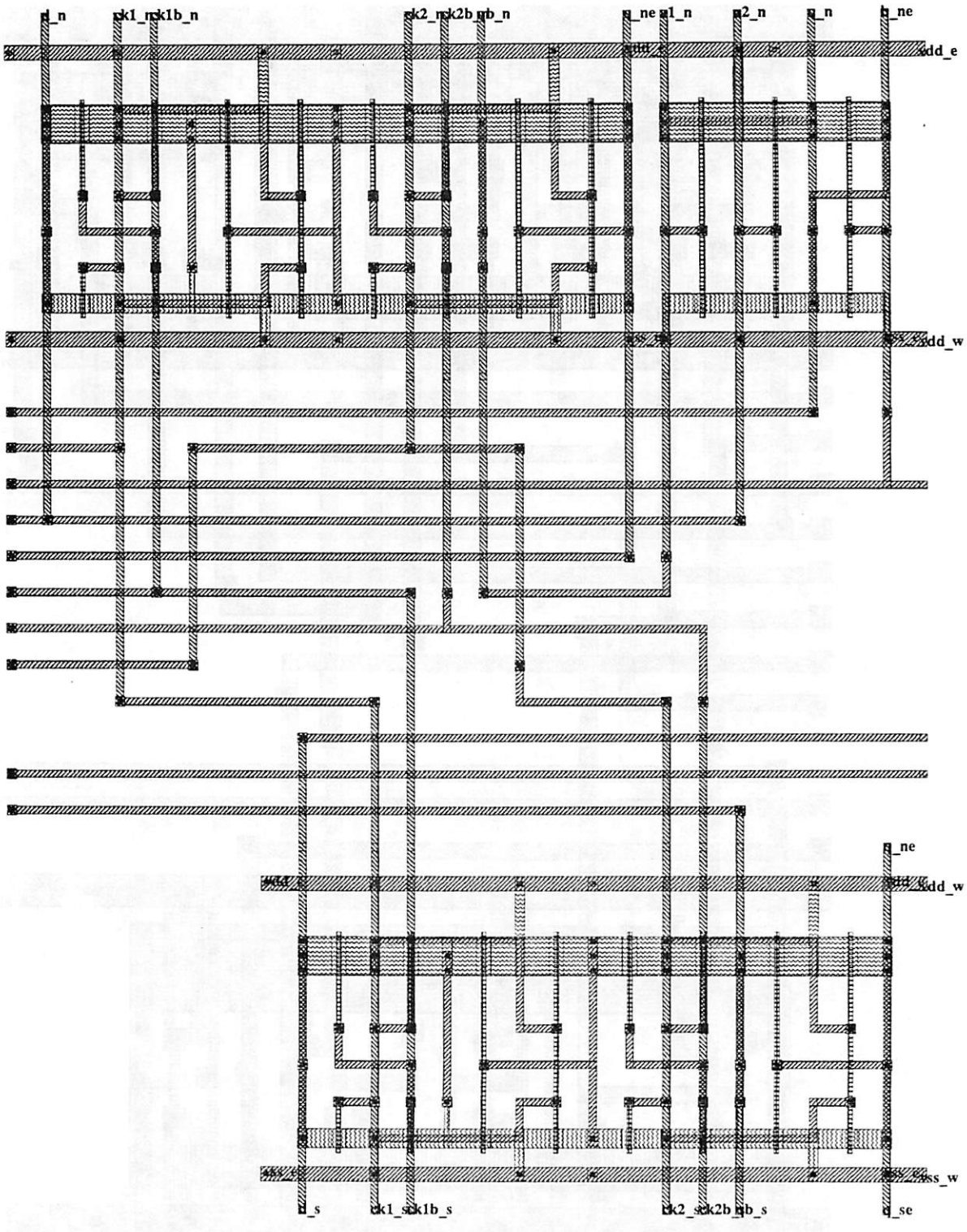


Figure 7.33: Symbolic layout before compaction.

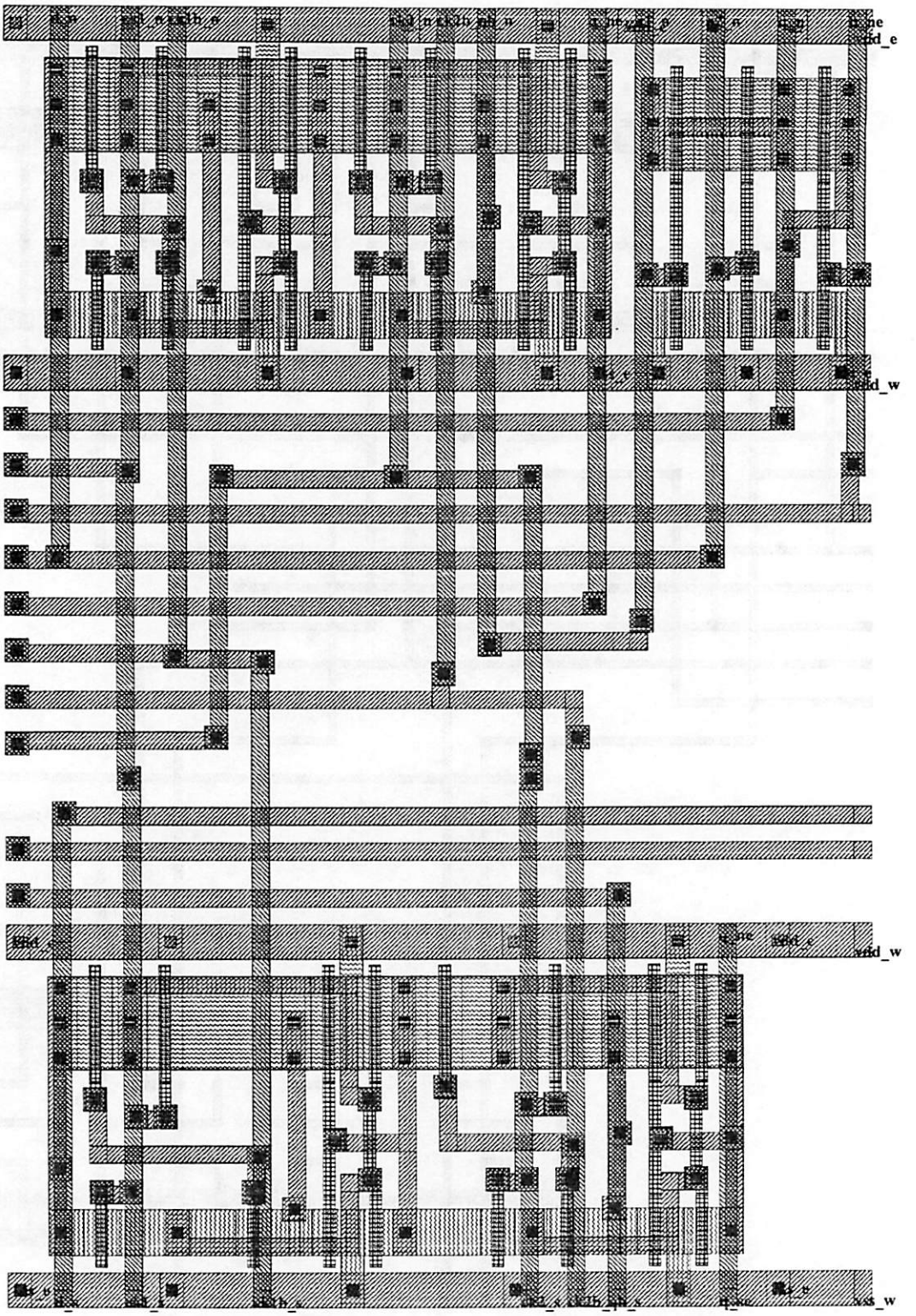


Figure 7.34: Result obtained via conventional pitch-matching.

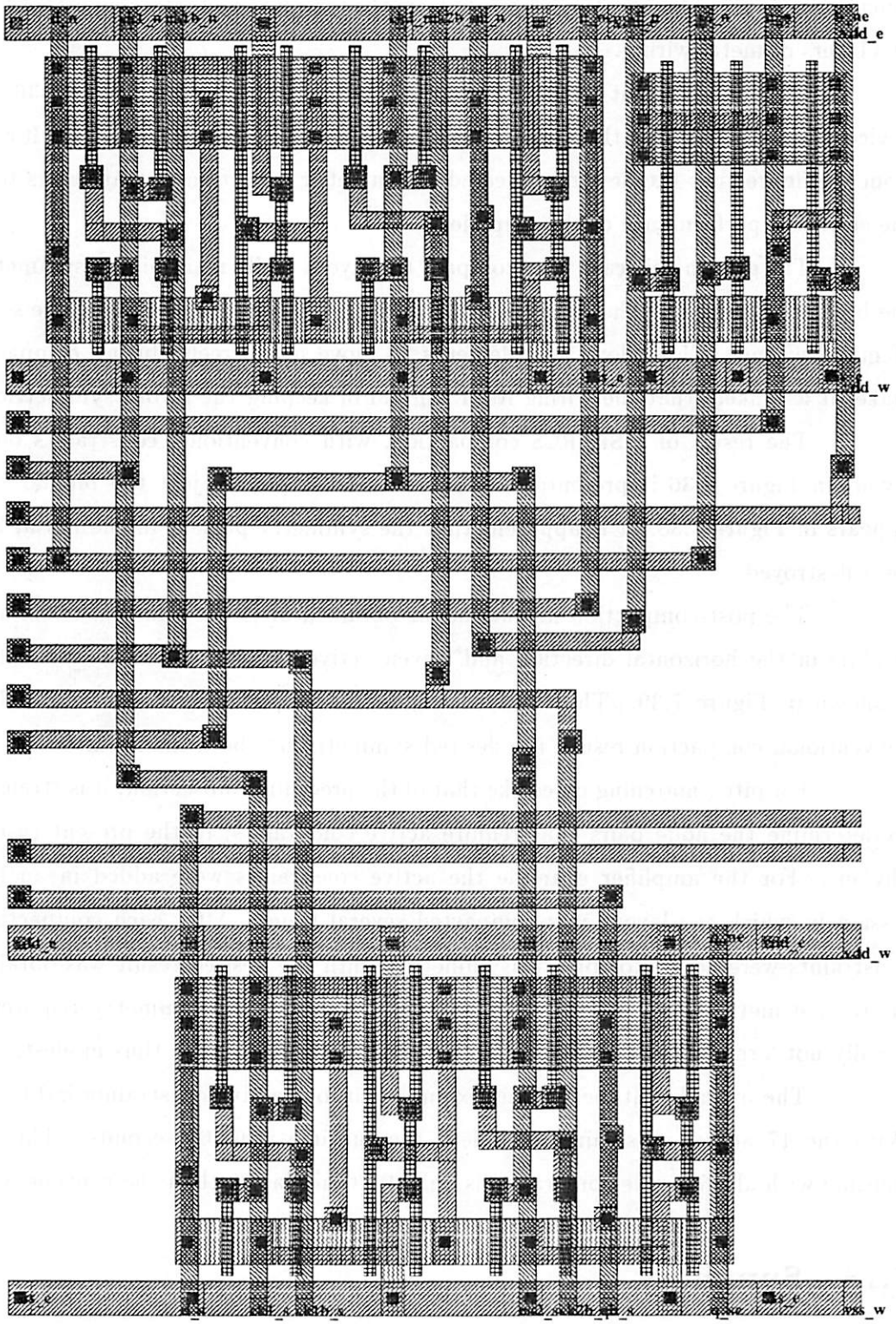


Figure 7.35: Result obtained with active constraints added.

function is obtained by selecting elements from the template and interconnecting them using two layers of metal wiring.

The initial layout of the differential amplifier is shown in Figure 7.36. Only the devices from the template that are used in the implemented circuit are shown. It can be seen from the figure that the designer selected a symmetric collection of transistors to optimize the electrical performance of the amplifier.

The goal in this case is to compact the layout while maintaining symmetry among the bipolar transistors. The problem was made more difficult by removing the second level of metal wiring. This allows the elements to move more freely during compaction, and makes it less likely that the wiring itself will aid in keeping the layout symmetric.

The result of a SPARCS compaction, with conventional constraints only, of the layout in Figure 7.36 is presented in Figure 7.37. A plot of just the bipolar transistors appears in Figure 7.38. It is apparent that the symmetry present in the initial layout has been destroyed.

The post-compaction layout that is obtained after the addition of six active constraints in the horizontal direction and eleven active constraints in the vertical direction is shown in Figure 7.39. The transistors alone are depicted in Figure 7.40. Unlike the conventional compaction result, the desired symmetry has been maintained.

For pitch-matching cases like that of the preceding subsection, it is straightforward to determine the node pairs that require active constraints; in the present case it is less obvious. For the amplifier example the active constraints were added in an interactive session in which the layout was compacted several times. After each compaction, active constraints were added to correct asymmetries until the desired result was obtained. This interactive methodology is reasonable, because circuits with symmetry requirements are usually not large and the number of active constraints required is thus modest.

The overall runtime for this example without active constraints is 14.77 seconds. With the 17 active constraints included, the runtime is 62.60 seconds. Thus the total runtime with all 17 active constraints is only 4.2 times larger than the runtime with none.

7.18 Summary

A new, four-variable constraint termed an active constraint has been proposed. The utility of active constraints in solving layout compaction problems that cannot be

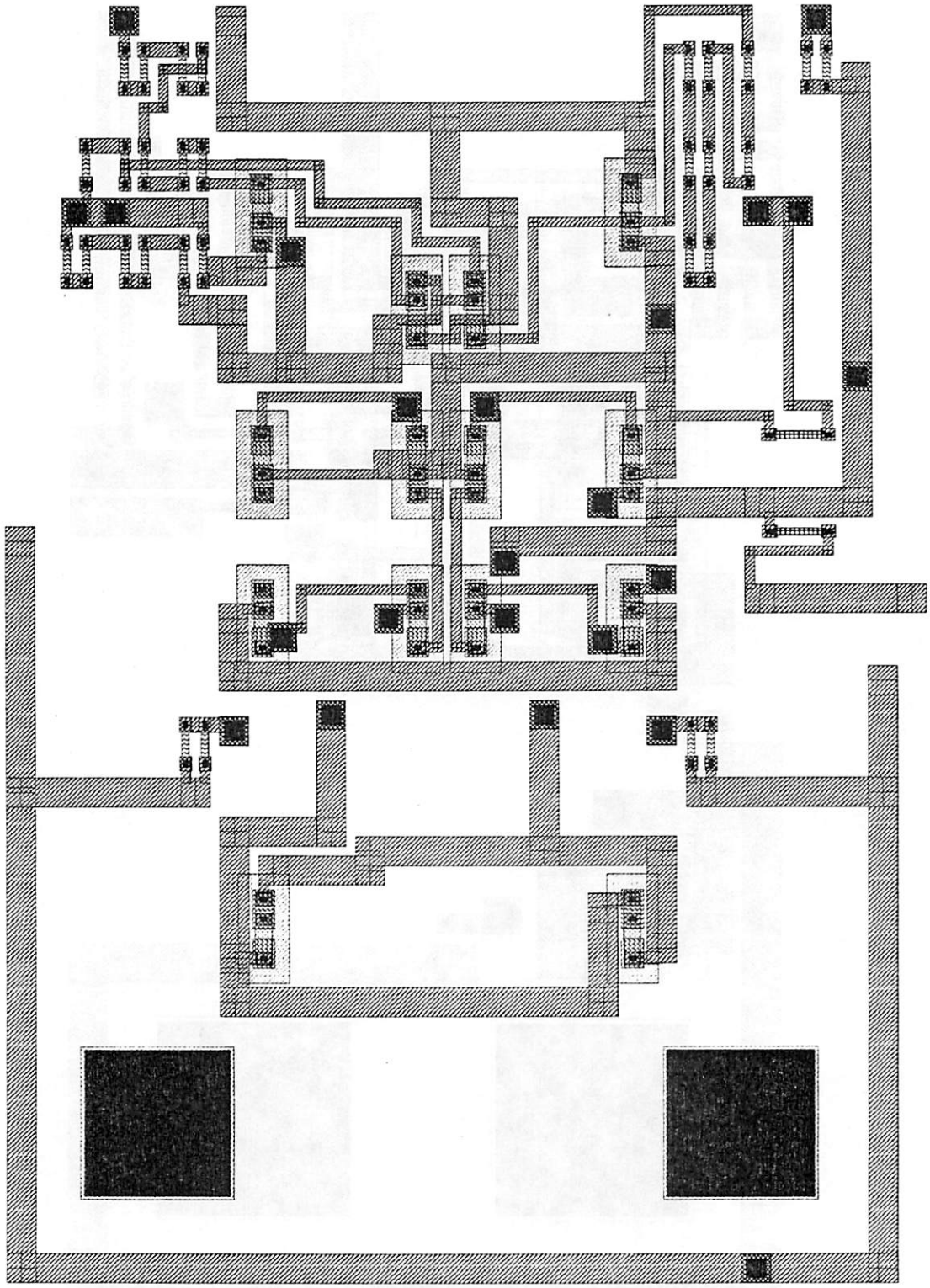


Figure 7.36: Initial layout of the differential amplifier.

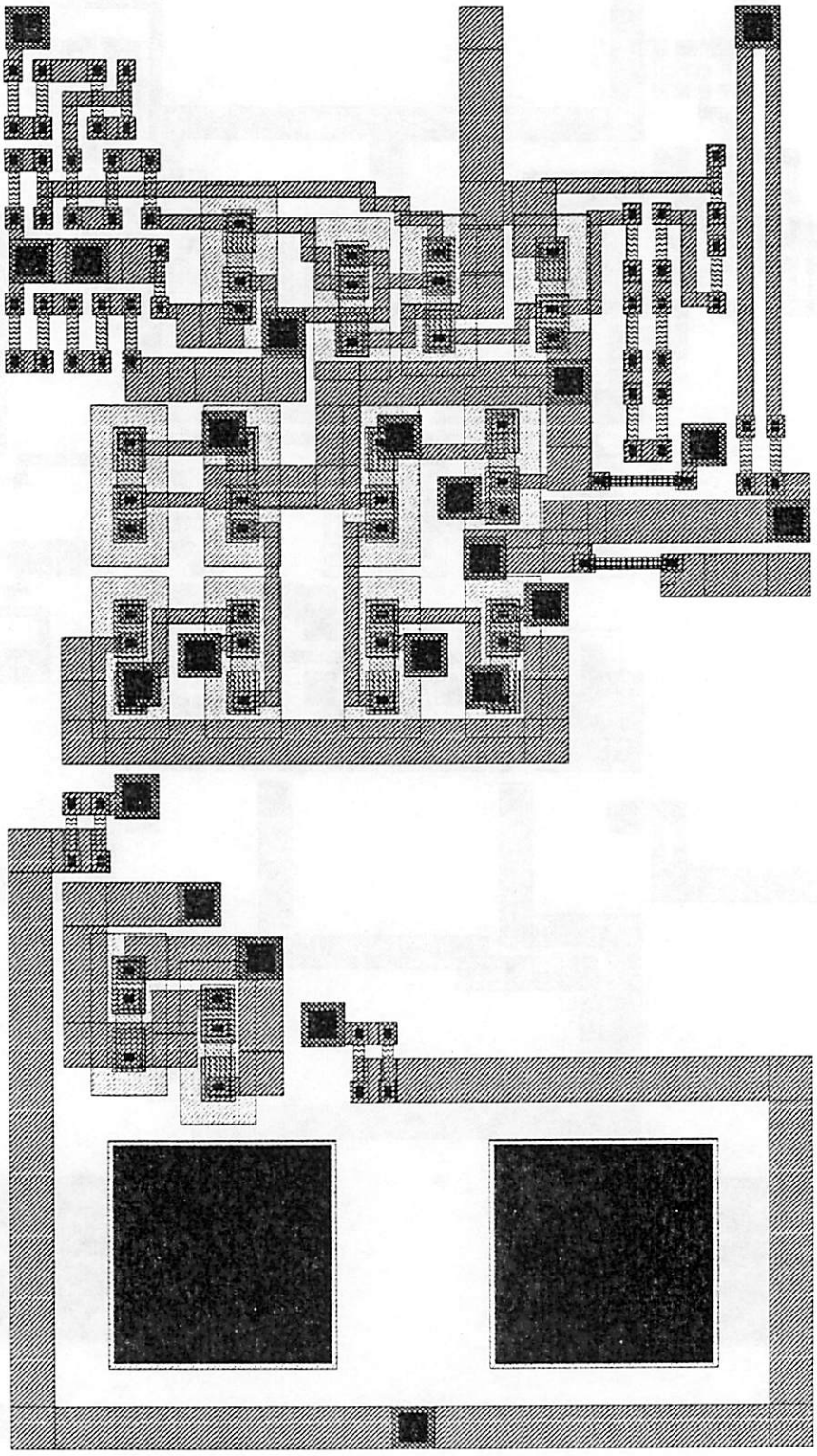


Figure 7.37: Amplifier after a conventional compaction.

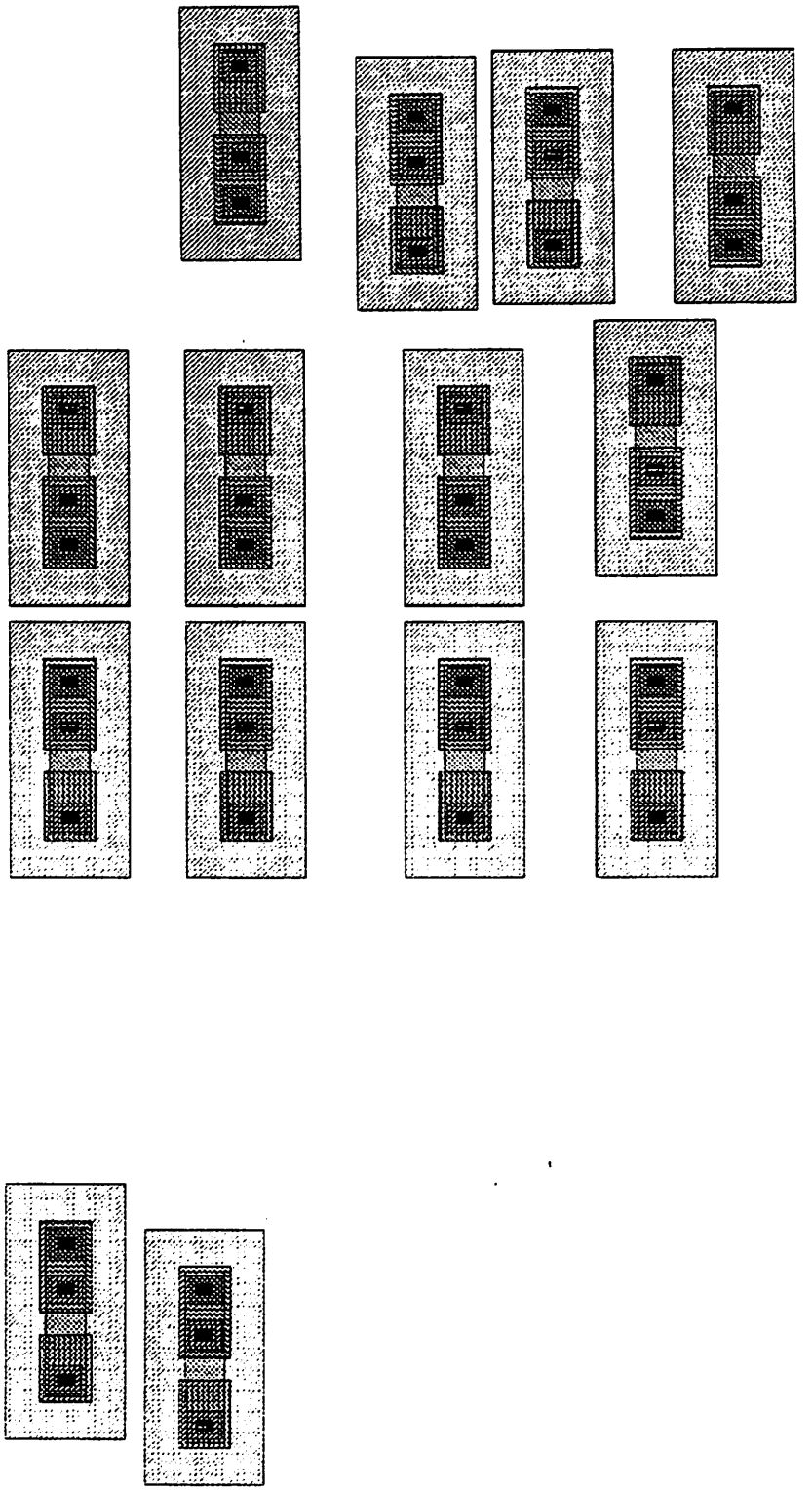


Figure 7.38: Transistors from Figure 7.37.

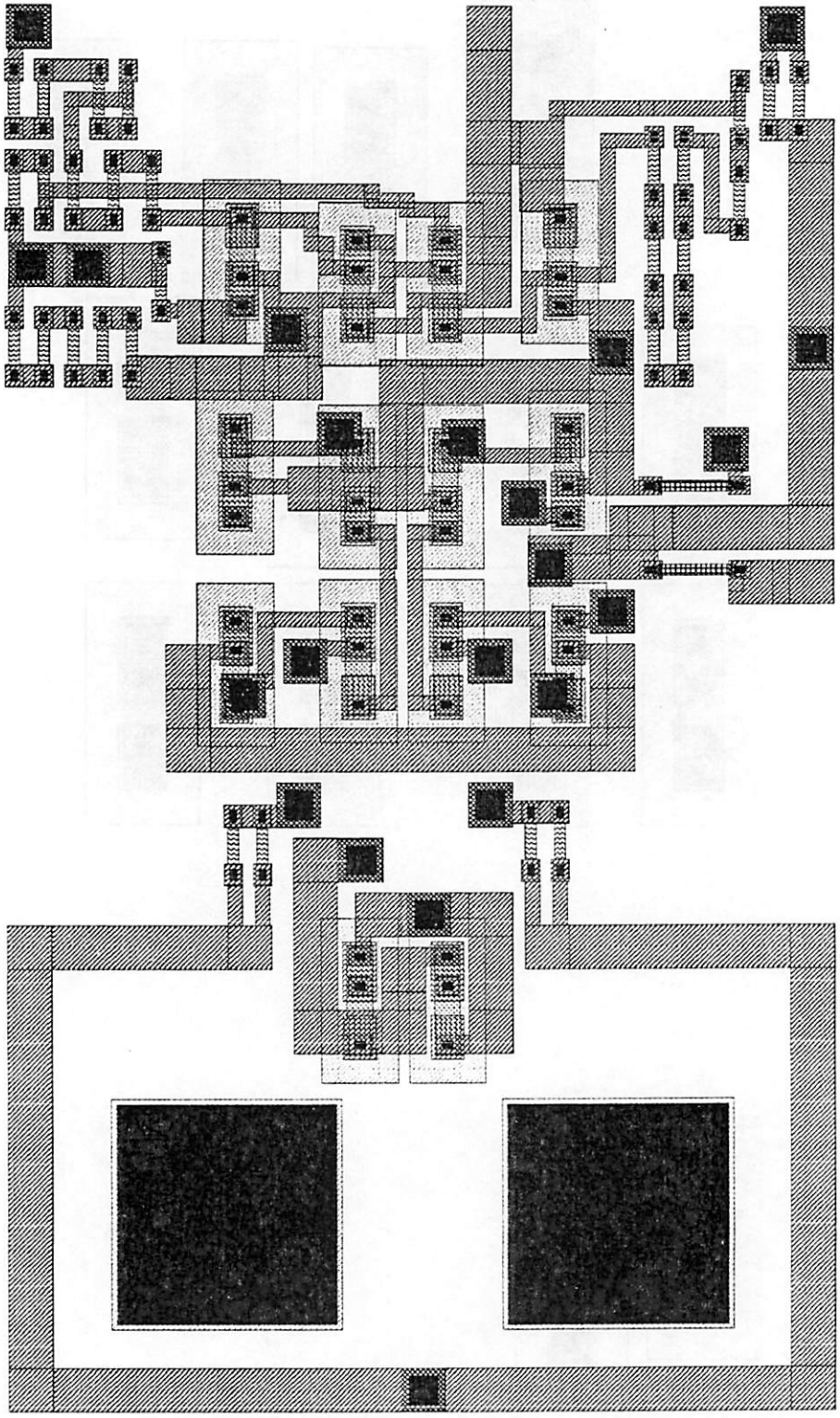


Figure 7.39: Amplifier compacted with active constraints.

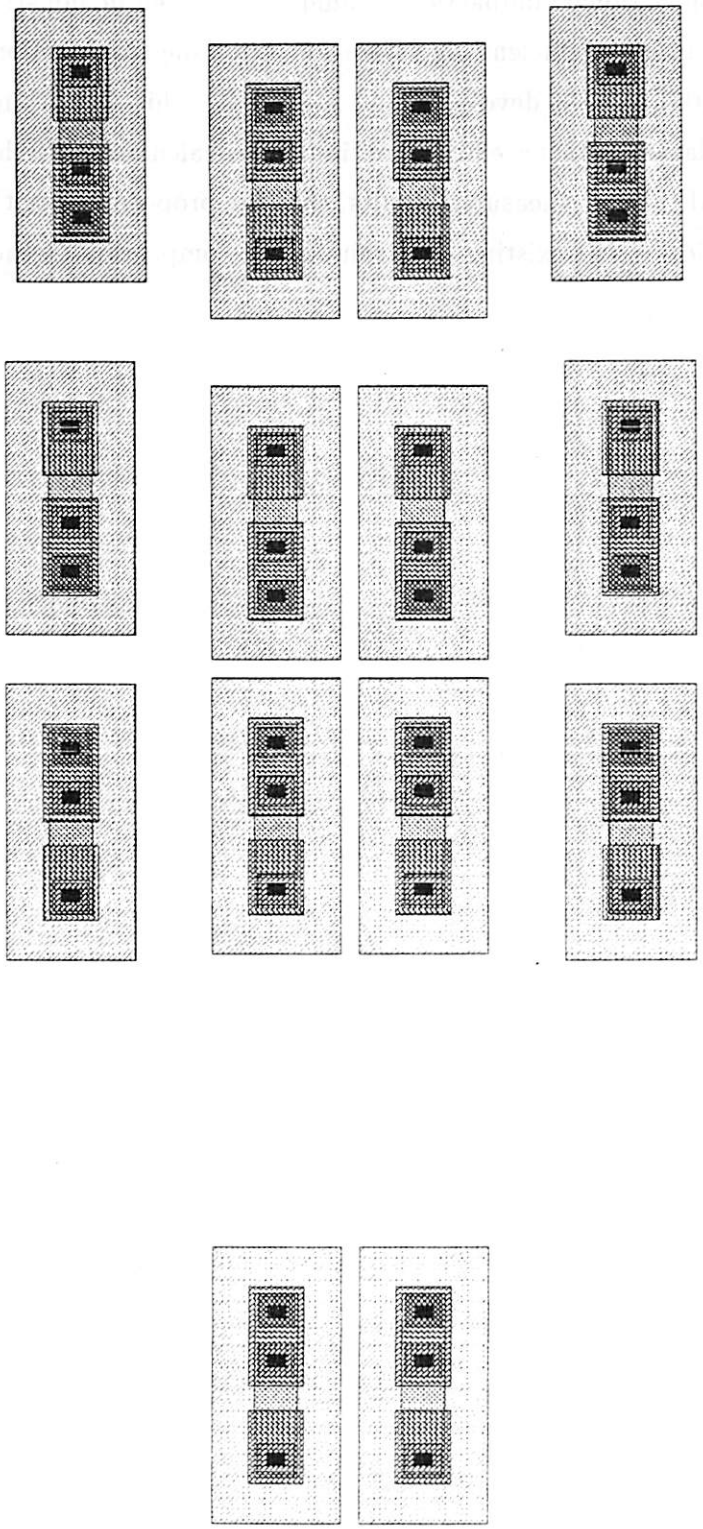


Figure 7.40: Transistors from Figure 7.39.

handled with conventional compaction techniques has been demonstrated.

In addition, an efficient algorithm for processing compaction graphs that include active constraints has been developed and presented. This algorithm combines a rigorous feasibility calculation with an efficient optimization calculation. It has been shown, both theoretically and through measured results, that the proposed algorithm is an efficient and practical extension to the existing constraint-based compaction methodology.

Chapter 8

SPARCS Implementation

8.1 Introduction

The SPARCS program [14,13] is a new layout compactor that has been written to test the algorithms described in this dissertation. The name "SPARCS" is actually an acronym for Spacing Program with ARbitrary ConstraintS. The architecture of SPARCS, and its relationship to other tools in the Berkeley CAD system, are the subject of this chapter.

The interfaces between SPARCS and the Berkeley CAD framework are described in the next section. The operation of SPARCS is then outlined. A description of the structure of the program and an indication of its size, in lines of code, concludes the chapter.

8.2 System Organization

The SPARCS program is part of a large IC CAD system under development at the University of California at Berkeley. The system is comprised of a number of application programs, such as SPARCS, integrated into a common framework. The major elements of the framework are the central data manager OCT, a graphical user-interface called *VEM*, and a facility for tool integration called *RPC* [19]. Symbolic layouts that are manipulated by SPARCS are stored in OCT, and displayed and edited via VEM.

There are two interfaces between the framework and SPARCS, and thus two variants of SPARCS that share the same core functions. The first interface allows the program to be invoked directly from the user's command shell: that is, it operates in a batch or

stand-alone manner. This interface is useful when user interaction is unnecessary or impossible, e.g., when SPARCS is called from another program as part of an automatic synthesis procedure. This variant of the program is linked directly with OCT's procedural interface, and each invokation maps to one CPU process. The second interface is used to invoke SPARCS from an interactive VEM session. When the distinction is important, this second variant of SPARCS will be referred to as *RPC-SPARCS* and the first as simply SPARCS. RPC-SPARCS communicates with OCT and VEM through RPC, which is a special-purpose remote-procedure-call package developed as part of the Berkeley framework. This mechanism enables RPC-SPARCS and OCT/VEM to run as separate, asynchronous processes; in fact, the two processes are not required to run on the same machine. The diagrams in Figures 8.1 and 8.2 show the relationships between OCT, VEM, SPARCS, and RPC-SPARCS.

8.3 SPARCS Operation

The basic operation of SPARCS is described in this section. The representation of symbolic layouts in OCT was presented in Chapter 3. A portion of that description is briefly summarized here to help illustrate the operation of SPARCS.

The highest-level object in OCT is the cell; each cell has one or more views. Each view has one or more facets, upon which the editing operations actually occur. According to the symbolic policy, a symbolic view has two facets, a contents facet, which is used to store the detailed definition of the view, and an interface facet, which is used to store its geometric abstraction. A facet is identified by specifying its cell name, view name, and facet name. For example, `inverter:unspaced:contents` refers to the contents facet of the unspaced view of the cell called inverter.

Assuming that SPARCS is called with `inverter:unspaced:contents` as the input facet to be compacted, SPARCS immediately copies the input facet into the output facet `inverter:spaced:contents`. The input facet is never actually read or modified, except at the user's request; this helps to avoid loss of the original data in the event of a program bug or system crash. The output facet is then opened, first to read the design data and later, when the spacing operation has completed, to update the data.

After `inverter:spaced:contents` is opened, SPARCS reads the input design data via the appropriate procedure calls provided by OCT. A number of consistency checks are

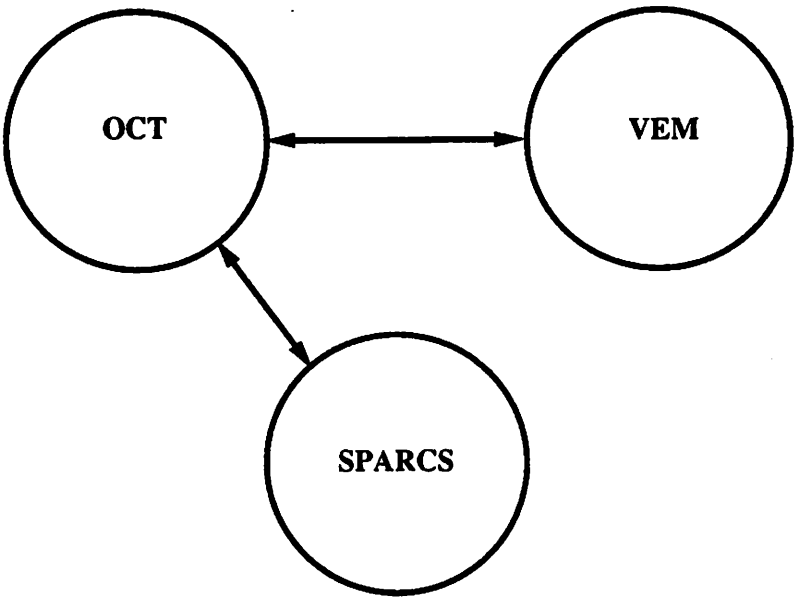


Figure 8.1: Relationship between SPARCS and the framework.

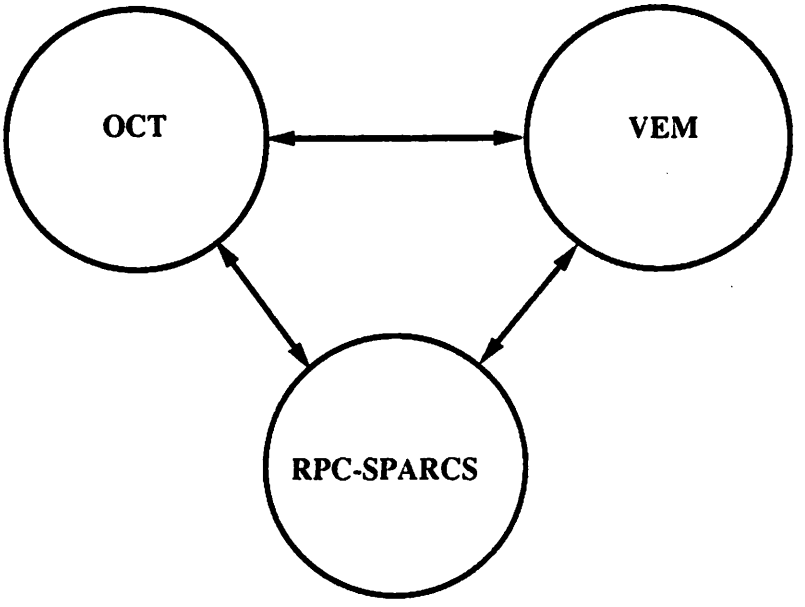


Figure 8.2: Relationship between RPC-SPARCS and the framework.

performed during the read operation. The input facet is read in three stages. First, the instances are read. For each instance, the interface facet of its master is opened and read to retrieve its protection frames and terminal frames. The wire segments are read in the second stage on a layer-by-layer basis. Each segment is checked as it is encountered to insure that the connectivity information is self-consistent. Any user-supplied compaction constraints are read in the third stage.

The first spacing iteration commences with the generation of the constraint graph. Each instance and each wire segment perpendicular to the spacing direction maps to a node in G. The PPS algorithm is executed to derive all of the design-rule constraints. The constraints necessary to force each wire segment to remain connected to the two terminal frames at its ends are added next. The source and sink nodes are then created and constraints are added from the source node to any nodes that have no fanins. Nodes that have no fanouts are similarly constrained to the sink node. The weights needed for slack distribution are installed in the graph after the source and sink are appended. Any user constraints that are present are then added to complete the generation of G.

Once it is constructed, the constraint graph is processed to determine new locations for the instances and segments. The instances translate in the direction of compaction. The perpendicular segments translate as well; the parallel segments translate and/or change length. The active constraints are processed first, as described in Chapter 7, if any are present for the current compaction direction. The critical-path and slack-distribution analyses are then performed using the algorithms presented in Chapter 6.

After the graph has been solved, the SPARCS data structures are updated with the new coordinates of the layout elements. This completes one compaction iteration. Compaction in the orthogonal direction is typically performed next, by repeating the constraint-generation, constraint-solving, and coordinate-updating steps. By default, SPARCS continues to iterate until the layout size becomes constant. SPARCS can optionally iterate until no layout elements move, or for a user-specified number of iterations.

The internal data structures of SPARCS contain the final coordinates of the elements in the compacted layout when the spacing iterations have completed. The output facet (`inverter:spaced:contents`) is then updated to reflect the final coordinates. SPARCS also stores some additional information in the output facet, namely the critical path from the final spacing iteration, the arguments with which SPARCS was invoked, and the convergence status.

At present, the design rules are stored in a text file. SPARCS runtime options can be set on the command line or through a control file.

8.3.1 Constraint Entry

Two stand-alone utility programs have been written to allow the user to add compaction constraints to OCT facets. The constraints are stored using a combination of OCT bag objects, OCT property objects, and the attachment operation. The *putConst* program adds a conventional constraint between two instances, or between an instance and a border of the layout. The second form enables the user to constrain instances to the source or sink of the graph; a typical use is to bind I/O terminals to the edges of the cell. The *putAConst* program adds an active constraint. The active constraint can be between two pairs of instances, or between two instances and a border. An example of the second form is the horizontal active constraint $x_a - x_s = x_b - x_s$, where a and b are two instances that are to be separated by equal amounts from the source node (left border).

8.3.2 Functions Specific to RPC-SPARCS

The SPARCS program can be used with or without a VEM graphics session. RPC-SPARCS must be used in tandem with a VEM session; it has some additional editing and display capabilities that are not possible in the the non-interactive variant.

RPC-SPARCS is invoked directly from any VEM window that contains the layout to be compacted. RPC-SPARCS registers several of its own menus with VEM. The first menu contains commands for starting a compaction run, killing a run, and exiting RPC-SPARCS. The second contains the same runtime controls as the stand-alone version. The third menu contains a number of commands for editing and displaying constraints graphically. The fourth menu contains commands for highlighting the elements on the critical paths of the layout.

8.4 SPARCS Structure

The structure of SPARCS follows its operation as described in Section 8.3. The major components common to both SPARCS and RPC-SPARCS are:

- a data-manager (OCT) reader.

- a constraint-graph generator,
- a constraint-graph solver,
- a data-manager writer.

RPC-SPARCS has an additional module that implements the commands and functions that use VEM graphics.

The OCT reader produces an an internal representation of the layout from the initial facet. The instances and segments are stored in separate linked lists. These geometric objects are further processed to generate a structure that makes the edge information used by the PPS algorithm readily accessible. For example, the protection-frame and terminal-frame edges of each instance are stored, with each instance, in separate lists, according to whether they are vertical or horizontal edges. Furthermore, additional data is stored on a per-instance basis to facilitate efficient construction of the MIN, MID, and MAX event queues of the PPS algorithm. Each user constraint is stored in one of four lists, as a function of compaction direction and whether it is a conventional constraint or an active constraint.

The constraint generator is comprised of an implementation of the PPS algorithm plus the additional functions described in Section 8.3. Implementations of the algorithms presented in Chapters 6 and 7 constitute the constraint solver.

The OCT writer scans the internal data structures of SPARCS, retrieves each corresponding OCT object in the output facet, and then updates its coordinate information. The extra critical-path and run information that SPARCS saves in the output facet is collected under a bag object named SPARCS-DATA.

8.4.1 Implementation

SPARCS is written in the C programming language [38]. To date it has run under several versions of the UNIX[1] operating system. Table 8.1 summarizes the size of SPARCS measured in lines of code. The entry labeled "OCT reader/writer" also builds much of SPARCS's internal data structure.

[1] UNIX is a trademark of AT&T.

Module/Function	Approx. lines of C
Data struct. definitions	700
OCT reader, writer	5400
Constraint generator	6950
Constraint solver	3350
Control routines	350
Updating routines	500
RPC-SPARCS additions	1900
Miscellaneous	1150
Total	20300

Table 8.1: Size of the SPARCS program implementation.

Chapter 9

Overall Results

9.1 Introduction

In the preceding chapters the major algorithms and models developed in this project have been described. As each particular algorithm has been presented, results on its performance have been presented as well.

The purpose of this chapter is to present some overall results for SPARCS that illustrate the entire compaction process. Initially, data is presented to illustrate the relative speeds of the various SPARCS algorithms. Then, a number of the applications for which SPARCS has been used are are enumerated. Two particular applications are described in more detail, namely a benchmark exercise from the 1987 Internation Conference on Computer Design, and compaction of macrocell layouts. Comments are then made on technology independence and hierarchy-level independence. The chapter concludes with a summary.

9.2 Applications of SPARCS

The utility of the methods described in this dissertation has been proven by applying SPARCS to many different IC layout-generation tasks. SPARCS has been used for a number of terms in the graduate-level introductory course in VLSI design at U.C. Berkeley, typically for manually-created hierarchical designs. The macrocell layout system Mosaico produces symbolic layouts only [12]; it uses SPARCS in two modes, which are both described later in this chapter. Several leaf-cell synthesis systems use SPARCS to produce their final

layouts. One is the GEM system [43], which produces random-logic cells. Another is the OPASYN program [40], which is a silicon compiler for CMOS operational amplifiers. SPARCS is also used by a technology-mapping system called KAHLUA [52]. This system first produces a symbolic layout from a physical layout. The symbolic layout is then mapped to a new technology by resizing the components and wiring, then performing a compaction using SPARCS.

9.3 Runtime Analysis

The data presented in this section provides an indication of the runtime performance of the major phases of SPARCS. The graph in Figure 9.1 is useful in comparing the constraint-generation time with the constraint-solution time. The eight examples used to produce the graph are those employed in Chapter 5 to measure the performance of the PPS algorithm. The examples range in size from 408 to 1881 elements, and the runtimes (in DEC VAX 8650 CPU-seconds) are for two spacing iterations, one per direction. This data indicates that SPARCS is well-balanced, in that the constraint generator and constraint solver consume about the same amount of CPU time. The constraint solver is slightly faster, but the constraint generator uses a less efficient data structure than necessary as its edge structure. A better edge-structure implementation would increase the speed of the constraint generator and further improve the balance between the two components.

The histogram presented in Figure 9.2 indicates the CPU-time distribution for a typical example. The data is for a two-iteration run of SPARCS.

9.4 ICCD Benchmark Session

SPARCS was one of four compaction programs invited to participate in a benchmarking session at the 1987 IEEE Internation Conference on Computer Design [13,8,9]. The other three programs were MACS, a one-dimensional constraint-based compactor [18], a virtual-grid compaction system from Symbolics [73], and Zorro, a zone-refining compactor [72]. The Symbolics compactor and MACS were developed in industrial laboratories, whereas SPARCS and Zorro were both developed at U.C. Berkeley.

All of the benchmark examples were created on a virtual-grid system, and a virtual-grid language was used to describe them. Unfortunately, this limited the experiment to

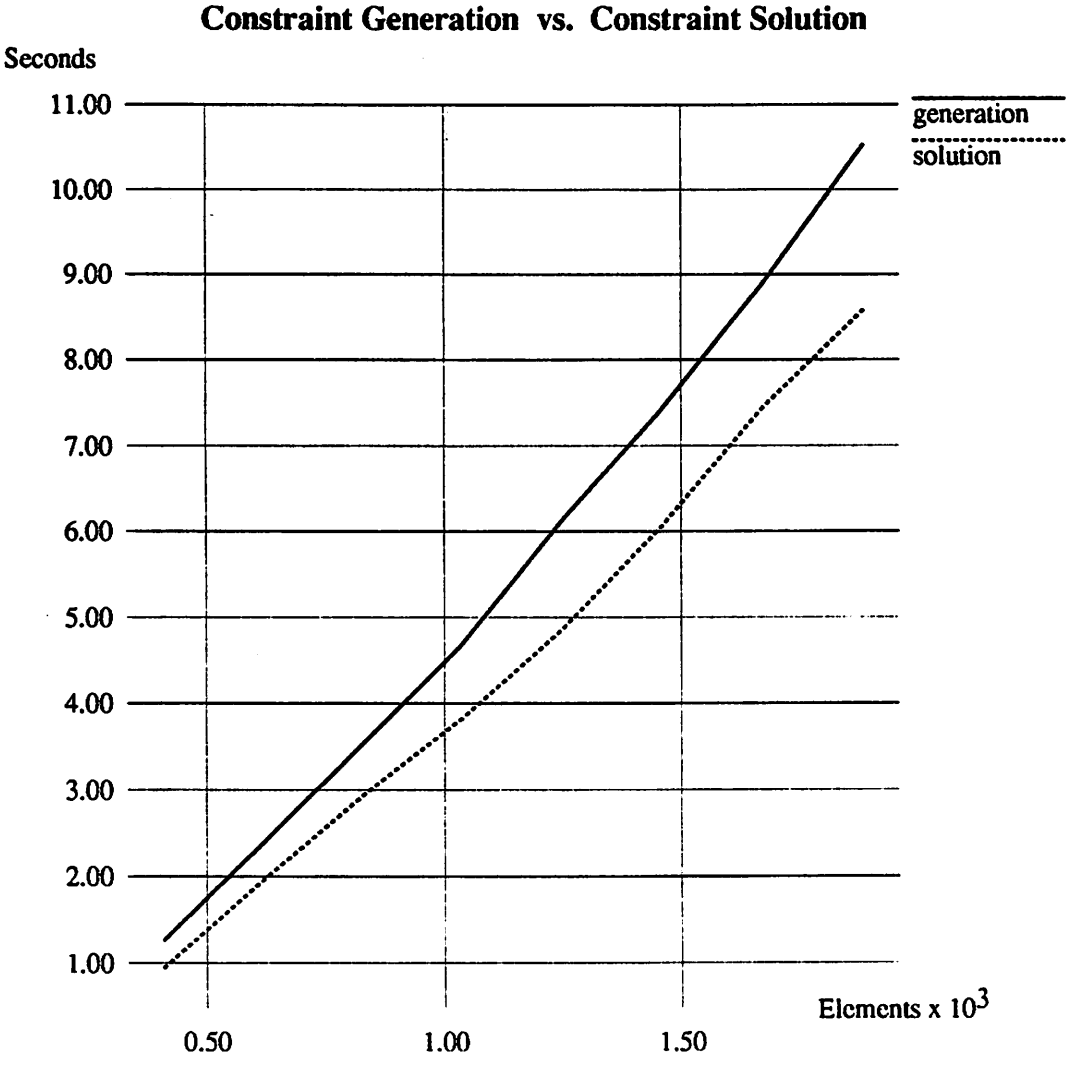


Figure 9.1: Constraint generation versus constraint solution.

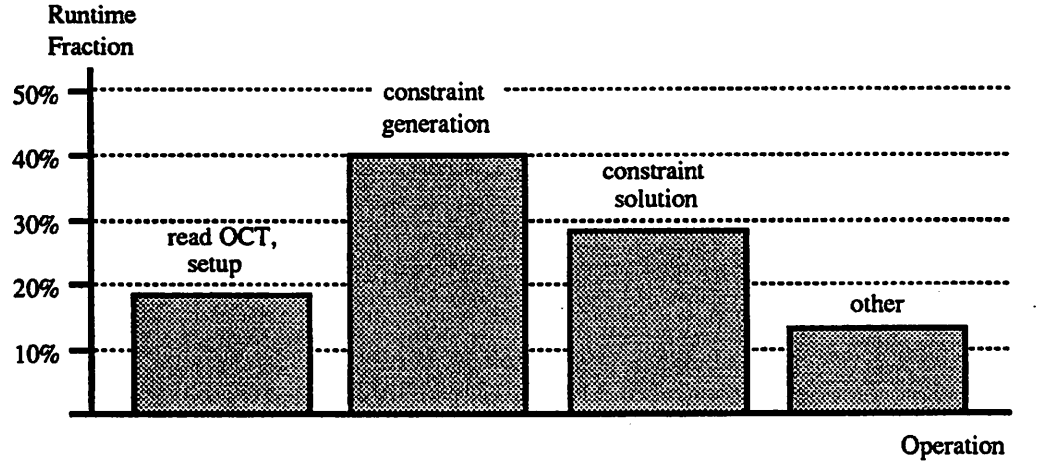


Figure 9.2: CPU-time distribution for a typical compaction run.

point-component, MOS-technology examples.

The benchmark results are reproduced in Table 9.1 [9]. Times are reported in VAX-8650 CPU-seconds, including the time required for well generation and conversion to CIF, which is a standard physical-layout language. All of the examples are CMOS layouts, except for **n28**, which is an NMOS layout. The **afa**, **afavg**, and **n28** examples are leaf cells. The **afavg** example was optimized manually for virtual-grid compaction by an experienced virtual-grid user [8]. The remainder are pitch-matching hierarchical examples; **mul2x2** is a multiplier circuit consisting of four cells, each of which is different. The **mul4x4**, **mul8x8**, and **mul16x16** examples are larger multipliers, all consisting of larger numbers of instances of the same four cells as **mul2x2**, plus instances of a fifth cell. The **c132** example is comprised of two rows of standard cells, each with six cells, with a wiring channel between the rows.

In terms of move-set generality, the four compactors can be ordered as follows. Zorro has the most general move set; is an intermediate compactor, meaning that its move set is more general than that of a one-dimensional compactor but less general than that of a two-dimensional compactor. MACS uses the one-dimensional constraint-based compaction model with wire-length minimization, plus jog generation. SPARCS differs from MACS in that SPARCS does not perform jog generation, and SPARCS uses a heuristic for slack distribution. The Symbolics Leaf Cell Compactor is an implementation of virtual-grid compaction, hence its move set is the least general. However, the Symbolics system

Example		
Compactor	Area (microns)	CPU (secs)
afa		
MACS	143 x 166 = 23738	9
SPARCS	157 x 180 = 28260	11
Symbolics	160 x 189 = 30240	5
Zorro	140.5 x 171 = 24025.5	430
afavg		
MACS	142 x 145 = 20590	5
SPARCS	157 x 151 = 23707	8
Symbolics	154 x 154 = 23716	5
Zorro	128.5 x 151 = 19403.5	524
n28		
SPARCS	123 x 198 = 24354	8
Zorro	108 x 187 = 20196	378
c132		
MACS	627 x 354 = 221958	41
SPARCS	685 x 339 = 232215	51
Symbolics	675 x 330 = 222750	57
Zorro	660 x 322 = 212520	301
mul2x2		
MACS	309 x 252 = 77868	16
SPARCS	343 x 255 = 87465	47
Symbolics	370 x 270 = 99900	10
Zorro	312 x 252 = 78624	838
mul4x4		
SPARCS	649 x 601 = 390049	66
Symbolics	654 x 638 = 417252	54
Zorro	577 x 577.5 = 333217.5	1904
mul8x8		
SPARCS	1285 x 1285 = 1651225	89
Symbolics	1276 x 1352 = 1725152	245
Zorro	1138 x 1207.5 = 1374135	11738
mul16x16		
Symbolics	2524 x 2780 = 7016720	1073

Table 9.1: ICCD benchmark results [9].

consists of several additional programs. One of these is a preprocessor called the Symbolic Compactor that attempts to improve symbolic layouts, prior to compaction, by changing components, moving components to alter the topology, changing wiring layers, etc.

As expected, the area results for the most part follow the above ordering. SPARCS generates denser layouts than the Symbolics system on the leaf cells, including **afavg** (the example optimized for virtual-grid compaction), and on all the other examples except for **c132**. It appears from the plot in [9] of the Symbolics **c132** result that the Symbolic Compactor was run on the routing area of this example. This effectively creates a different initial topology for the wiring region, which includes jogs and wrong-way wiring. These topological changes have a large influence on the final area, which accounts for the better result of the Symbolics system in this case.

The MACS compactor produced better results in terms of area than SPARCS on the leaf-cell examples. The jog-generation capability of MACS plays a large role in this difference; there is also an argument that wire-length-minimization leads to higher densities than other slack-distribution methods [24]. In addition, the two programs do not model contacts identically. Close inspection of the plots ([9]) indicates that MACS used a more aggressive contact representation than SPARCS; in fact, of the four programs, SPARCS used the most conservative contact model. The Zorro program usually produced the densest layouts; however, it is interesting to note that one-dimensional compaction with jog generation (MACS) was better than this intermediate compaction method in two of the cases.

Overall, the runtime efficiency of one-dimensional compaction relative to the intermediate approach is apparent from the data in Table 9.1. The **afa** and **afavg** examples can be used to compare the runtime performance of the three one-dimensional compactors for leaf-cell compaction.

Using only the data in Table 9.1, it is clear that the runtime performance of SPARCS is competitive with the two industrial programs. However, not all of the relevant information appears in Table 9.1. In particular, the number of spacing iterations is not indicated. At the time the benchmarks were submitted, SPARCS iterated until no elements in the layout moved. This led to five and three iterations for the **afa** and **afavg** cases, respectively. The Symbolics Leaf Cell Compactor performs two iterations. The MACS paper ([18]) implies, but does not state, that MACS performs two iterations as well.

For the **afavg** example, SPARCS actually produced the reported area in two iterations. For the **afa** example, the area of the layout after two iterations is within 0.9% of the

reported area. Hence SPARCS could have been stopped after two iterations in both cases without significantly changing its area results.

The two programs used to post-process the SPARCS output (one to generate the wells and one to transform the OCT data into CIF) were simple programs that were not optimized for efficiency. As a result, these procedures accounted for about 2.5 seconds of the reported runtimes for both examples. That is, the actual compaction times are about 8.5 seconds for **afa**, and about 5.5 seconds for **afavg**, for five and three iterations, respectively. If two iterations had been performed in each case, and if better programs for the post-processing steps had been available, SPARCS would have produced layouts of equivalent area in significantly less time than the data that appears in Table 9.1.

All of the the hierarchical examples (**mul2x2**, **mul4x4**, **mul8x8**, **mul16x16**, **c132**) are described using two levels of hierarchy, and are designed for a pitch-matching methodology. However, a decomposition using more than two levels of hierarchy is useful for the larger multipliers. The following example, using **mul4x4**, illustrates how SPARCS can be used to exploit this structural regularity.

The **mul4x4** example consists of a 3x2 array of the **afa** cell, surrounded by instances of **fal3**, **aaha1**, **nand1**, **nand3**, and **celllc**, as shown in Figure 9.3. Initially each leaf cell was compacted until no elements moved. Table 9.2 summarizes this processing (**celllc** consists of only four wires, and hence was not processed in this step). To pitch-match the leaf cells for use in assembling **mul4x4**, the pitch constraints required to align their ports were added manually, then the cells were respaced and their interface facets were created. The time required for respacing and for creating the interface facets via the vulcan program, and the resulting cell sizes, are given in Table 9.3. The multiplier has three unique columns as shown in Figure 9.3. An intermediate hierarchy level was created by constructing these three unique columns. For example, the pitch-matched versions of **aaha1** and **nand3** were loosely placed and routed together to create the fourth column.[1] The fourth column, composed of four pitch-matched leaf cells, was then compacted and its interface facet was created. This procedure was also used for the first and second columns. The data representing the column processing is summarized in Table 9.4. Finally, **mul4x4** was assembled by placing and routing an instance of the first column, two instances of the

[1]The place-and-route steps were performed manually due to lack of a suitable tool. Since these steps require only simple arraying and river-routing algorithms, they could be easily automated, and the time required to execute them would be insignificant.

second column, and an instance of the fourth column. That is, the top-level representation had four instances only, rather than the sixteen that would be present if a two-level hierarchy was used instead. This representation was compacted to generate the final layout. The **mul4x4** example was also compacted flat and the results of the two approaches are given in Tables 9.5 and 9.6. The flat case takes 2.73 times longer and is 5.1% smaller than the hierarchical case. The area figures are in λ and include the well area, and the times, in VAX-8650 CPU-seconds, include all processing except for CIF-file generation. A plot of the hierarchical result is presented as Figure 9.4.

celllc	**nand1**	**nand1**	**nand3**
fal3	**afa**	**afa**	**aaha1**
fal3	**afa**	**afa**	**aaha1**
fal3	**afa**	**afa**	**aaha1**

Figure 9.3: Cell arrangement of **mul4x4**.

Cell	Iter.	Time	Width	Height	Area
aaha1	3	3.77	94	100	9400
afa	5	8.75	105	120	12600
fal3	5	9.17	128	112	14336
nand1	3	1.22	25	58	1450
nand3	3	1.16	22	53	1166
TOTAL		24.07			

Table 9.2: Initial leaf-level compaction, no constraints.

Comparing SPARCS with the Symbolics system on the hierarchical examples, SPARCS is slower on the small examples, but its rate of CPU-time growth with problem size is smaller, hence SPARCS becomes faster once the problem size reaches a certain point. The Symbolics system is significantly faster on **mul2x2**. On **mul4x4**, SPARCS is less than 20% slower, and on **mul8x8**, which is roughly four times the size of **mul4x4**, SPARCS is significantly faster. (SPARCS was executed on all the examples except for the largest multiplier,

Cell	Iter.	Time	Frames	Total	Width	Height
aaha1	2	3.19	1.80	4.99	96	123
afa	2	5.14	2.17	7.31	105	123
fal3	2	5.07	2.37	7.44	129	123
nand1	2	1.11	0.97	2.08	105	58
nand3	2	1.08	0.95	2.03	96	58
celllc	2	0.45	0.61	1.06	129	58
TOTAL				24.91		

Table 9.3: Leaf-cell respacing, with pitch constraints.

Cell	Iter.	Time	Frames	Total
col. 1	1	2.08	1.42	3.50
col. 2	1	2.11	1.90	4.01
col. 4	1	1.72	1.59	3.31
TOTAL				10.82

Table 9.4: Column processing.

	Hierarchical		Flat	
Cell	Iter.	Time	Iter.	Time
mul4x4	1	4.31	5	175.09
leaf cells	—	48.98	—	—
columns	—	10.82	—	—
TOTAL		64.11		175.09

Table 9.5: Total time, hierarchical versus flat.

Method	Width	Height	Area
hier.	416	401	166816
flat	405	392	158760

Table 9.6: Area comparison, hierarchical versus flat.

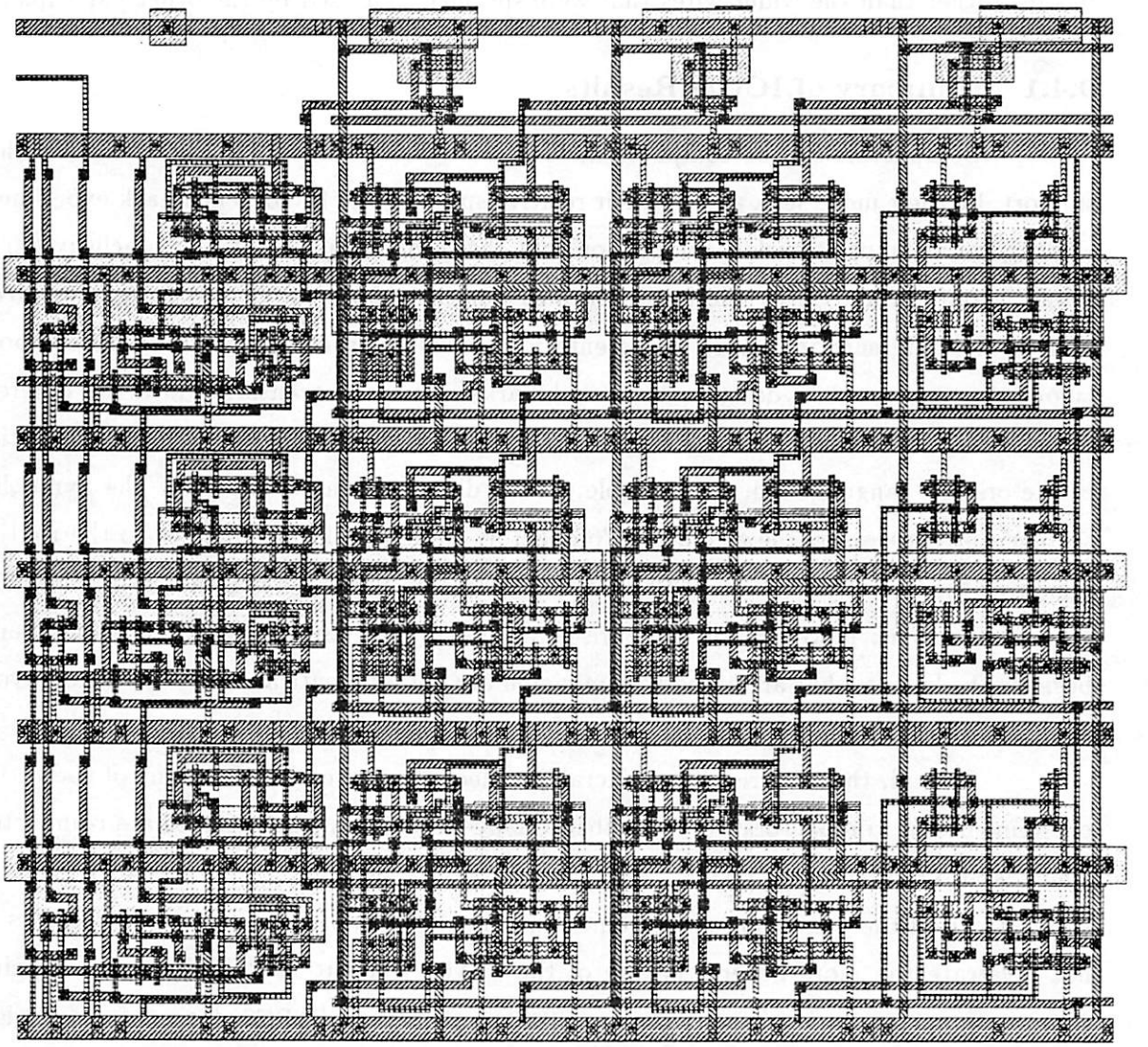


Figure 9.4: **mul4x4** result via hierarchical compaction.

which was not completed due to lack of time.)

Comparison with MACS on the hierarchical examples is not meaningful. The MACS team did not submit results for any multiplier except the one with no repeated cells (**mul2x2**). In the case of **c132**, the MACS layout uses minimum-width wiring for the power busses, rather than the wider wires that were specified and used by the other participants.

9.4.1 Summary of ICCD Results

Compactors can be compared in several ways, such as by the layout models they support, by their move sets, and by their relative speeds. The ICCD benchmark experiment was a good attempt at such a comparison, but the results are not entirely conclusive. The initial layouts processed by the four programs were not identical. Some of the differences, such as the difference in contact representations, resulted from the fact that the symbolic layout language used to describe the benchmarks had to be translated into the different formats that each tool uses as its input representation. Slight differences in interpretation of the original language led, for example, to the different contact models. The Symbolics system's pre-processor, the Symbolic Compactor, made significant changes to the routing region of the initial layout of **c132**. Such changes are perfectly reasonable in a production environment; however, for comparison purposes they partially invalidate the results because the layouts that are actually compacted are not all identical. Also, the **c132** layout compacted by MACS used different bus widths than those that were specified.

Overall, the area results predictably reflect the move sets. In terms of speed, the one-dimensional compactors were all substantially faster than the intermediate compactor, which is also a predictable result.

All of the benchmarks are point-component layouts. Point-component layouts do not illustrate the more general nature of the SPARCS layout model. In a sense, point-component layouts are a worst-case for SPARCS, because SPARCS does not, by design, exploit element types whereas other compactors can. Nevertheless, two important conclusions can be drawn from the benchmark results. SPARCS produced layout areas that are consistent with its move set. This implies that *the proposed generic layout model does not degrade layout density* over well-implemented programs that use the more restrictive, typed layout model. Also, the SPARCS runtimes are very competitive compared to the industrial programs. This implies that *the added generality of the generic model does not incur a run-*

time cost, due to the design of the model and due to the proposed PPS constraint-generation algorithm. It should be noted that the two industrial programs are strong competitors; the Symbolics system is a third-generation virtual-grid system, and the MACS program is a sophisticated implementation of constraint-based compaction.

9.5 Macrocell Compaction

It was mentioned above that SPARCS is used in the Mosaico macrocell layout system. At the phase in the design cycle where compaction is applied, macrocell layouts are comprised of large, rectilinear cells with many terminals. The cells are not stretchable. The cells are wired together using a variety of styles, including channel routing and switchbox routing. As a result, pitch-matching hierarchical compaction cannot be applied to macrocell designs.

The Mosaico system produces symbolic layouts only; hence all Mosaico layouts must be compacted. These layouts are stored in OCT using the generic layout model. Mosiaco uses SPARCS in two modes, one in which SPARCS compacts the entire top level of the design flat, and one in which compaction is performed on each routing region individually [12].

The flat case is the more difficult of the two, because the layouts are much larger. Tables 9.7, 9.8, and 9.9 contain the results from [12]; the three examples are from industrial sources. These examples were compacted flat. This data indicates that SPARCS produces significant improvements in area for the automatically-placed examples, **ck1** and **ck3**. The **ck2** example was manually placed. In addition, the SPARCS runtimes are quite reasonable compared to those of the other tools in the system.

Circuit	Macrocells	Pads	Nets	Pins	Channels
ck1	8	5	29	58	17
ck2	12	39	262	691	54
ck3	23	17	129	458	43

Table 9.7: Test circuit statistics.

9.6 Technology Independence

SPARCS has been successfully used on layouts from a number of IC technologies.

Circuit	Placed	Routed	Compacted	Area Savings (%) After Compaction
ck1	1592x1415	1678x1562	1533x1433	16
ck2	17305x15620	17033x14982	15670x14982	1
ck3	2902x3607	4245x4904	4036x4112	20

Table 9.8: Areas (λ) during the design cycle.

Circuit	Placement	Routing		Compaction
		Global	Detailed	
ck1	140	5	22	21
ck2	n.a.	926	482	960
ck3	1203	74	140	480

Table 9.9: Runtimes, VAX 8650 CPU seconds.

The ICCD results presented in this chapter include both NMOS and CMOS layouts. The CMOS technology used is a modified version of the Mosis CMOS technology, which was altered to complicate the design rules. SPARCS has been used to compact layouts from several other CMOS technologies as well. In Chapter 7, results were presented for a bipolar layout from an industrial fabrication process.

In all cases, no modifications to SPARCS were required when the technology was changed. That is, no new element types or algorithms were necessary. All that is needed to adapt SPARCS to a new technology is to modify its rules file to reflect the layer names and spacing values of the new technology. These results prove that the algorithms described in this dissertation are indeed technology independent.

9.7 Hierarchy-Level Independence

It has been stated that the algorithms and models used in SPARCS are hierarchy-level independent, unlike those used in other systems. The results presented in this chapter support this statement.

The **mul4x4** example from the ICCD benchmarks shows that SPARCS can operate in the pitch-matching mode used by other compactors. Other hierarchical styles require non-rectangular subcells, and/or are not amenable to pitch-matching. SPARCS has processed a large number of hierarchical designs in these other styles. One particular type of hierarchical compaction for which SPARCS has been used extensively is the channel-by-channel style

employed in Mosaico [12]. These problems include large numbers of non-rectangular cells, and pitch-matching cannot be performed because the macrocells are of fixed size when the compaction phase has been reached. The standard pitch-matching hierarchical method simply cannot accommodate such designs.

9.8 Summary

The overall goal of this research has been to create practical symbolic layout and compaction methods that apply to a wider variety of designs than the existing methods. The results presented in this and preceding chapters show that this goal has been met.

SPARCS, the program that implements the methods that have been proposed, has been shown to be technology independent and hierarchy-level independent. It has also been shown that SPARCS is able to satisfy the special requirements of symmetry and hierarchy preservation through the use of active constraints. The practical nature of the proposed methods has been demonstrated by the fact that SPARCS has been used to compact a very large number of designs from a wide range of design methodologies.

Finally, the ICCD benchmark results indicate that the proposed methods do not sacrifice runtime or area efficiency compared to high-quality implementations of the existing, less-general symbolic layout and compaction techniques.

Chapter 10

Conclusions and Future Work

Various forms of symbolic layout and layout compaction have existed for more than a decade. These techniques have been employed in some settings, but, due to a number of limitations, their use has not been widespread. The overall goal of this research has been to investigate and develop more general models and algorithms for symbolic layout and layout compaction, thereby widening the domain of problems suited to such a methodology. This goal has been reached, through the three main contributions summarized in the following section.

10.1 Contributions

10.1.1 A General Layout Model

Any system is only able to process the inputs that can be described within the scope of its input model. Symbolic layout and compaction systems typically use a layout model (input model) that can effective capture only a particular class of leaf-level MOS-technology designs, because the layout model is defined in terms of a few simple, typed primitives. Systems that use this model have been shown to produce area-efficient results for the class of layouts that the model is intended for. However, the typed nature of the model renders such systems cumbersome or unusable for many other classes of IC layouts.

A new layout model has been developed to address this problem. Its primary characteristic is that it is generic rather than typed. In addition, the level of abstraction of the proposed model is lower than that of the standard model. The proposed model is

therefore more general, since it is not restricted to pre-encoded types of elements, nor to elements with specific geometric configurations.

The results presented in this dissertation show that the proposed model effectively covers a much wider domain of layout problems than typed models. In particular, it is technology and hierarchy-level independent. It has also been shown that the generic model is just as efficient as typed models on the class of layouts that typed models are designed for.

10.1.2 Efficient Constraint Generation

The CPU time consumed by a one-dimensional compactor during constraint generation is an important, if not dominant fraction of the overall runtime. The difficulty of the constraint-generation problem increases when a generic layout model is used instead of a typed model, because the constraints must be derived from lower-level information. That is, the element types cannot be used as hints to aid the constraint-generation process, since they are not present.

A new algorithm for constraint generation has been developed along with the generic layout model, such that the model and the algorithm complement one another. The layout model has been designed to ease the constraint-generation process wherever possible. The constraint-generation algorithm has been designed to exploit the ability of the model to capture geometric detail, without consuming an excessive amount of CPU time.

The proposed constraint-generation algorithm scans the layout in the direction perpendicular to the compaction direction, unlike the standard methods, which scan in the parallel direction. The primary advantage of perpendicular scanning is that nearly all of the pruning operations used in the scan can be performed without the need for explicit data structures. As a result higher efficiency is achieved, because the cost of maintaining the data structures as the algorithm executes is reduced. This characteristic is of particular importance for low-level models like the proposed generic model.

The previous work in perpendicular scanning is mostly theoretical in nature and does not address many of the requirements of a practical constraint-generation algorithm. The work described in this dissertation has addressed these requirements, for a very general class of layout problems. Unlike previous perpendicular-scanning algorithms, the proposed algorithm accommodates multi-layer, non-rectangular geometries and non-transitive spacing

rules. In addition, it supports terminal merging and the generation of corner constraints. If these features are missing, the utility of the algorithm is severely reduced. The results which have been presented herein show that the new algorithm performs well, in terms of both CPU time and layout area.

10.1.3 Active Constraints

Conventional, two-variable compaction constraints are unable to model a number of practical layout problems. For example, symmetry relationships, and the relationships necessary to preserve hierarchy when multiple instances of the same cell are being pitch-matched, cannot be represented by two-variable constraints. These relationships are particularly important in analog layouts, which was the primary motivation for this work. The addition of a four-variable constraint, termed an active constraint, has been proposed to model relationships such as these.

It has been shown in this dissertation that a mixed-constraint system, comprised of two and four-variable constraints, cannot be efficiently solved with existing algorithms. A new algorithm has thus been developed to solve this problem. The proposed algorithm combines a robust feasibility analysis with an optimization phase to compute the value for each four-variable constraint. The proposed methodology has been shown to be efficient both in theory and in practice. The usefulness of active constraints, and the algorithm for solving the resulting mixed-constraint systems, has been demonstrated via realistic examples. In particular, this approach can be used to maintain the required relationships among components that arise in the output of tools such as analog-oriented routers, under layout modifications or changes in layout rules.

10.2 Future Work

The compaction program SPARCS (Chapter 8) was written not only to test the models and algorithms described in this dissertation, but to be a useful compactor as well. The main limitations of SPARCS at present are its lack of an optimal slack-distribution routine and its lack of a jog-generation capability. Neither of these capabilities was addressed in this research, but both should be added to the program.

The constraint-generation (PPS) algorithm and the active-constraint algorithm are both amenable to speed improvements. For example, the PPS implementation uses a linear

list for the active edges, which unnecessarily increases the edge-insertion time. The active-constraint solver performs the `lp(`s`)` analysis a number of times. Each time, the entire solution is re-computed; much of this re-computation can be avoided through the use of the previous solution and the event-driven paradigm.

Compactors that use the conventional constraint model are in use today, even though the existing constraint model is sometimes inadequate. Constraints of the form $x_b \geq x_a \pm k$ are *continuous* constraints in that x_b and x_a can assume any values that satisfy the equation. In reality, some of the values that satisfy equations of this form are illegal. For example, two contacts that are electrically connected are allowed to be either coincident (fully merged), or separated by at least one spacing rule; intermediate separations are illegal. Intermediate values are not prohibited by the existing constraint model. Situations such as this require a *discrete* relationship between x_a and x_b that prohibits the illegal values. Some technology rules, such as reflection rules, also map to non-continuous constraints. Ideally, there should be two forms of two-variable constraints, a continuous form and a discrete form. Problems with this structure can be formulated as mixed-integer-linear-programming (MILP) problems. Unfortunately, MILP optimization problems are NP-complete problems [61] and thus very difficult to solve. At present, it appears that only the longest-path problem has been addressed in the MILP context [49]. The slack-distribution and positive-cycle problems have yet to be considered. Also, constraint-generation algorithms that synthesize the discrete constraints have not been developed.

One-dimensional compaction was selected for this project for the reasons presented in Chapter 2. However, there are situations in which the added complexity of a more general, two-dimensional move set is justified. To date, only the extremes of two-dimensional compaction have been studied. Very general methods have been addressed, but they are limited to very small problems. At the opposite end of the spectrum, several methods that are heavily based on one-dimensional compaction have been proposed as well. The middle ground between these extremes has not been explored. Relatively structured designs which must be of high quality, for instance the library cells of a standard-cell system, may be amenable to such an approach.

Bibliography

[1] Microelectronics Center of North Carolina. *MCNC International Workshop on Symbolic Layout and Compaction*, November 1986.

[2] *Mosis CMOS Scalable Rules/Mosis NMOS Lambda-Based Design Rules.* Information Sciences Institute, University of Southern California. (Mosis is an IC fabrication service for universities.).

[3] Brian Ackland and Neil Weste. An automatic assembly tool for virtual grid symbolic layout. In *VLSI 83 International Conference*, 1983.

[4] Sheldon B. Akers, James M. Geyer, and Donald L. Roberts. Ic mask layout with a single conductor layer. In *Proceedings of the 7th Design Automation Workshop*, pages 7–16, SHARE/ACM/IEEE, San Francisco, CA, 1970 1970.

[5] Mark W. Bales. *Layout Rule Spacing of Symbolic Integrated Circuit Artwork.* Technical Report UCB/ERL M82/72, UC Berkeley Electronics Research Laboratory, May 1982.

[6] Donald T. Barnes and Thomas L. Davis. Graphics layout systems for faster tooling from faster designs. In *1975 Wescon Professional Program*, Western Electronic Show and Convention, San Francisco, CA, September 1975.

[7] David G. Boyer. Split grid compaction for a virtual grid symbolic design system. In *Proceedings of the International Conference on Computer-Aided Design*, pages 134–137, IEEE, November 1987.

[8] David G. Boyer. Symbolic layout compaction benchmarks - introduction and ground ruless. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers & Processors*, pages 186–191. IEEE, Rye Brook, NY, October 1987.

[9] David G. Boyer. Symbolic layout compaction benchmarks - results. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers & Processors*, pages 209–217, IEEE, Rye Brook, NY, October 1987.

[10] David G. Boyer. *Virtual Grid Compaction using the Most Recent Layers Algorithm.* Master's thesis, Department of Computer Studies, North Carolina State University, Raleigh, NC, 1983.

[11] David G. Boyer and Neil Weste. Virtual grid compaction using the most recent layers algorithm. In *Proceedings of the International Conference on Computer-Aided Design*, pages 92–93, IEEE, November 1983.

[12] J. Burns, A. Casotto, M. Igusa, F. Marron, F. Romeo, A. Sangiovanni-Vencentelli, C. Sechen, H. Shin, G. Srinath, and H. Yaghutiel. Mosaico: an integrated macro-cell layout system. In *Proceedings of the IFIP TC10/WG 10.5 International Conference on Very Large Scale Integration*, pages 165–179, Vancouver, Canada, August 1987.

[13] Jeffrey L. Burns and A. Richard Newton. Efficient constraint generation for hierarchical compaction. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers & Processors*, pages 197–200, IEEE, Rye Brook, NY, October 1987.

[14] Jeffrey L. Burns and A. Richard Newton. Sparcs: a new constraint-based ic symbolic layout spacer. In *Proceedings of the IEEE 1986 Custom Integrated Circuits Conference*, pages 534–539, IEEE, Rochester, NY, May 1986.

[15] Jeffrey L. Burns and Rick L. Spickelmier. Oct symbolic view specification. In Rick L. Spickelmier, editor, *OCT Tools Distribution 3.0.* Electronics Research Laboratory, University of California, Berkeley, 1989.

[16] Y. E. Cho, A. J. Korenjak, and D. E. Stockton. Floss: an approach to automated layout for high-volume designs. In *Proceedings of the 14th Design Automation Conference*, pages 138–141, ACM/IEEE, New Orleans, LA, June 1977.

[17] Y. Eric Cho. A subjective review of compaction. In *Proceedings of the 22nd Design Automation Conference*, pages 396–404, ACM/IEEE, Las Vegas, NV, June 1985.

[18] W. H. Crocker, Ravi Varadarajan, and Chi-Yuan Lo. Macs: a module assembly and compaction system. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers & Processors*, pages 205–208, IEEE, Rye Brook, NY, October 1987.

[19] Rick L. Spickelmier David S. Harrison, Peter Moore and A. Richard Newton. Data management and graphics editing in the berkeley design environment. In *Proceedings of the International Conference on Computer-Aided Design*, pages 24–27, IEEE, November 1986.

[20] James Do and William M. Dawson. Spacer II: a well-behaved ic layout compactor. In *VLSI 85 International Conference*, pages 277–285, Tokyo, August 1985.

[21] Jurgen Doenhardt and Thomas Lengauer. Algorithmic aspects of one-dimensional layout compaction. *IEEE Transactions on CAD*, CAD-6(5):863–878, September 1987.

[22] Alfred E. Dunlop. *Integrated Circuit Mask Compaction.* PhD thesis, Department of Engineering, Carnegie-Mellon University, October 1979.

[23] Alfred E. Dunlop. Slip: symbolic layout of integrated circuits with compaction. *Computer-Aided Design*, 10(6):387–391, November 1978.

[24] Peter A. Eichenberger. *Fast Symbolic Layout Translation for Custom VLSI Integrated Circuits.* Technical Report 86-295, Computer Systems Laboratory, Stanford University, April 1986.

[25] David Gibson and Scott Nance. Slic — symbolic layout of integrated circuits. In *Proceedings of the 13th Design Automation Conference*, pages 434–440, ACM/IEEE, San Francisco, CA, June 1976.

[26] David Gibson and Scott Nance. Symbolic system for circuit layout and checking. In *1977 IEEE International Symposium on Circuits and Systems Proceedings*, pages 436–440, Phoenix, AZ, April 1977.

[27] Paul R. Gray and Robert G. Meyer. *Analysis and Design of Analog Integrated Circuits.* Wiley, 1977.

[28] Douglas J. Hamilton and William G. Howard. *Basic Integrated Circuit Engineering.* McGraw-Hill, 1975.

[29] Thomas Hedges, William Dawson, and Y. Eric Cho. Bitmap graph build algorithm for compaction. In *Proceedings of the International Conference on Computer-Aided Design*, pages 340–342, IEEE, November 1985.

[30] Dwight D. Hill, John P. Fishburn, and Mary D. P. Leland. Effective use of virtual grid compaction in macro-module generators. In *Proceedings of the Design Automation Conference*, pages 777–780, June 1985.

[31] Min-Yu Hsueh. *Symbolic Layout and Compaction of Integrated Circuits.* Technical Report UCB/ERL M79/80, UC Berkeley Electronics Research Laboratory, December 1979.

[32] Min-Yu Hsueh and Donald O. Pederson. Computer-aided layout of lsi circuit building-blocks. In *Proc. 1979 IEEE International Symposium on Circuits and Systems*, pages 474–477, 1979.

[33] Masaki Ishikawa, Tsuneo Matsuda, Takeshi Yoshimura, and Satoshi Goto. Compaction-based custom lsi layout design method. *IEEE Transactions on CAD*, CAD-6(3):374–382, May 1987.

[34] D. B. Johnson. Priority queues with update and finding minimum spanning trees. *Information Processing Letters*, 53–57, 1975.

[35] Fumio Kato and Hiroshi Shiraishi. Efficient compaction technique for lsi layout. In *IEEE International Conference on Computer Design*, pages 646-649. IEEE, 1985.

[36] Gershon Kedem and Hiroyuki Watanabe. Graph-optimization techniques for ic layout and compaction. *IEEE Transactions on CAD*, CAD-3(1):12-20, January 1984.

[37] Kenneth H. Keller. *An Electronic Circuit CAD Framework.* Technical Report UCB/ERL M84/54, UC Berkeley Electronics Research Laboratory, July 1984.

[38] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[39] Christopher Kingsley. A hierarchical, error-tolerant compactor. In *Proceedings of the Design Automation Conference*, pages 126–132, IEEE, 1984.

[40] Han Y. Koh, Carlo H. Sequin, and Paul R. Gray. Opasyn: a compiler for cmos operational amplifiers. *IEEE Transactions on CAD*, 9(2):113–125, February 1990.

[41] Paul Kollaritsch and Bryan Ackland. Coordinator: a complete design-rule enforced methodology. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers & Processors*, pages 302–307, IEEE, Rye Brook, NY, October 1986.

[42] Robert M. Kossey. Kcomp: a full chip compaction strategy. In *Proceedings of the IEEE 1987 Custom Integrated Circuits Conference*, pages 610–613, IEEE, 1987.

[43] Chuck Kring. *MKARRAY: An Array Assembler, GEM: A Gate-Matrix Module Generator.* Master's thesis, University of California, Berkeley, May 1988.

[44] Tom Laidig. Vulcan manual pages. In Rick L. Spickelmier, editor, *OCT Tools Distribution 3.0*, Electronics Research Laboratory, University of California, Berkeley, 1989.

[45] Robert P. Larsen. Computer-aided preliminary layout design of customized mos arrays. *IEEE Transactions on Computers*, C-20(5):512–523, May 1971.

[46] Robert P. Larsen. Interactive computer-aided editing of custom mos device layout designs. In *Proceedings of the 1973 International Microelectronic Symposium*, pages 2A–7–1—2A–7–10, International Society for Hybrid Microelectronics, San Francisco, CA, October 1973.

[47] Robert P. Larsen. Versatile mask generation techniques for custom microelectronic devices. In *Proceedings of the 15th Design Automation Conference*, pages 193–198, ACM/IEEE, Las Vegas, NV, June 1978.

[48] Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Reinhart and Winston, 1976.

[49] Jin-Fuw Lee and Donald T. Tang. Vlsi layout compaction with grid and mixed constraints. *IEEE Transactions on CAD*, CAD-6(5):903–910, September 1987.

[50] Thomas Lengauer. Efficient algorithms for the constraint generation for integrated circuit layout compaction. In *Proc. 9th Workshop on Graphtheoretic Concepts in Computer Science*, pages 219–230, 1983.

[51] Y-Z Liao and C. K. Wong. An algorithm to compact a vlsi symbolic layout with mixed constraints. *IEEE Transactions on CAD*, CAD-2(2):62–69, April 1983.

[52] Bill Lin and A. Richard Newton. Kahlua: a hierarchical circuit disassembler. In *Proceedings of the 24th Design Automation Conference*, pages 311–317, ACM/IEEE, Miami Beach, FL, June 1987.

[53] Sching L. Lin and Jonathan Allen. Minplex - a compactor that minimizes the bounding rectangle and individual rectangles in a layout. In *Proceedings of the 23rd Design Automation Conference*, pages 123–130, ACM/IEEE, Las Vegas, NV, June 1986.

[54] Chi-Yuan Lo, Ravi Varadarajan, and W.H. Crocker. Compaction with performance optimization. In *1987 IEEE International Symposium on Circuits and Systems Proceedings*, pages 514–517, Philadelphia, PA, May 1987.

[55] Abdul A. Malik. An efficient algorithm for generation of constraint graph for compaction. In *Proceedings of the International Conference on Computer-Aided Design*, pages 130–133, IEEE, November 1987.

[56] Carver Mead and Lynn Conway. *Introduction to VLSI Systems*. Addison-Wesley, 1980.

[57] Edward Minieka. *Optimization Algorithms for Networks and Graphs*. Dekker, 1978.

[58] A. Richard Newton. Event-driven algorithms for constraint solving. private communication, 1983.

[59] Lars S. Nyland, Stephen W. Daniel, and Durward Rogers. Improving virtual-grid compaction through grouping. In *Proceedings of the Design Automation Conference*, pages 305–310, June 1987.

[60] R. Okunda, T. Sato, H. Onodera, and K. Tamaru. An efficient algorithm for layout compaction problem with symmetry constraints. In *Proceedings of the International Conference on Computer-Aided Design*, pages 148–151, IEEE, November 1989.

[61] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, 1982.

[62] Stephen D. Posluszny. Sls: an advanced symbolic layout system for bipolar and fet design. In *Proceedings of the International Conference on Computer-Aided Design*, pages 346–348, IEEE, November 1985.

[63] Franco P. Preparata and Michael I. Shamos. *Computational Geometry.* Springer-Verlag, 1985.

[64] 25th DAC Program committee, editor. *25 Years of Electronic Design Automation.* Association for Computing Machinery, Inc., 1988.

[65] Mark Reichelt and Wayne Wolf. An improved cell model for hierarchical constraint-graph compaction. In *Proceedings of the International Conference on Computer-Aided Design*, pages 482–485, IEEE, November 1986.

[66] Sarma Sastry and Alice Parker. The complexity of two-dimensional compaction of vlsi layouts. In *Proceedings, IEEE International Conference on Circuits and Computers*, pages 402–406, New York, NY, September 1982.

[67] W. L. Schiele. Improved compaction by minimized length of wires. In *Proceedings of the 20th Design Automation Conference*, pages 121–127, ACM/IEEE, Miami Beach, FL, June 1983.

[68] M. Schlag, Y. Z. Liao, and C. K. Wong. An algorithm for optimal two-dimensional compaction of vlsi layouts. *INTEGRATION, the VLSI Journal*, 1:179–209, 1983.

[69] Robert Sedgewick. *Algorithms.* Addison-Wesley, 1983.

[70] Hyunchul Shin. *Two-Dimensional Routing and Compaction in Computer-Aided Design of Integrated Circuits.* Technical Report UCB/ERL M87/92, UC Berkeley Electronics Research Laboratory, October 1987.

[71] Hyunchul Shin, Alberto Sangiovanni-Vincentelli, and Carlo Sequin. Two-dimensional compaction by zone-refining. In *Proceedings of the 23rd Design Automation Conference*, pages 115–122, ACM/IEEE, June 1986.

[72] Hyunchul Shin, Alberto Sangiovanni-Vincentelli, and Carlo Sequin. Two-dimensional module compactor based on zone-refining. In *International Conference on Computer Design*, IEEE, 1987.

[73] David Tan and Neil Weste. Virtual grid symbolic layout 1987. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers & Processors*, pages 192–196, Rye Brook, NY, October 1987.

[74] Robert E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, 1983.

[75] Robert E. Tarjan. *Shortest Paths.* Technical Report, AT&T Bell Laboratories, Murray Hill, NJ, 1981.

[76] Masayuki Terai, Yoshihide Ajioka, Tomoyoshi Noda, Masaru Ozaki, Tsunenori Umeki, and Koji Sato. Symbolic layout system: application results and functional improvements. *IEEE Transactions on CAD*, CAD-6(3):346–354, May 1987.

[77] A. Thesen. *Computer Methods in Operations Research.* Academic Press, 1978.

[78] A. K. Tsakalidis. *Finding a Negative Cycle in a Directed Graph.* Technical Report A 85/05, Universitat des Saarlandes, Saarbrucken, West Germany, March 1985.

[79] Hiroyuki Watanabe. *IC Layout Generation and Compaction Using Mathematical Optimization.* PhD thesis, Department of Computer Science, University of Rochester, Rochester, NY, 1984.

[80] N. H. E. Weste. Mulga–an interactive symbolic layout system for the design of integrated circuits. *The Bell System Technical Journal*, 60(6):823–857, July–August 1981.

[81] Neil Weste. Virtual grid symbolic layout. In *Proceedings of the 18th Design Automation Conference*, pages 225–233. ACM/IEEE, Nashville, Tennessee, June 29 - July 1 1981.

[82] Neil Weste and Brian Ackland. A pragmatic approach to topological symbolic ic design. In *VLSI 81 International Conference*, 1981.

[83] John D. Williams. Sticks–a graphical compiler for high level lsi design. In *AFIPS Conference Proceedings, 1978 National Computer Conference*, pages 289–295, Anaheim, CA, June 1978.

[84] John D. Williams. *STICKS–A New Approach to LSI Design.* Master's thesis, Massachusetts Institute of Technology, 1977.

[85] Wayne Wolf, Robert Mathews, John Newkirk, and Robert Dutton. Two-dimensional compaction strategies. In *Proceedings of the International Conference on Computer-Aided Design*, pages 90–91, IEEE, November 1983.

[86] Wayne H. Wolf. *Two-Dimensional Compaction Strategies.* Technical Report, Integrated Circuits Laboratory, Stanford University, March 1984.