Monte Carlo Methods For 2–D Compaction

Thesis by

R.C. Mosteller

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

1986

(Submitted May 19, 1986)

Acknowledgments

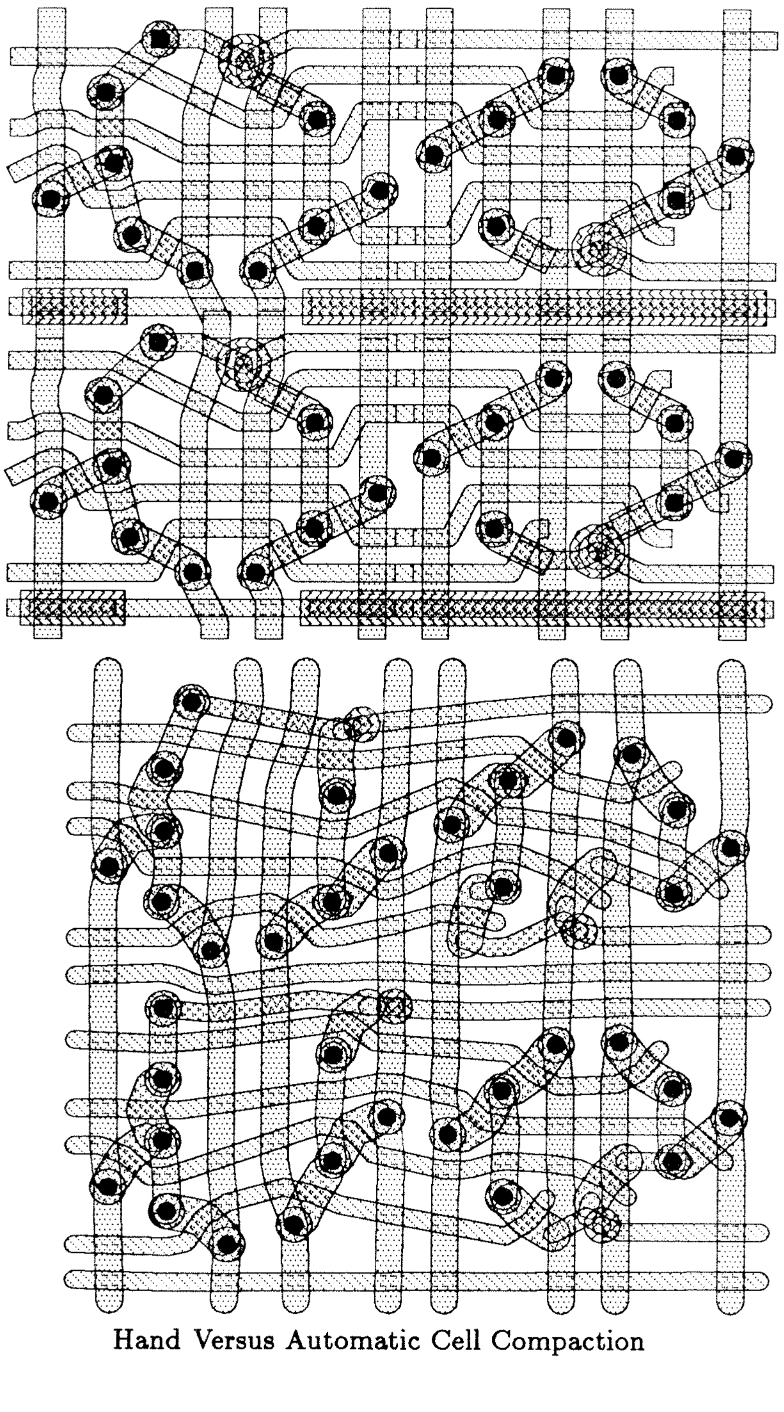
I would like to thank Chuck Seitz for starting me onto the road towards a PhD. I owe special appreciation to Sandy Frey for his constant help and guidance, and for his patience with me. Special thanks also to my advisor, Jim Kajiya, for his years of patience, support and insight. I would like to thank Roberto Suaya for his many nightly discussions. I would like to acknowledge some of the special people who worked with me at Caltech: Don Speck for his insight into cell design, Telle Whitney for discussions on data representation, Bill Dally for his help with cell design and discussions, Calvin Jackson for his help with the intricacies of TEX, Bill Athas for his help with text review, and Dan Whelan for his patience with me while we shared an office.

Double, double toil and trouble,
Fire burn and caldron bubble.

SHAKESPEARE, *MACBETH*

Abstract

A new method of compaction for VLSI circuits is presented. Compaction is done simultaneously in two dimensions and uses a Monte Carlo simulation method often referred to as simulated annealing for optimization. A new curvilinear representation for VLSI circuits, specifically chosen to make the compaction efficient, is developed. Experiments with numerous cells are presented that demonstrate this method to be as good as, or better than the hand compaction previously applied to these cells. Hand compaction was the best previously known method of compaction. An experimental evaluation is presented of how the run time complexity grows as the number, N, of objects in the circuit increases. The results of this evaluation indicates that the run time growth is order $O(N\log(A))f(d)$ where $f(d)$ is a function of the density, d, and A is the initial cell area. The function $f(d)$ appears to have negligible or no dependence on N. A hierarchical composition approach is developed which takes advantage of the capability of the curvilinear representation and the 2–dimensional compaction technique.



Hand Versus Automatic Cell Compaction

Table of Contents

List of Figures

List of Tables

1
The Problem: 2–D Compaction

The amount of time required to design integrated circuits has been increasing rapidly with design complexity[Moore 79, Lattin 79]. Present day chip designs take several years to complete with a large portion of the time spent on layout, the transformation of electronic design to physical mask geometry. "Traditional" layout is accomplished by a hand design process with the aid of automatic drafting machines. Much of this effort is spent on compaction, i.e. minimizing the silicon area needed for the design. Elaborate geometric design rule checking programs are then necessary to insure an error-free design. These programs are run as a batch process. Correction of geometric design rules errors is another time consuming process. Updates to this design are tedious at best.

Early attempts at automating layout compaction were simplified by using only rectangles orientated along the x and y axis. The first attempts minimized the layout along each coordinate axis independently. This method is known as 1–D compaction. One problems with this approach is that compaction in the horizontal axis and the vertical axis, do not commute and the resulting layout is extremely sensitive to the input geometry. More sophisticated attempts coupled the two 1–D compactions into a single compaction. Unfortunately this method has been shown to be NP-complete[Sastry 82]. Both methods are limited to wires that do not bend. The automatic insertion of bends in wires has been called automatic jog insertion. Some limited automatic jog insertion has been tried with mixed results. In all of these approaches devices are rigid elements with fixed orientation. They are only allowed to move along the x or y axis. Current automatic integrated circuit compactors are not able to produce a design as compact as a design using traditional methods. The automated compactors do not take advantage of the freedom allowed by the chip media, i.e., they are limited to orthogonal geometry and orthogonal compaction methods.

This thesis presents a new method for integrated circuit compaction using a curvilinear framework. With non-rectangular geometry, even without compaction, a better utilization of the area on the plane can be achieved by allowing arbitrary relative orientations of the different circuit elements. Wires are treated as rubber bands that are tautly stretched around obstacles. There is no need to have automatic jog insertion because there are no rectilinear constraints in the framework. Devices are treated as malleable objects that are allowed to rotate and bend within parametrically controlled limits.

The optimization of this framework is more complex than the 2–D compactors previously mentioned. We compact the framework by simulated annealing[Kirkpatrick 83]. Annealing is the process of heating and very slowly cooling a metal to form an optimized structure. Simulated annealing is similar to this technique for metals. This compaction method produces extremely compact designs which combines area and wire capacitance as optimization criteria. The compacted design is as dense or denser than a hand compacted design. This technique was first presented at the Workshop on Statistical Physics in Engineering and Biology[Mosteller 84].

1.1 Problem Definition

Integrated circuit compaction is the translation of a geometric physical representation of a cell which is geometrically design rule correct to a second geometrically design rule correct physical cell with a near minimum area. Geometric design rules are constraints associated with the relative position and overlap of the geometric figures. Geometric design rules imply a set of inequality constraints. These rules are necessary to insure a working integrated circuit. A sample of an nMOS rectilinear cell is shown in Figure 1.1. A cell is composed of devices and interconnecting wires. Devices and wires may be on one or more layer as shown.

The purpose of compaction is to reduce the area of the design while preserving topology and without violating any geometric design rules. A reduced area, given a fixed die size, provides the opportunity for additional functionality over the non-compacted design. By reducing area, wire lengths are also shortened, resulting in less time for signal propagation and in turn increasing the operational speed of the circuit.

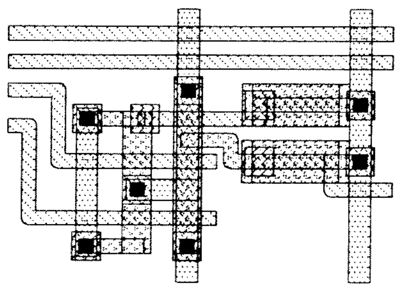


Figure 1.1 Sample nMOS Design

The input specification from which layout is compacted is one that can be characterized as a sized schematic representation of the circuit. This is to say that in the input phase one gives the ports, contacts, transistors(appropriately sized), and wiring. This representation is a geometrically design rule correct initial configuration of the circuit. This representation by itself contains all of the elements necessary to generate a correct layout. The purpose of the translation is to optimize the output mask geometry while still preserving the geometric design rules and circuit topology. This translation is the task that is commonly done "by hand", and often referred as "micron-hacking". The task is quite time consuming and error prone. To be able to do it automatically is certainly a great leap forward.

We limit the compaction problem to a fixed topology. A topology for an integrated circuit cell can be viewed as a set of planar graphs. The nodes in the graphs represent the devices such as contacts, transistor and ports that are the interface to the outside world. The arcs in the graph are the wires. A graph is constructed for each set of wires that has a geometric design rule. For example, the nMOS cell in Figure 1.1 would have a planar graph for the metal wires and a planar graph for the polysilicon and diffusion wires. The nodes in the graph may be moved as long as the planarity is preserved.

All compactors use a fixed topology. The reason for this is that topological changes are extremely difficult. Simple local topological changes could be achieved such as merging connected contacts of the same type which are in close proximity. Previous compactors are far more restrictive in that geometry is limited to rectangles, wires are not allowed to bend, and devices have fixed orientation and shape.

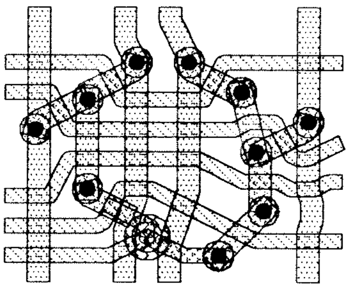


Figure 1.2 Curvilinear Sample Design

Hand designers do not limit themselves to orthogonal geometry as shown in Figure 1.2 of a hand designed nMOS cell. Notice in the cell that the wires deform around obstacles and use many angles. The devices are at angles that produce a compact cell. This type of hand design produces tightly compacted cells. Curvilinear hand compactions is an art that requires a highly skilled designer. The success of compaction greatly depends upon the skill of the designer. Curvilinear hand compaction is extremely time consuming. Our compactor is designed for this type of compaction.

Although hand compaction produces extremely compact designs, it is almost impossible to alter or update. Generally if a curvilinear cell requires a logic change, a complete reconstruction of the hand compaction process is required. This would not be a major problem with an automatic curvilinear compactor.

When a hand designer compacts a cell he starts with an idea of the desired cell shape and port position. Cells are not compacted in a vacuum but with respect to the environment in which they will be placed. A cell may be placed in a design at two different location which could require completely different compaction. Our compactor will allow the ability to control the final shape and port position.

1.2 Approach

To apply simulated annealing to the two-dimensional circuit compaction problem requires a simple representation that can be modified easily. The variety of objects -transistors, contacts, resistors, wires, etc.- and sizes of objects found in VLSI circuits makes finding such a representation more difficult. Earlier applications of simulated annealing to molecular dynamics suggested the representation that we have developed. Primitive components called bubbles are connected by wires. Bubbles are hard objects that will be moved around as if they were molecules during the annealing process. Wires connecting bubbles are stretchable objects which are continuously modified to follow the shortest path between the bubbles they connect while preserving all required geometric design rule constraints. They constitute an attractive force between bubbles in the annealing process.

Geometric design rules are described through a set of tables reflecting the technology of the current implementation. Rules, used to reference these tables, are assigned to individual bubbles and wires. A unique feature of the representation is that all geometric design rules are broken up into constraints in several independent layers. Each layer is assigned a color, some of which may be hypothetical for geometric design rule checking. To check the geometric design rules, only a logical function of Boolean variables is needed to determine which layers are relevant between two primitives. Checking the geometric design rules within one layer is independent of any other layer. The checking is therefore reduced to a monochrome problem. Finally, the geometric design rules are all satisfied if they are satisfied in each of the relevant independent layers. When circuits are initially entered through an interactive graphic editor, a check is made to assure that no geometric design rules are violated. The geometric design rules continue to be maintained through out the annealing process. Details of how the geometric design rules are represented are described in Chapter 3. An interesting feature of this method of compaction occurs when the technology changes. Since it is table driven, the tables can be modified to reflect the changes and the circuit reannealed to incorporate the new geometric design rules.

The variety of objects in VLSI circuits all are built from the two types of primitive components as described in Chapter 3. A structure that will be used many times can be constructed using several bubbles and wires as a model. It can then be instantiated wherever it is needed. By building the models out of the the two basic primitive components, the individual pieces of each model may be moved independently and the model is thus malleable in the annealing process.

As this research was being developed, the absolutely essential requirement for simplicity - particularly simplicity of modification of the data structure - became increasingly obvious. Very small cells might be compacted with a reasonably small number of computer cycles with standard representations, but those cells could also be compacted reasonably by hand. As cells became large enough to make the hand compaction time burdensome, the need for a representation that would reduce the annealing time also became apparent. That has been the major concern in the development of this representation.

The desirability of including an automated composition capability also was clear from the start as well as the need for initial floor planning. The compaction system presented here has been developed to support an overall VLSI design system that includes these capabilities. Such a composition methodology is described in Chapter 8.

1.3 How We Compact a Cell

An example is presented to gain an understanding of the compaction process. We are going to show how we compact a 4 to 1 multiplexor cell from start to finish. The logic diagram for the 4 to 1 multiplexor is shown in Figure 1.3. First, the cell is shown in its initial input specification. The representation of the devices and wires will be discussed. Next a series of figures will show how the cell is annealed from the high temperature configuration through the low temperature configuration. Simulated annealing will be discussed along with the objective function to be optimized. The final phase of compaction will be shown. A hand compacted version of the cell will be compared to the automatically compacted version.

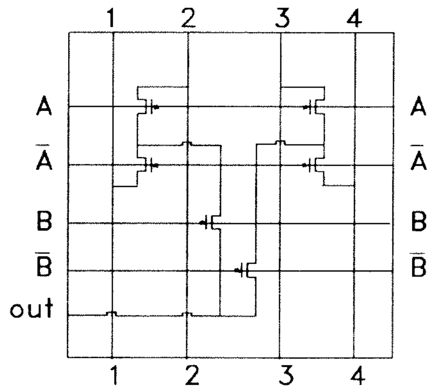


Figure 1.3 4:1 Multiplexor Logic Diagram

A transistor diagram for the example nMOS cell is shown in Figure 1.3. The relative position of the connectors or ports to the outside world are shown. The input signals are 1 through 4 where the select lines are A, $\bar{A}$, B, and $\bar{B}$. The output of the cell is *out*.

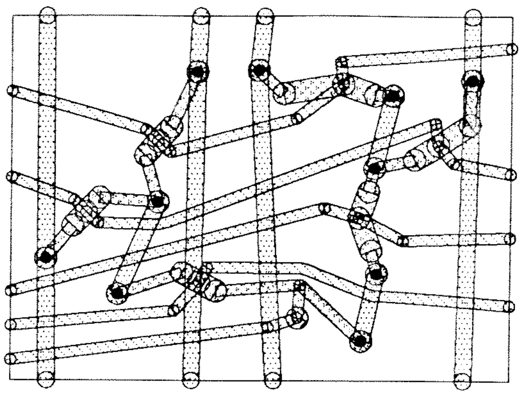


Figure 1.4 4:1 Multiplexor Initial Design Configuration

The initial specification for the cell is shown in Figure 1.4 where the pattern for each layer is shown in Figure 1.5. Notice that the cell is composed of two types of primitives. There are round objects called "bubbles" and interconnecting lines called "wires". These objects are the primitive constructors from which integrated circuits are built. The bubbles are the end points of the wires. If you look closely you will notice that the wires curve around obstacles.

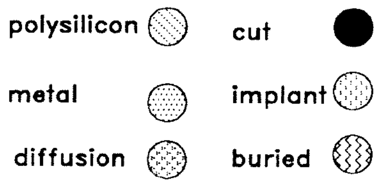


Figure 1.5 nMOS Color Pattern Representation

The cell was created using a graphic editor which is described in Chapter 5. This editor allows the placing, moving, and deleting of devices. A device, that we call a "model", is a composition of wires and bubbles that forms a transistor or a contact. These devices can be connected together by a wiring command. The editor does not allow any instantiation that would generate a geometric design rule error.

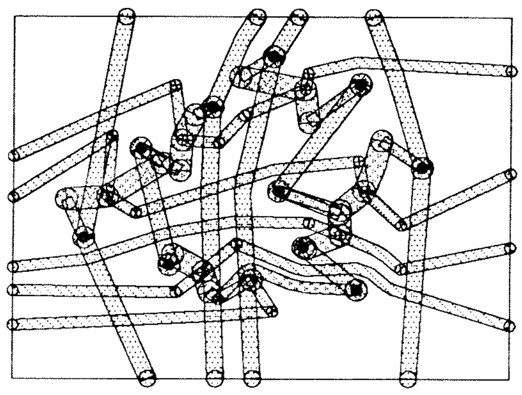


Figure 1.6 4:1 Multiplexor at a High Temperature Configuration

The first step in the annealing process is to "heat" the cell. This randomizes the system to an initial configuration, removing any past history of configurations. The cell is shown at a high temperature in Figure 1.6. The bubbles in the cell can be compared to molecules in a metal. A major force on the bubbles is the pull from the interconnecting wires. Each bubble can move in any direction provided it is not blocked by another bubble and does not create a geometric design rule error.

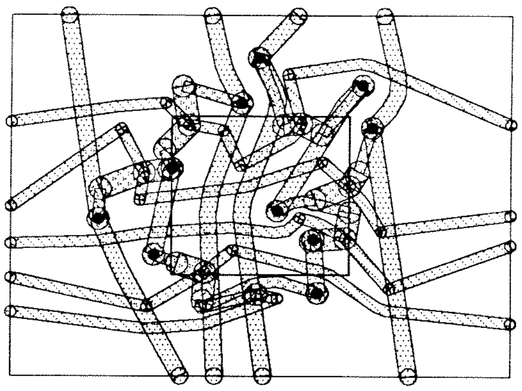


Figure 1.7 4:1 Multiplexor at Initial Cooling

A close look at the cell will reveal that the bubbles have been pulled toward the interior of the cell. This pull is what shapes the cell. The pull is caused by a central potential that we control with the "membrane". The membrane position is shown in Figure 1.7 as an interior rectangle.

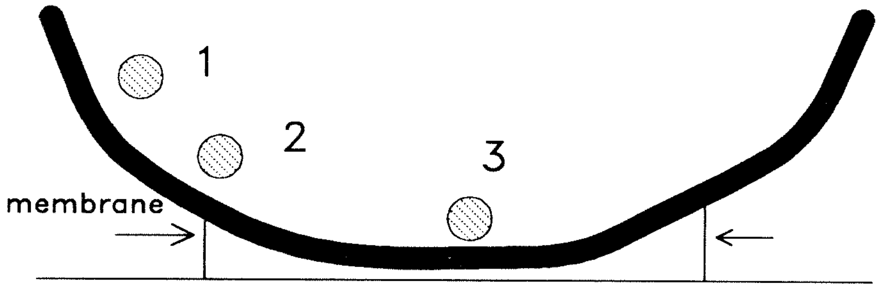


Figure 1.8 4:1 Membrane Bowl

Conceptually the central potential can be considered as a gravity bowl as shown in Figure 1.8. Bubbles near the edge of the bowl have a strong pull toward the center while bubbles near the center have weak pull and can move about freely. There is a separate central potential function for each of the X and Y axes.

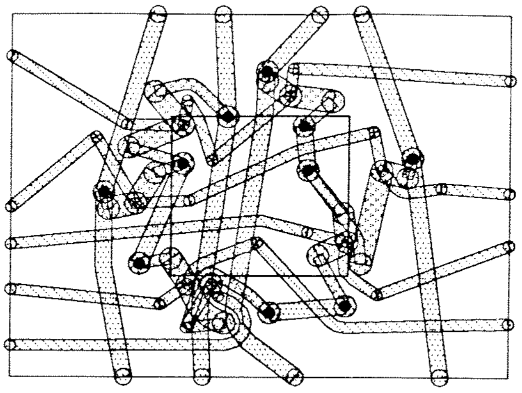


Figure 1.9 4:1 Multiplexor at Cooling stage 1

As the system is cooled the effects of the membrane on the cell is shown in Figure 1.9. Bubbles near the perimeter are subjected to a large pull which implies a lower effective temperature than the bubbles within the membrane that have a higher effective temperature. This means that bubbles near the perimeter are restricted to move toward the center while bubbles inside are only slightly affected by the central potential and mostly influenced by the pull of the wires and bubble bumping. The effect becomes more evident as shown in Figure 1.10 which shows the cell after further cooling.

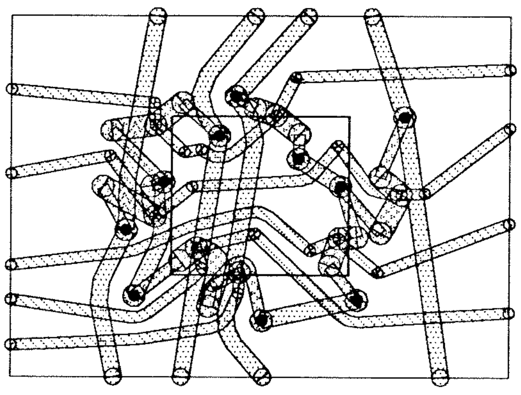


Figure 1.10 4:1 Multiplexor at Cooling stage 2

Taking a closer look at the structure of the cell, shows that it is composed of two primitives, bubbles and wires, shown in Figure 1.11. Bubbles are solid objects that can be moved. They have a location, a list of connected wires, and a rule. The rule specifies properties of the bubble. For example, the rule specifies each fabrication layer and the width of the object in those layers. A wire can be considered as a tightly stretched band wrapping around obstacles. Wires are composed of straight pieces called "segments" and curved pieces called "arounds". Like bubbles they have a rule. Wires are moved only as a side-effect of a bubble move.

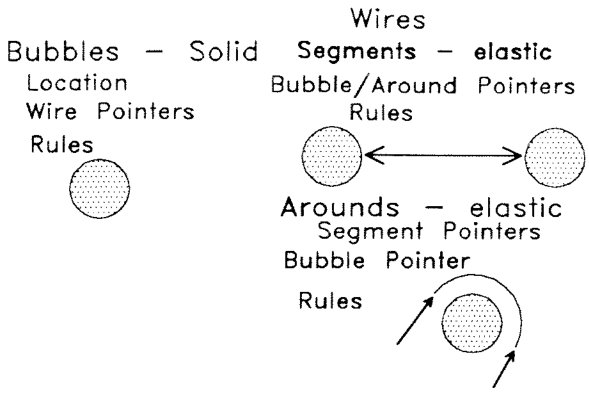


Figure 1.11 Bubble and Wires

The transistors and contacts are constructed from the primitives which are called models as shown in Figure 1.12. The models are multi-layered objects. Transistors have a requirement that their shape be preserved at the completion of compaction in order to perform properly. During the annealing process, the transistors are allowed to deform so that they can move to a more compact configuration. In the final stages of annealing the transistors are pulled back together to form a proper device. The models in their proper shape are shown in Figure 1.12 while the distorted models are shown in the cell being compacted. In order to insure that the models return to their proper shape there is a quadratic cost function on the wires within models.

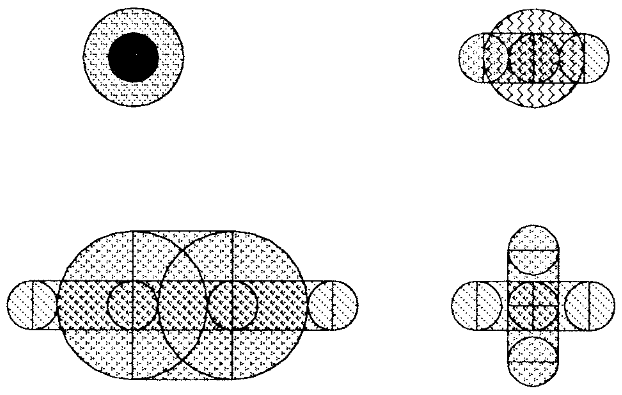


Figure 1.12 Model Examples

Models may have more than one valid shape. The long transistor in Figure 1.12 can bend slightly and still be valid. The control over bending is described in Chapter 3.

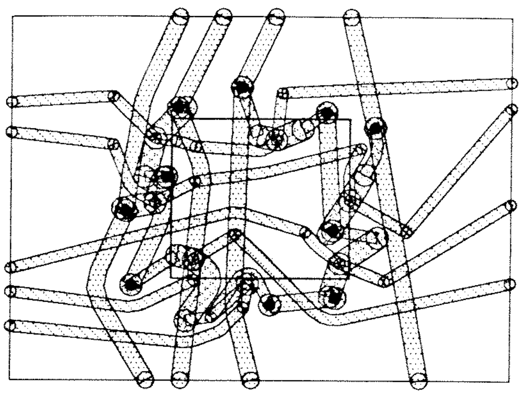


Figure 1.13 4:1 Multiplexor at Cooling stage 3

Figure 1.13 and Figure 1.14 shows the cell at progressively further stages of cooling. Notice how the transistors are being pulled by the membrane and by their internal pull.

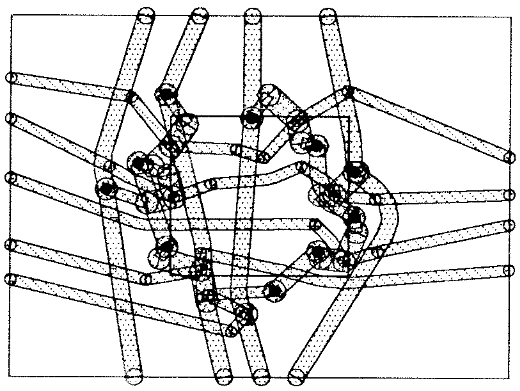


Figure 1.14 4:1 Multiplexor at Cooling stage 4

The following is an overview of how the cell is compacted using simulated annealing. More detail is described in Chapter 6. The annealing process operates by first simulating the system at a high temperature. The simulation at a specific temperature operates by moving bubbles in a probabilistic manner. First a bubble and a direction is chosen at random. If that move can be made without moving any other bubble and without generating any geometric design rule error, then the changes in the cost function, which is referred to as the delta cost, caused by that move are computed. If the delta cost of the move is negative, the move is accepted. This reduces the system energy. If the delta cost is positive the move is accepted based on a probability function of the temperature and delta cost. This approach allows bubbles that are stuck in a local minimum to jump out with some probability. After the system nears equilibrium at that temperature, the temperature is reduced in stages where near equilibrium is reached at each stage. The temperature stages and the equilibrium criterion are referred to as the "annealing schedule". This process allows the cell to be compacted to a near global minimum without being stuck in a local minimum.

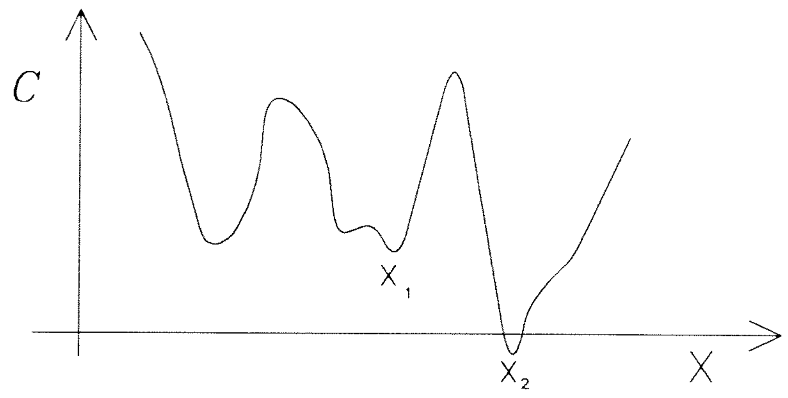


Figure 1.15 Hypothetical Cost Function

For example, consider a cost function such as that shown in Figure 1.15. If only down hill moves were taken it would then be possible to get stuck at configuration x_1. However with simulated annealing the hill could be jumped over and x_2 found. Simulated annealing will always find the global minimum provided at each stage of cooling equilibrium is reached. For some problems, reaching this point can take an extremely large amount of time.

The objective function that we will be optimizing is a sum of the cost of the wires and the central potential cost. Each wire type may have a different cost. The reason is that some wires need to be shorter than others for electrical reasons. For example, in nMOS we would like the polysilicon wires to be shorter at the expense of longer metal wires. The cost for polysilicon wires would be greater than the metal wires. Model wires would also have a larger cost. The central potential cost controls the shape of the cell.

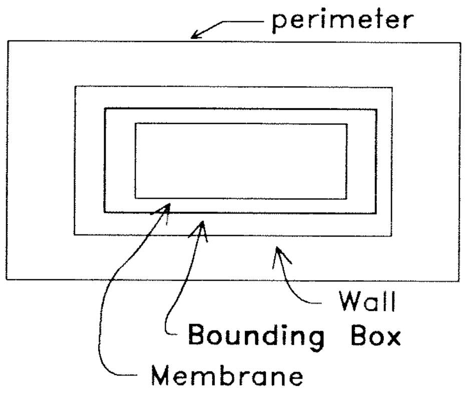


Figure 1.16 Cell Boundary

The membrane rectangle as shown in Figure 1.16 is placed at the desired shape for the final cell. Bubbles outside of the membrane incur a large cost pulling them toward the membrane. Bubbles inside the membrane incur a decreasing cost toward the center of the cell. The previous compaction example shows how the membrane affects the final shape.

Bubbles are restricted to be within the cell boundary as shown in Figure 1.16. Further restrictions can be imposed by the wall which limits the bubbles to be within the wall boundary. The shape of the cell is not limited to a rectangle as shown but may be a polygonal shape. Connectors to the outside world of the cell are called "ports". They are restricted to have their center on a perimeter side. A port may not change sides automatically.

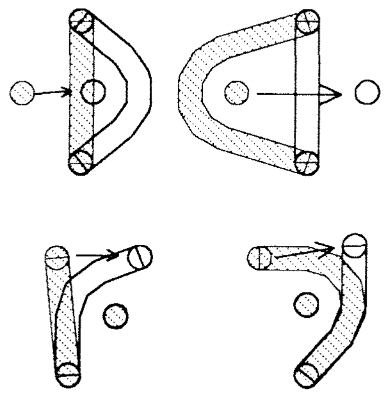


Figure 1.17 Bubble Move Example

When a bubble is chosen to be moved by the simulator, an algorithm is used to see if the bubble can be moved and if so, the proper data is constructed. This algorithm is called "bubble move". The algorithm first checks if the bubble can be moved without violating a design rule error. If it can move, then the appropriate data structures are constructed. Bubbles are allowed to push wires where examples are shown in Figure 1.17. The filled-in regions are the current position and the outlined positions are the new locations. Bubbles are not allowed to push other bubbles. A conceptual way to visualize the bubble move algorithm is to consider the bubble to be moved as a solid round object that is being pushed. The wires are rubber bands tautly stretched wrapping around pegs, and the other bubbles are the pegs. Notice that the solid object can push the bands but is blocked by the pegs. This allow only local moves. The bubble move algorithm is explained in Chapter 4.

When a bubble move can be performed the necessary changes for the move are stored in two queues called "add queue" and "delete queue". The delta cost for the simulation is then calculated from these queues.

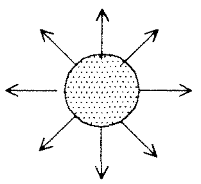


Figure 1.18 Allowable Bubble Moves

We will use one of the eight possible directions as shown in Figure 1.18 for the move direction. This direction is picked at random. The move length is fixed for each temperature. However, the move length is decreased as the cell gets tighter.

Initially there is a lot of freedom for the bubbles in the cell, hence we use a large move distance. When the cell is cooled the freedom for the bubbles is reduced, hence a smaller move size. For all the previous stages the move size was large. For the next stages the size has been reduced.

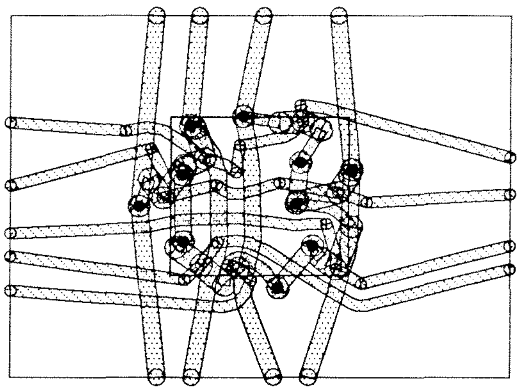


Figure 1.19 4:1 Multiplexor at Cooling stage 5

As the cell is cooled as shown in Figure 1.19, the bubbles move into the membrane and the models start to close in although they still can move. At this time the move length is reduced. Further cooling is shown in Figure 1.20.

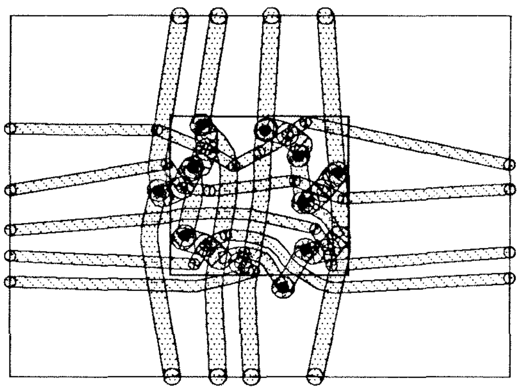


Figure 1.20 4:1 Multiplexor at Cooling stage 6

In the final stages of cooling the affects of the central potential are clearly visible as shown in Figure 1.21.

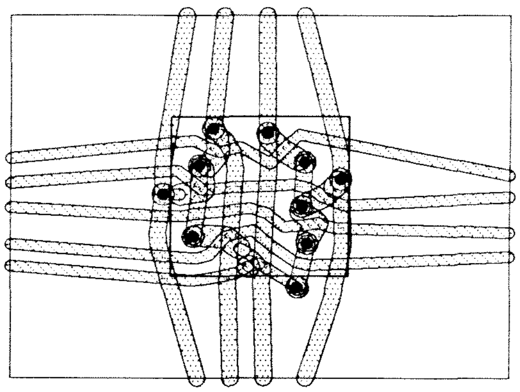


Figure 1.21 4:1 Multiplexor at Cooling stage 7

The final stage of compaction removes the central potential and places the wall at the minimum bounding rectangle region which is described in Section 3.9.1. At this time all distortions due to the central potential are removed. The final cell is shown in Figure 1.22.

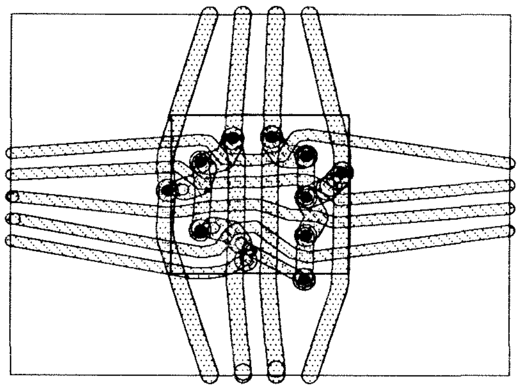


Figure 1.22 4:1 Multiplexor at Freezing

The cell is constricted to its minimum bounding rectangle and shown in Figure 1.23. This cell is placed next to the hand compacted cell. Notice that the automatically compacted cell is smaller than the hand compacted cell by about 10%.

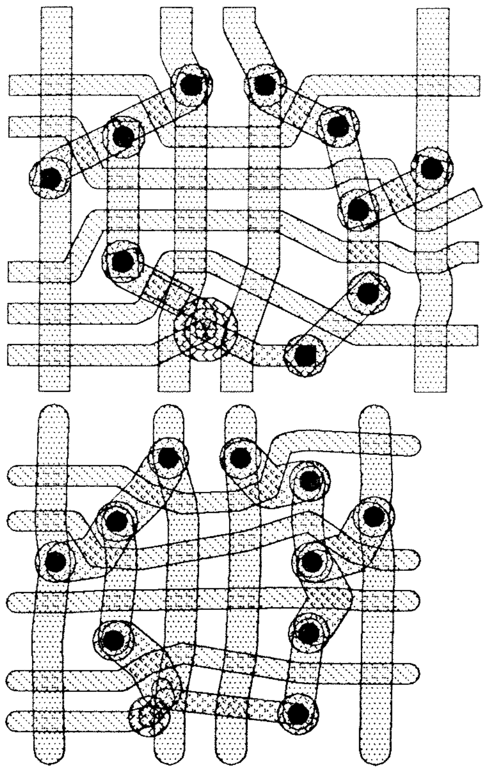


Figure 1.23 4:1 Hand and Automatic Compacted Cell

This cell would not be very useful if it could not be composed easily with other cells. An example of composition is shown in Figure 1.24. This cell is a composition of the cell just compacted and a similar cell.

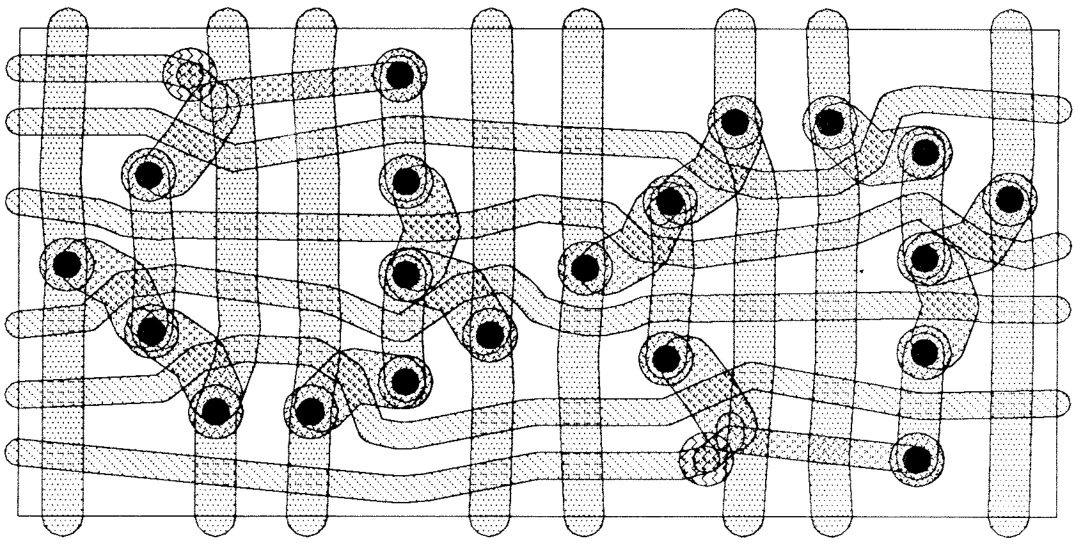


Figure 1.24 4:1 Composed Cells

1.4 System Overview

The “Bubbleman” system is composed of a collection of interacting modules through a common data base. Each module is a functional unit that is devoted to a particular aspect of the system. The modules are executable code units that are loaded and executed upon demand.

The Bubbleman system is written in the language **Mainsail**[1][Wilcox 85]. **Mainsail** is an Algol like language which supports object oriented programming. An advantage of **Mainsail** is that it is portable. The Bubbleman system was create on a DEC 2060 computer system. The exact same code was compiled and executed on a VAX 780, VAX 750, and a Sun which run the Unix operating system. Also, Bubbleman was compiled on a IBM 4341 using CMS. The Bubbleman program run successfully on all the mentioned systems with no changes to the sources code.

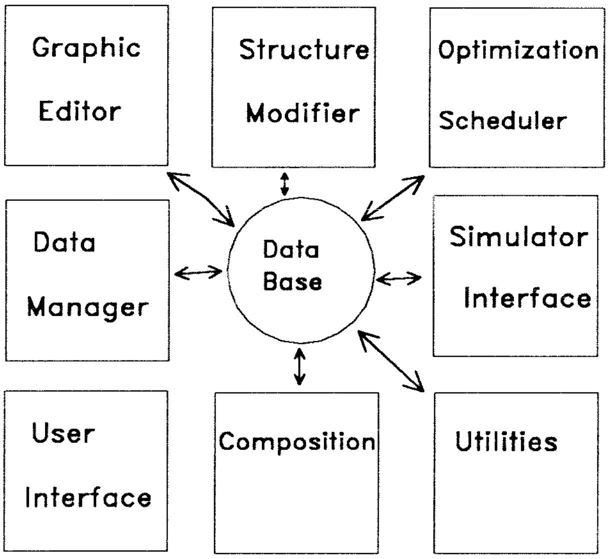


Figure 1.25 4:1 Bubbleman System

The modules for the Bubbleman system are shown in Figure 1.25. In the center is shown the database with the data manager on the side. The user interface module is a centralized interface to the user. All control over the modules comes from here.

The graphic editor is the editor for modifying cells which will be described in Chapter 5. The structure modifier, consisting of the bubble move algorithm and associated functions, is described in Chapter 4. The graphic editor, composition and optimization scheduler use this module. The optimization scheduler is the implementation of the annealing process as described Chapter 6. An advantage of this module is that it is very small and can easily be altered for the experiments that are described in Chapter 7. The simulator interface allows translation of cells to simulator formats such as Mossim[Bryant 82]. The utilities provide functions such

[1] **Mainsail** is a registered trademark of **XIDAK**, Inc.

as trimming wires, and absorbing bubbles. The composition implements the binary composition as described in Chapter 8.

1.5 Thesis Overview

The general strategy of the thesis is first to give an overview of previous work on structured design methodology and early compaction methods. Next the structure of the primitives is presented followed by how the structures are modified. The simulated annealing optimization technique is presented with how it is applied to cell compaction. Experimentation and analysis of the compaction method is presented. How the compacted cells are assembled into a chip and the way in which the compaction is guided for composition is presented.

Chapter 2 gives the background of layout methodology and compaction. The method known as structured design for VLSI is presented as related to the layout problem. An overview of leaf cell design methods is given. Current composition techniques are explored and criticized. Next, the method of 1–D compaction is reviewed followed by the early attempts at 2–D compaction. The limitations of current compactors are explored.

Chapter 3 introduces a new framework specifically designed for 2–D curvilinear compaction. This framework is easily tailored for a specific technology. A simple set of primitives are introduced that will be used to construct integrated circuits. The sophisticated method of representing geometric design rules in a simple manner for the primitives is described. An example of a nMOS technology is described in addition to ways in which the structure may be modified for a change in a technology. A metric for a cell area is introduced which will be used in the composition of cells.

Chapter 4 describes how a structure is modified. The bubble move algorithm is described. Utility are also presented. Modification to the cell polygonal boundary is described.

Chapter 5 describes a graphic editor for initially creating a cell and performing updates to the cell. A unique feature of the editor is a uniform method to update the data structure and process errors.

Chapter 6 describes the simulated annealing optimization method. How we use this method to compact integrated circuits is described. The objective function is presented.

Chapter 7 describes experimental results of the compaction process. Analysis is performed on these results. The complexity of the system is estimated. To determine if a cell compacted with the curvilinear models can be fabricated, a set of cells was fabricated and tested.

Chapter 8 describes a new method for composition. A binary composition operator is presented which can be used to assemble an integrated circuit chip. A new method for directing the compaction for cells specifically from the floor plan is described. With this method cells are compacted for their environment.

Chapter 9 ties the thesis together by unifying the various pieces of thesis in the conclusions. Extensions to the system are presented, as are ideas for future research.

2
Background

Clearly problems associated with designing integrated circuits has only increased as the complexity of the circuits has increased. The structured design technique of Mead and Conway has been generally accepted as the best approach for managing this complexity. The basis of the technique is to separate the design of individual functional units from the design of their interconnection. This chapter presents a summary of the tools that have been developed for these tasks.

2.1 Layout Methodology - Design Styles

The structured VLSI design methodology[Mead 80] provides many paradigms to assist designers in managing large complex designs. Using this methodology, one partitions the design into a hierarchy of blocks with well-defined interfaces between the blocks. Each block may have a functional, structural, and physical representation, and occupies a contiguous area on the chip. Replicated blocks need to be designed only once, reducing the complexity. The hierarchy allows the design task to be partitioned into several disjoint tasks.

The design process proceeds in a top down fashion with decomposition and refinement at each level. The design is then assembled with a bottom up implementation. The top down design is the architectural design phase where wiring strategy, major functional blocks, and the general floor plan is defined. The wiring strategy is addressed at the beginning of the design which eliminates costly routing and placement. This phase continues with cell estimation and cell detailing. Next there is a bottom up assembly of the design by composing the blocks into larger blocks until the chip is completed.

The separated hierarchy[Rowson 80] restricts the design into two type of blocks called leaf cells and composition cells, as shown in Figure 2.1. The leaf cells contain the primitive elements. A leaf cell is a physical block with an interface to other cells called connectors which lie around the perimeter of the cell. The composition cells define the interface between cells where a cell is either a composition cell or a leaf cell. The leaf cell contains the representation while the composition cell contains the interface between cells. This restricted hierarchy allows definition of the composition independent of representation.

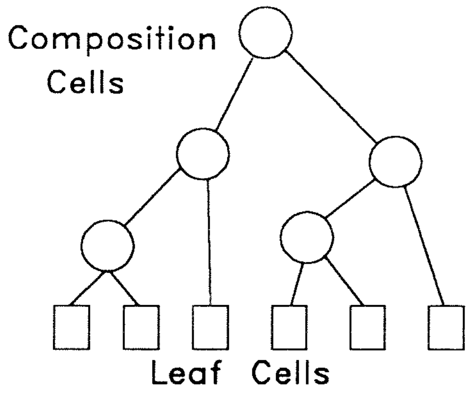


Figure 2.1 Separated Hierarchy

The philosophy for the layout method of designing a cell is either to design the cell followed by a geometric design rule checking method to insure a valid design, or to use a method that only allows correctly designed cells. The design and check method allows for the most flexibility to designer. The drawback is that the design and check phase needs several iterations to produce a valid cell, whereas the second method only allows correctly designed cells. These systems are generally not as flexible.

The design process for a cell starts from an initial design, and proceeds through many design refinements. The method used to design the cell should be amenable to this design process. The design and check method performs reasonably well for small designs; however this method becomes unwieldy as the problem size grows. The methods that only allow correctly designed cells operate at all levels.

Separate layout strategies have evolved for leaf cell design and cell composition. Methods for leaf cells design are usually graphical in nature. Tools are either interactive graphic or a graphic language. Composition tools specify the tiling in the plane. These tools are usually language based because of the flexibility of such languages[Trimberger 81].

2.1.1 Leaf Cell Design

Leaf cell design is the process of creating a physical representation for a cell. Leaf cells contain as primitive elements both polygons and wires. The physical representation is in the form of mask geometry, and is usually created with an interactive graphic tool or a graphic language.

The language entry method provides the most flexibility since languages generally provide capabilities such as variables, expressions, conditionals, and loops. The first graphic language Lap[Locanthi 78] was embedded in the Simula[Birtwistle 73] programming language which allowed the full power an object-orientated language to be applied to leaf cell design. The primitives specified in the Earl[Kingsley 82] design language system were in terms of X and Y constraints that provided a lot of flexibility; however, Earl was quite tedious to use. Initially, designs were specified in terms of CIF primitives namely - polygons, boxes, and wires[Sproull 79]. Later languages provided symbolic input as in Pooh[Whitney 83]. The language based systems are of the design and check type.

The interactive graphic method provides a good user interface with rapid updates. Entry in early systems such as Icarus[Fairbairn 78] is in terms of low level primitives with batch geometric design rule checking. More recent systems such as Magic[Ousterhoust 84] provided on the fly geometric design rule checking. Symbolic systems such as Rest[Mosteller 81] allows entry in terms of a "stick" notation where the leaf cell is defined with a sketch. This system couples a compactor to the editor to produce a correct design. Other symbolic editors such as Pooh[Whitney 83] couple a geometric design rule checker with the interactive design and disallow geometric design rule errors.

2.1.2 Cell Composition

Cell composition is the process of assembling the leaf cells together to form an integrated circuit. The process involves mating cells so that connectivity is insured between the cells and the produced cell is geometric design rule correct. The process of mating cells is called "pitch matching". This is accomplished by abutment if they perfectly align, or by stretching the cells, or adding an interface cell to mate the cells. The purpose of the mating is to line up the connectors to insure connectivity and to place the cells in such a manner that connectivity is preserved and the combination is geometric design rule correct.

The specification of a composition is best described with a language[Trimberger 81, Mosteller 82, Segal 84, Ackland 83]. Generally these languages assume the cells are rectangular with connectors along the perimeter. When two cells are joined along a side it is assumed that there is a one to one mapping of the connectors. It is not necessary that the connectors line up. The advantage of a language based specification is that large composition can be specified with a small number of statements. Languages provide replication via iteration for cells that are duplicated. Some experiments with graphic composition have been tried on the Riot[Trimberger 82] system. The language based composition specification is amenable to the update design process while the graphic specification is difficult to update.

The three techniques that have been used for composition are: abutment of cells, stretching cells, and adding routing cells. The abutment strategy requires that the cells to be joined have their connectors lined up and when the cells are juxtaposed no geometric design rule violation is generated. The stretching method increases the size of both cells to be joined so that the connectors line up and the cells are pitched matched. The routing method adds a routing cell to provide the connectivity between the cells to be composed. All composition systems use one or more of these composition methods.

Abutment systems rely on the designer of the leaf cells to place the connectors at the proper position and establish the correct cell pitch for composition. This method performs reasonably well for small designs and with minimum updates. The problem with abutment is that the burden of composition is forced on the leaf cell designer. An interesting abutment system is Pooh[Whitney 85] which allows composition of polygonal cells. This system provides connectivity checking and geometric design rule checking.

The stretching method assumes that the leaf cells are geometric design rule correct and stretching the cell at connector boundaries would not introduce a geometric design rule error inside the cell. The stretching process expands the cells to be composed until the connections line up and the cells are pitch matched. The stretching at each connector creates a rift line at the connector through the cell where the cell is extended. The advantage of this method is that a correct composition is formed. The composition strategy is simplified. The disadvantage is that the cell size is always increased and possibly destroys the circuit integrity. In addition, the total wire length may be substantially increased.

Stretching has a major disadvantage. Suppose cell A which is to be composed with itself side by side horizontally and the connectors on the left side are below the connectors on the right side. If two of these cells are composed, the second one would need to stretch for the connectors to mesh with the first, and both cells would need to grow in order to pitch match. Each additional composition of the cell with the composed cells would increase the size of the final composition. This problem could be alleviated by careful design of cell A; however it is difficult for a cell designer to foresee all possibilities.

Composition systems that use the stretching method rely on leaf cells being properly designed with respect to geometric design rules. Locations of primitive elements for leaf cells in the Slap[Rowson 80] and Earl[Kingsley 82] systems are defined by user specified constraints. The external constraints are then extracted for the connectors and cell boundaries. These constraints are used to control the stretching of the cells. Constraint extraction has also been used with leaf cell compaction systems[Kingsley 84]. The Mulga[Weste 81] system uses a virtual grid compactor for leaf cell compaction. These compacted cells are then stretched at virtual grid boundaries for composition. The cells are designed so that there is one half of the maximum separation geometric design rule from the boundary to internal objects. This is done so that upon composition there will be no geometric design rule errors.

A system that combines the technique of abutment, stretching and routing is Rcomp[Mosteller 82]. The leaf cells for the system are designed in the Rest[Mosteller 81] stick system. The leaf cells are designed so that there is one half of the maximum separation geometric design rule at the cell boundaries to internal objects. Composed cells are thus geometric design rule correct. The composition performs a trade-off by area between adding a routing cell, or stretching the cells to be composed if they cannot be abutted. Language specification features allow customizing cells for a particular composition by a connector omit option. Also there is a feature to compose cells at closer than the half separation distance where the user takes the responsibility of design rules correctness. This system has been used for very large designs[Lien 81].

The problem with composition systems is that the optimization is done locally at the cell level and not globally for the whole composition. Even though leaf cells may be compactly designed, the resulting composition of the leaf cells could be quite loosely compacted. Cells are compacted first before any guiding information is given about the composition. For manually designed leaf cells the composition requirements should be considered. However for a reasonably large design, the composition constraints could be overbearing for manual designed leaf cells.

Past attempts at good composition systems is a reasonable starting point. The composition should optimize the integrated circuit at the global level where the compaction of individual leaf cell is directed from the global optimization.

2.2 Compaction

The compaction problem is minimizing the area, perimeter, wire length, or some linear combination of the above. The compaction process is the translation of a symbolic or physical representation of an integrated circuit cell to a physical cell representation which is design rule correct and has a near minimum area. The traditional compaction optimization criteria is that of minimizing area. At the present the best known compaction method is that of manual compaction. This process is onerous, error prone,and tedious but produces the best results. Manual compaction depends on the skill of the designer.

Early automatic and human directed compactors reduced the two dimensional problem into two separate compaction processes for the vertical and horizontal axes which are called 1–D compactor. These compactors use only orthogonal geometric boxes and limit the transistor and contacts to box symbols which can move only along the vertical or horizontal axis. The optimization criteria is minimum cell area. However with 1–D compactors, the optimization is minimum length for the axis to be compacted. The intention is that this method will reduce area. The next section will describe algorithms used for 1–D compaction.

Extensions to the 1–D compactors where the two compaction processes for each axis are coupled to make a single compactor have been called 2–D compaction. The next section will describe early attempts at 2–D compaction.

2.2.1 1–D Compaction

The first compaction method was the "shear line" approach first introduced in 1970 by Akers[Akers 70]. This approach operates on a fixed size grid in either of the horizontal or vertical axes by identifying space that could be removed perpendicular to an object as shown in Figure 2.2. The central idea is that a compression ridge of uniform width is identified across the cell where the ridge could be split at shear lines as shown. The area occupied by the compression ridge is then removed. The compaction proceeds in the horizontal axis until no space can be removed and then is repeated for the vertical. The first working program using the shear line method was developed by Dunlop[Dunlop 79, Dunlop 80]. The shear line method was extended to use a “node and line” representation. A limited jog insertion method was then introduced at connection points of wires and device. The drawback of this method is that it is computationally intensive.

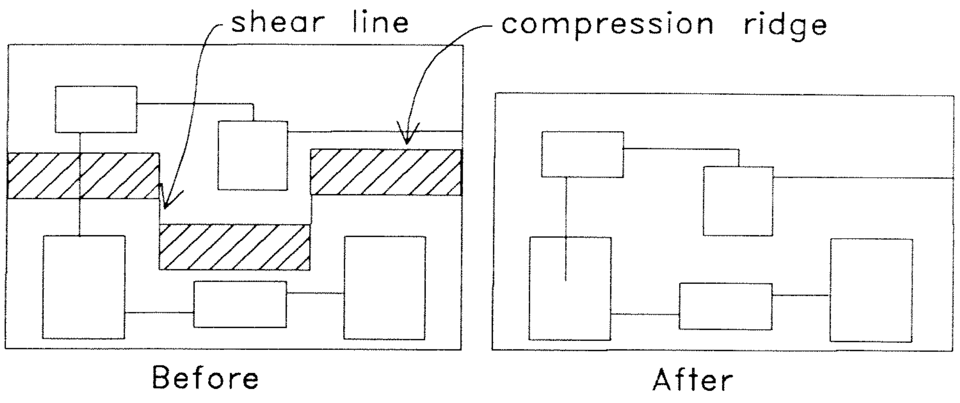


Figure 2.2 Shear Line Example

A second method for 1–D compaction is the directed graph model. A constraint graph is constructed and solved for each of the vertical and horizontal axes. This compaction method assigns *features*, which are groups of objects that move as units, to nodes in the graph. Each node will have a location, which is the placement of the feature along the compaction axis. The directed arcs of the graph represent the minimum geometric design rule constraint between nodes. An example is shown in Figure 2.3 for a vertical compaction. The graph is then solved by assigning placement position to the nodes in the graph.

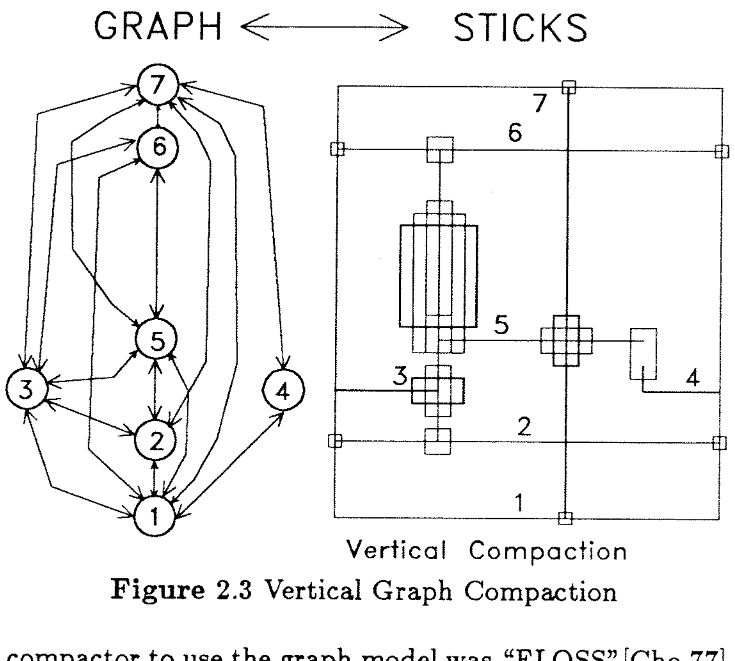


Figure 2.3 Vertical Graph Compaction

The first compactor to use the graph model was “FLOSS” [Cho 77] which solved the graph using a longest path approach. A formal presentation of the graph method was done by Hsueh[Hsueh 80] and implemented in the CABBAGE system. A method for automatic jog insertion was developed to further reduce the size of the cell. The jog insertion method did not give favorable results[Bales 82] and would eliminate the symmetry in symmetrical cells. Features in the graph that were not on the critical path of the graph were centered between the corresponding nodes which was called directed placement. Extension to the CABBAGE system were accomplished by Bales[Bales 82].

Most of the computation cost in the graph method is in the generation of the graph which is akin to performing a design rule check. The REST[Mosteller 81] system provides a shadowing method to reduce the computation time in construction of the graph and the number of nodes in the graph. This method reduced the checking of a feature to every other feature to only the directly covered feature. REST has no automatic jog insertion since it was found detrimental in previous systems. Instead REST couples a human with the compaction process for jog insertion. This method produced very compact cells over previous compactors.

REST is one of the first graph based compactors to be used for the construction of working chips. Previous compactors had not been used to build chips[Weste 81]. A few of the chips that were constructed with REST was a theorem proving chip[Lien 81], a motion detector[Tanner 83] and a quaternary multiplier[Frey 83]. REST proved to be an excellent tool for the construction of VLSI designs.

The first compactor to employ a wire length criteria into the cost function was REST. A wire affinity weight was added to the arcs in the graph and additional arcs were added where necessary between features that were directly connected. The graph was then solved as previously described by the critical path method. The nodes in the graph that had slack were then optimized based on the affinity weight. Each type of wire may have had a different weight. This innovation greatly improved the compaction. Other compactors that use a wire minimization criteria were developed later at VLSI Technology[Kingsley 84] and by Schiele[Schiele 83].

An interesting algorithm was developed by Maley[Maley 85] for automatic jog insertion. This method used the graph to place objects, leaving room for the wiring which is added later. A sketch was developed for the wiring of the cell that was used to construct the 1–D constraint graph which leaves room for the wiring. The graph was solved and the wires were then routed. The algorithm has not been implemented and as pointed out by Maley, it is somewhat unclear how it would perform in comparison to the previous methods. Also this method could increase the length of the wires since there is no criteria for wire length.

The Virtual Grid method which is used by MULGA[Weste 81, Weste 81a] places the components on a virtual grid which can be adjusted based on adjacency information. The virtual grid method compacts each axis separately. The compaction proceeds by assigning location to each virtual grid based on adjacency information which is a purely local operation. This method is fast because only local checking is necessary in the virtual grid. The drawback is that the size of the cell is not as good as one compacted with the graph based method. There is no jog insertion with this method. Although limited, this system has been used to create a large number of designs[Cho 85].

A necessary requirement in a compactor is the ability to direct the final shape of the compacted cell and the position of the interface connectors on the edge of the cell. This is necessary to compact cells for an environment for composition. One of the first compactors to allow the user to direct the shape of the final cell and to control the position of the connectors was REST. This was easily accomplished by adding user constraints as arcs in the compaction graph for connector position and final shape control. The user constraints could add cycles in the graph which possibly could create conflicting goals. When conflicting cycles in the graph are detected the user arcs causing the conflicting goals are deleted and the graph solving method proceeds.

The addition of user goals to the graph method is easily accomplished as previously described. For the shear line method and the virtual grid method, user defined goal would be difficult at best, if not impossible.

2.2.2 2–D Compaction

Early in the research for a two dimensional compaction method, Sastry and Parker[Sastry 82] showed that the orthogonal constraint based two dimension compaction was NP–complete[Garey 79].

The first 2–D compaction algorithm was developed at IBM by C.K. Wong[Wong 83a] which he also proved to be NP-complete. The model is composed of rectangles, and vertical and horizontal connecting wire segment rectangles. The method uses a set of base constraints which control the sizes of the boxes, the widths of the wire segments, and the relative location of the wire segments to there corresponding rectangles. Assume we have N rectangles, then there would be $O(N)$ base constraints. The overlap constraints are used to prevent two rectangles from overlapping where there are four constraints for each pair of rectangles. Only one constraint in the four needs to be satisfied. There are $O(N^2)$ overlap constraints. The initial position starts with the cell collapsed to a point. The heuristic algorithm proceeds by satisfying the invalid constrains. A pruning method is used to reduced the run time. There are some controls for the final results. There is no jog insertion. This algorithm has been applied to small test cases with about 17 elements. No real cells have been tried.

Another method for a 2–D compaction algorithm was developed by Watanabe[Watanabe 83] in the program Squash. This method couples the X and Y 1–D compaction graphs with a decision table where the decision variable determines whether an X or Y graph arc will be active. The coupling of the graphs limits the relation of two element as shown in Figure 2.4 where the constraints $d_x > C_x$ and $d_y > C_y$ are always valid. This limits the position of element A to be above and to the left of B. The constructed graphs are static in that if an element moves sufficiently away from the one that it is constrained too, the constraint is never removed from the graph. The algorithm uses a "branch and bound" method to solve the graph. Unlike the previous method the initial starting point is an exploded cell where all the constraints are valid.

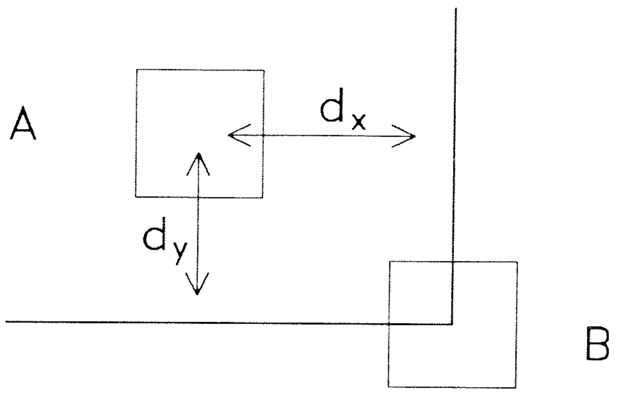


Figure 2.4 Two way symbol limit

Several techniques have been tried to improve the compaction. However they are used by iterating on the NP–complete algorithm. A method was devised to use the jog insertion algorithm of CABBAGE[Hsueh 80] which was never successful[Bales 82]. There were no examples of jog insertion so it is unclear if the jog insertion method was ever used. An iterative technique was used for the two symbol constraint limit.

Some control over the final shape and position of the connectors was provided. Several cells of approximately 25 transistors each were compacted with this method.

2.3 Limitations of Compactors

All of the current compactors are limited to rectangular geometry with compaction taking place along the horizontal or vertical axis. Each compactor considers the framework to be composed of fixed symbols which can move in the horizontal or vertical direction with interconnecting perpendicular wires. Forty five degree wires are not allowed in the framework. Wires are not allowed to bend nor are jog points inserted. This imposes a restriction on symbols that are connected by a straight wire by only allowing the connected symbols to move as a unit perpendicular to the wire. The orientation of the symbols is fixed. This framework severely limits the compaction space. The human designers is not limited to these artificial restrictions.

The desired optimization criteria for the 1–D compactors is that of area, however, the optimization criteria used is to minimize the width of the side being compacted. This criteria does not approximate the area. The 1–D compactors are extremely sensitive to the order in choosing axes axes. The X Y compaction does not permute. Compaction in one axis can block further compaction in the other axis.

The current 2–D compactors are extensions of the 1–D compactors and suffer from the same limitations with the addition of longer running time. The 1–D compactors such as REST[Mosteller 81] could easily compact the example of the 2–D compactor Squash[Watanabe 83] with approximately the same area or better. Further reduction would result if human directed jog points were used. The 1–D compactors are just as good as the early 2–D compactors.

Automatic jog insertion has been tried by several of the compactors with very limited results[Bales 82]. The human directed jog insertion in REST[Mosteller 81] has been used with very good results. Currently there are no effective automatic jog insertion algorithms.

Some of the compactors have a wire minimization criteria which is applied after the compaction has been completed. This possibly reduces the wire length of connecting elements that have slack. However this is only a sub-goal and applied after compaction. There is no real control over the lengths of wires.

An important property of a good compactor is the ability to direct the compaction of a cell for an environment. This control is used by some composition processes. The controls that are needed are to direct the shape of the cell and the connector position. The 1–D compactor REST which allows the shape and connector position to be directed was used by the composition system Riot[Trimberger 82]. Cabbage and Squash provided limited control.

To overcome the shortcomings of previous compactors we will use a curvilinear framework as described in Chapter 3. This framework does not have the artificial limitation of rectangular geometry and orthogonal wires. This framework allows models to rotate and deform to natural positions while older compactors use rigid non-rotating devices which have restricted movement. Jogs or wires wrapping obstacles is an inherent part of the frameworks. We do not need an ad hoc process to add jogs as previous compactors.

Previous compactors did not focus on a minimization criteria but tried to somehow minimize area by using the longest distance along each axis. We use a criteria where wire lengths, area, and composition constraints are uniformly addressed. Users can make a tradeoff between area, wire length and composition constraints.

Along with this framework we use the optimization technique know as simulated annealing, described in Chapter 6, to compact cells as well as a very good human designer or better. There is no longer a need to hand compact cells.

3 Structure of Primitives and Models

To apply simulated annealing to the two-dimensional circuit compaction problem requires a simple representation that can be modified easily. The variety of objects -transistors, contacts, resistors, wires, etc.- and sizes of objects found in VLSI circuits makes finding such a representation more difficult. Earlier applications of simulated annealing to molecular dynamics suggested the representation that we have developed.

In this chapter we define a curvilinear circuit description framework. This framework allows devices such as transistors and wires to be positioned at any angle. Wires can be considered as elastic bands tautly stretched from connection point to connection point passing as closely as possible around obstacles. Each device is not fixed to a specific shape or orientation but is a malleable entity. For example a long transistor may bend to conform to a region in a VLSI circuit where a smaller design would be possible over a design where the transistors are not pliable. The extent of the deformation of the device beyond the canonical form is limited by the electrical properties of the device and the geometric design rules.

In the design of the framework our objectives were to model the physical "real world" that a hand designer would use as close as possible and not be limited by artificial restrictions imposed by a compaction algorithm. The framework needs to be easily modified for user editing and for simulated annealing compaction. We wanted the representation of the design rules to be modular, easily altered. The representation for specific technology should be easily defined in a technology file.

One of the interesting points about silicon is that it does not know about X or Y as stated in a meeting with Ivan Sutherland in the early days of the Silicon Structures Project at Caltech. Early design methods employ constraints to the X and Y axes and fixed 90 degree device angles. Utilization of silicon area can be achieved by using curvilinear geometry.

The framework is composed of three geometric primitives that are designed for efficient manipulation. Transistors and other structures that form an integrated circuit chip are introduced as compositions of the primitive objects. These compositions are called models and define the archetype for transistors, contacts and other devices.

A design represented in the framework is maintained geometric design rule correct. All allowed modifications are to transform a correct design to another correct design.

The geometric design rules are defined for our framework. The primitive geometric design rules is a separation rule between objects. A geometric rule can be between objects on the same layer or between objects on different layers. The geometric design rules are formulated to reduce a multi-layer problem to several independent monochrome problems. In integrated circuit design the design rule that applies to electrically connected elements is different then the design rule for non-connected elements. We introduce the concept of "related" elements to provide for variation in design rules such as those based on connectivity. This method is also useful when defining models that are structures consisting of several elements connected and related in a specific manner.

The framework we present is defined so that a technology file can define the rules for our design. We will show how a nMOS technology file is constructed using our framework.

3.1 Background

Traditional integrated circuit designs use rectilinear geometry based on boxes and polygons with 90 degree and sometimes 45 degree angles. The reason for using rectilinear geometry is twofold. First, only simple calculations need to be performed for checking design rules. Second, early manufacturing processes were limited to boxes. With modern day fabrication technique this is no longer true.

The use of curvilinear geometry is not new and has been used by design systems as in Earl[Kingsley 82] and Pooh[Whitney 83]. The use of curvilinear geometry has been coined "Boston" geometry while orthogonal has been called "Manhattan", corresponding to the city streets where Boston has spaghetti-like streets and Manhattan has orthogonal streets.

An interesting study was done by Don Speck[Speck 85] at Caltech on savings of rectilinear versus curvilinear geometry. This study showed as much as 25% of cell area could be saved using curvilinear over rectilinear geometry. Speck found that to get the maximum savings the initial topology needed to be designed for curvilinear geometry although a substantial savings would result for cells designed for rectilinear geometry.

Curvilinear geometric objects have been used by contemporary designers in the design of the Mosaic[Lutz 84, Rabin 84] using the Earl system and in a datapath[Hedges 82] using Pooh. The Mosaic chip is working and will be used in a homogeneous machine. The cells in the Mosaic chip are extremely compact. Several of these cells will be used to demonstrate the ability of our system to compact.

An early curvilinear geometric design system "Coma"[Mosteller 82] had an extensive graphic editing system. The graphic part employed a virtual graphic interface with device independence. This system was intended to provide a framework for compaction. "Coma" data representation was in terms of wires and paths. Logical layers were used to describe design rules and stylized devices. During research of this system it was found that update and design rules checking was expensive. This system was abandoned for the framework presented in this dissertation.

A formalized representation of curvilinear geometry was presented in the "Pooh" system[Whitney 83]. Pooh provides an integrated approach of a uniform representation for circuit and layout representation. Entering data is through a layout language where placement is specified or through a layout graphic editor.

The framework we present here is restricted over the freedom allowed in Earl and in Pooh. Besides allowing curved wires Earl allows rectangular objects and free form polygons. Each object in Earl has only one layer. Pooh allows wire ends to be square or defined with some minimum radius.

3.2 Primitive Geometric Objects

We define our three primitive objects as a "Bubble", "Segment" and "Around" as shown in Figure 3.1. We will build VLSI chip structures from these primitives. For now we will consider the primitives to be single layered or monochrome. Each primitive has an associated N–tuple called the *rule* that contains properties about the primitive. One property is the radius of the primitive. The *rule* for a primitive is described in Section 3.3.2.

The primitive will first be presented as single layered objects. In coming sections we will extend the single layer concept to primitives existing on many layers.

The motivation for the type of primitives, and the structure and data fields of the primitives is as follows.

- The primitives meet the specific geometric design rules that are defined in a technology file. The cells constructed from the primitives is always design rule correct.
- We have a reduced data structure where related objects on many layers are represented by a single object on many layers. In past systems, geometric related objects were partitioned by layer. For example a contact was represented as three separate boxes on the polysilicon, metal and cut layer. We would like to combine several related objects that are on different layers into a single object on many layers. In our example the contact would be represented by a single object on the polysilicon, metal and cut layer. This method reduces the data structure and simplifies the modification of the data structure.
- We use the shortest path for wires interconnecting bubbles.
- Our structure is easy to manipulate. As you will notice the data structure of the primitives are not in a natural form but a form designed specifically for fast checking and manipulation. For example notice the segment definition is not defined by its end points but by its line equation.

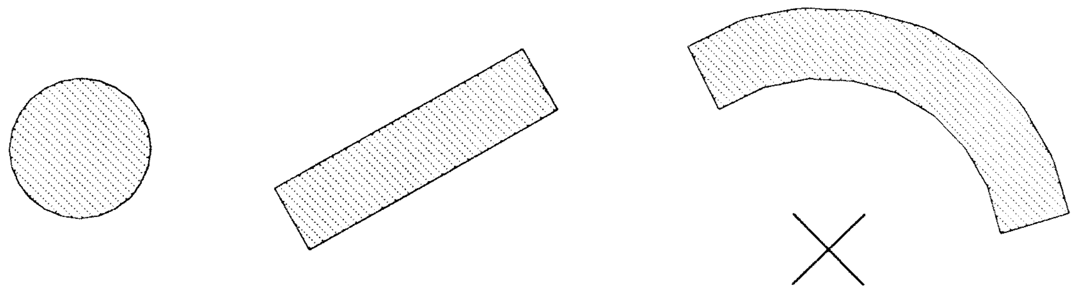


Figure 3.1 Bubble, Segment, and Around

There is a set of geometric inequalities that needs to be preserved in order to construct a working chip. These constraints are geometric design rules. There are geometric design rules for minimum widths, layer overlaps, layer extensions and minimum separations. We can classify the geometric design rules into two types where the first type affects the shape and size of wires and devices. The second affect the location of primitives. The first class of geometric design rules is preserved by construction consistent with definition in a technology file. The second type, which is the minimum separation rules is preserved by not creating nor moving a primitive to a location where the minimum separation rule would be violated.

For our primitives we will define the function MINW(O_1, O_2) as the minimum separation distance between the two primitives O_1 and O_2. The function MINW encapsulates the minimum separation geometric design rule. We define MINW as follows

$$\text{MINW}(O_1, O_2) = O_1.\mathit{rule} + O_2.\mathit{rule} + \mathit{mindistance}. \tag{3.1}$$

For our design the minimum separation distance between monochrome primitive objects is *mindistance*.

This function is a simplified version of the function for single layer primitives. In Section 3.3.3 on geometric design rules we will define MINW for the multilayer design. In the function (3.1) the *rule* can be considered the radius of the objects. MINW sums the radius of the first object with the radius of the second object plus the minimum separation distance. The *mindistance* is the minimum separation between two primitives.

For our framework we have two invariant conditions. The first invariant is that the design described by our framework will not violate the geometric design rules described in the technology file. The first class of geometric design rule is preserved by definition in the technology file. The second type, the minimum separation inequality rule, is preserved by not allowing any two primitives to be closer than MINW.

The second invariant condition is that a wire will always have the shortest path with a given topology. When wires are initially created they connect between two points in a straight line that is the shortest possible wire. Later when the wire is moved it will wrap objects which are as close as the geometric design rules allow.

The initial creation of primitives is accomplished by a graphic editor. We will show in the definition of the primitives how the checking for a geometric design rule violates is accomplished. If one is found, in the initial creation of a primitive, the creation is aborted in the editor.

3.2.1 Notation In The Plane

Before we define our primitives we will review the nomenclature and vector definitions that will be used in this dissertation. A point P will be defined as an order pair (x, y) of real numbers x and y,

$$P = (x, y) \tag{3.2}$$

where the point represents a position in the rectangular cartesian coordinates as shown in Figure 3.2. We will use the period to access a field of an N-tuple as in $P.x$.

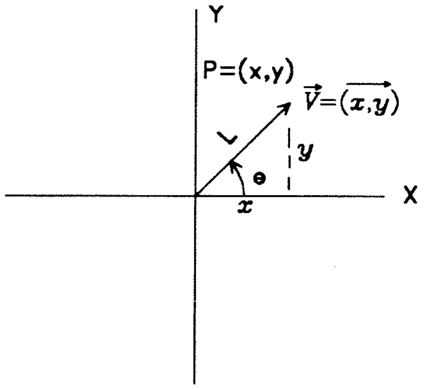


Figure 3.2 Point and Vector

We will consider a vector $\vec{V}$ associated with the point (x, y) to start at $(0,0)$ and end at (x, y). We write

$$\vec{V} = \overrightarrow{(x, y)} \tag{3.3}$$

as shown in Figure 3.2. Most operations on these vectors are carried out by considering each vector as a complex number and applying the ordinary arithmetic of complex numbers to the vectors. We will use the following notation to denote the components of a vector $\vec{V}$

$$\text{REAL}(\vec{V}) = x, \quad \text{and} \quad \text{IMG}(\vec{V}) = y.$$

The magnitude or length of vector $\vec{V}$ is denoted by

$$|\vec{V}| = |(x, y)| = L = \sqrt{x^2 + y^2}. \tag{3.4}$$

The angle θ of the vector $\vec{V}$ is defined from the x-axis to the vector as shown in Figure 3.2 and denoted $\Theta(\vec{V})$. Thus we have

$$x = L\ \cos(\theta) \tag{3.5}$$
$$y = L\ \sin(\theta) \tag{3.6}$$

We will use this property later on to test if vector $\overrightarrow{V}$ is in the top, bottom, left or right half plane.

Since the product of two vectors $\overrightarrow{V_1}$, and $\overrightarrow{V_2}$ is

$$\overrightarrow{V_1} * \overrightarrow{V_2} = (x_1, y_1) * (x_2, y_2) = (x_1 * x_2 - y_1 * y_2, x_1 * y_2 + x_2 * y_1), \tag{3.7}$$

we have their lengths $L_1 * L_2 = |\overrightarrow{V_1} * \overrightarrow{V_2}|$ and the sum of their angles $\theta_1 + \theta_2 = \Theta(\overrightarrow{V_1} * \overrightarrow{V_2})$.

We denote the conjugate of vector $\overrightarrow{V}$ as

$$\overline{\overrightarrow{V}} = \overline{\overrightarrow{(x, -y)}}, \tag{3.8}$$

and we have

$$\begin{aligned} |\overrightarrow{V_1} * \overline{\overrightarrow{V_2}}| &= L_1 * L_2 \text{ and} \\ \Theta(\overrightarrow{V_1} * \overline{\overrightarrow{V_2}}) &= \theta_1 - \theta_2. \end{aligned} \tag{3.9}$$

We will use this property later on in the definition of several algorithms.

3.2.2 Bubble Definition

A bubble is a circular object with a location defined by the center of the bubble with a radius defined by the bubbles ***rule***. Bubbles are used as connectors between wires and as terminator for wires ends. Wires emanating from a bubble will have a radius rule less than or equal to the bubble radius rule. A bubble can be considered as a solid object. We define a Bubble as a n-tuple:

$$\text{Bubble}(X, Y, \boldsymbol{segs}[*], \boldsymbol{arnds}[*], \boldsymbol{rule}, \boldsymbol{reln}[*]).$$

Bubble Attribute	Type	Attribute Description
X	Real	The x coordinate
Y	Real	The y coordinate
segs[*]	Array[†]	Attached segments
arnds[*]	Array	Wrapped arounds
rule	N-tuple	the Rule for this bubble
reln[*]	Integers	Related numbers

† An array is an n-tuple of objects of similar type.

Table 3.1 Bubble Attributes

3.2.2.1 Bubble to Bubble Geometric Design Rule Checking

We would like to know if two bubbles conform to a minimum separation distance. Consider two bubbles, B_1 and B_2 at the location $P_1 = (P_1.x, P_1.y)$ and $P_2 = (P_2.x, P_2.y)$ respectively. We can use the following inequality to determine if the two bubbles conform to a minimum separation design rule.

$$\mathrm{MINW}(B_1, B_2) \le \sqrt{(P_1.x - P_2.x)^2 + (P_1.y - P_2.y)^2}. \tag{3.10}$$

3.2.3 Wire Definition

A wire is a geometric path of constant width that starts and terminates at a bubble as shown in Figure 3.3. Wires are constructed from straight pieces called segments and curved pieces called arounds. A wire can be considered as a non-empty sequence of alternating segments and arounds that starts with a segment and ends with a segment. The wire is directed from a start bubble to an end bubble. The components of a wire are similarly directed.

The segments and arounds that construct a single wire have the same *rule*. Notice how the wire in Figure 3.3 wraps the two bubbles with the minimum separation distance rule.

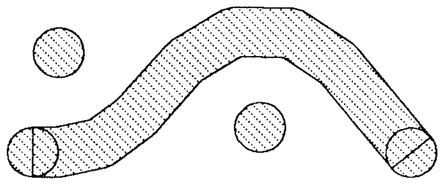


Figure 3.3 Wire

Wires were defined with two goals in mind - we wanted to be able to modify a wire or a portion of a wire very easily and we wanted to be able to treat the modification as a transaction with an add and delete pair that is used for compaction. The reason for the transaction is to be able to calculate very quickly the delta energy function that will be discussed in Chapter 6 on simulated annealing. We also need the ability to quickly check for a geometric design rule violation. One possible representation for a wire is a string of points. This representation was rejected because it is difficult to update a string with an add delete pair.

Each element of a wire is related to its neighbor by links. Elements of a wire can easily added or deleted by modifying links. Only the segments and arounds that modified the shape of the wire need to be considered. At the global level, we consider the framework to contain bubbles, segments and arounds. A wire is only an abstract quantity not specifically included in the framework. This allows modifying a wire by only examining segments and arounds. There is no need to consider the wire as an entity. The framework for a wire satisfies our first goal.

The second goal in considering the definition of a wire is that of easily checking for a geometric design rules violation. In the definition of segments and arounds in the next subsections the data structure carries the information needed for geometric design rules checking in a compact form.

We would also like our representation of segments and arounds to be consistent and easily generated from bubble position with minimal calculation. This consistency will be described In Section 3.2.3.2 on arounds. Because of this consistency, the calculation for representation of an around and segment group needs only to be done once and not for each piece.

The characteristic for wires is that all components of an individual wire will have the same rule. The bubbles at either end of each individual wire will have a radius greater than or equal to that of the wire radius. This simplifies the checking of geometric design rules for segments as described in the next section. Also if a segment is rotated about a bubble the geometric design rule checking of the segment end is not necessary. Notice in Figure 3.4 part A, if the wire is rotated the end attached to the bubble as shown could create a design rule error. Checking for this would be a particularly difficult special case. In case B this is not necessary since the end is not protruding.

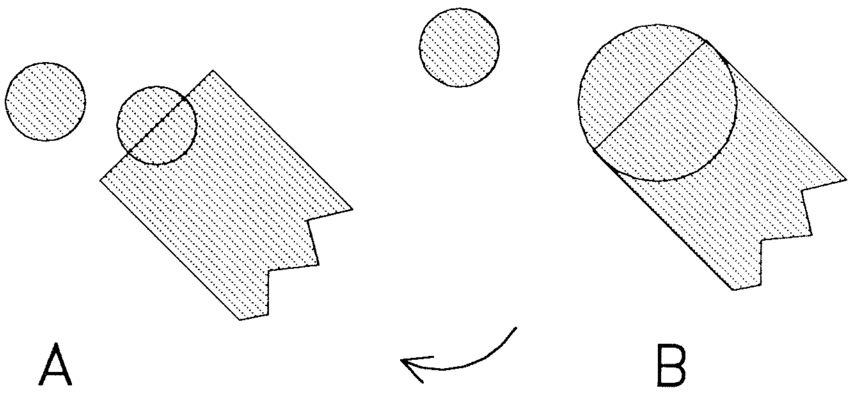


Figure 3.4 Wire-Bubble Interface

Wires are initially created by the graphic editor as straight lines segments between bubbles which is the shortest path for the wire. Arounds may be created when wires are modified by the move algorithm as described Chapter 4. The clearance between a around and the object that the around wraps will be the minimum separation rule. The wires will be maintained with the shortest path possible without violating a design rule.

3.2.3.1 Segment Definition

Our aim for the definition of a segment is to be able to quickly check the segment against other objects for design rule violations. We could represent a segment by its end points; however this representation would require creating the line equation of the segment for checking. The segment will be represented with a length, line equation, perpendicular line equation bisecting the segment at the center point. This representation allows simple checking that satisfies our goal.

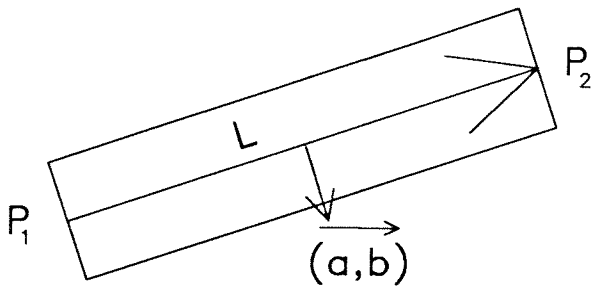


Figure 3.5 Segment

We define a segment S as shown in Figure 3.5 as directed from a point at location $P_1 = (P_1.x, P_1.y)$ to a point at location $P_2 = (P_2.x, P_2.y)$. We will represent a segment with a length, a normalized line equation, and a perpendicular normalized line equation bisecting the segment at the center point. The line equation is

$$a * x + b * y + c = 0 \tag{3.11}$$

and normalized so that

$$\sqrt{(a^2 + b^2)} = 1. \tag{3.12}$$

The vector $\overrightarrow{(a,b)}$ defined from (3.11) is a unit vector orthogonal to the segment as shown in Figure 3.5.

The normalized line equation perpendicular to the segment and intersecting the center is given by

$$a' * x + b' * y + nc. \tag{3.13}$$

We define a segment as an n-tuple

$$\text{Segment}(\mathit{starto}, \mathit{endo}, a, b, c, a', b', nc, L, \mathit{rule}, \mathit{reln}[*]).$$

Segment Attribute	Type	Attribute Description
starto	Tuple	The start of segment*
endo	Tuple	The end of segment*
(a,b)	Reals	Line equation vector
c	Real	Line equation constant
(a',b')	Reals	Line equation perpendicular vector
nc	Real	Line equation constant
L	Real	Line equation constant
rule	Tuple	The rule for this segment
reln[*]	Integers	Related numbers

* This tuple will be either a bubble or an around.

Table 3.2 Segment Attributes

This n-tuple does not represent the data structures used for implementation. It is presented to define the segment characteristics used throughout this thesis.

3.2.3.1.1 Segment Initialization

We will initialize the segment tuples fields based on bubble center points. We will first consider the case where the segment's end points are bubbles. The case where both or either end is an around will be covered in Section 3.2.3.2.1. The reason for this is that the calculation of the arounds fields also definite the segment fields. The length of the segment from the bubble at P_1 to the bubble at P_2 is

$$L = \sqrt{(P_2.x - P_1.x)^2 + (P_2.y - P_1.y)^2}. \tag{3.14}$$

If $L \neq 0$ then we set

$$\begin{aligned} a &= (P_2.y - P_1.y)/L, \\ b &= -(P_2.x - P_1.x)/L, \\ c &= -a * P_1.x - b * P_1.y. \end{aligned} \tag{3.15}$$

Otherwise we set

$$\begin{aligned} a &= 0.0, \\ b &= -1.0, \\ c &= P_1.y. \end{aligned} \tag{3.16}$$

The normalized perpendicular line equations constants are

$$\begin{aligned} a' &= -b, \\ b' &= a, \\ nc &= -a' * p_1.x - b' * p_1.y - L/2. \end{aligned} \tag{3.17}$$

3.2.3.1.2 Bubble to Segment Geometric Design Rule Checking

We wish to test if a bubble conforms to the minimum separation rule to the segment as shown in Figure 3.6. If the bubble B is in the region as shown in Figure 3.6, testing against the segment is necessary. If the bubble is outside the region, the testing would be accomplished when testing the primitive to what segment is attached too.

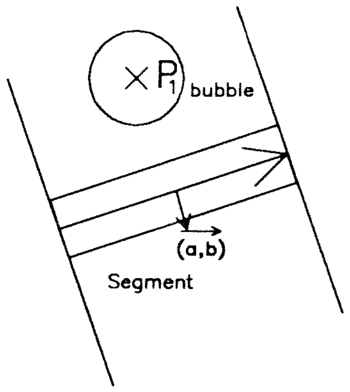


Figure 3.6 Segment Test

The normalized line equation has the following property: if it is evaluated at a point in the plain, the resulting value is the directed distance from that point to the line. The distance is positive if the point is on the outward pointing side of vector $\overrightarrow{(a,b)}$ otherwise negative.

We can test for a bubble being inside the region and being too close to the segment by using the perpendicular line equation and the length of the segment as follows. If we have

$$S.L/2.0 \geq |S.a' * P.x + S.b' * P.y + S.nc|, \tag{3.18}$$

Then the bubble is inside the region and we have

$$\text{MINW}(S, B) \; > \; |a * x + b * y + c| \tag{3.19}$$

then the bubble is too close to the segment.

3.2.3.2 Around Definition

The arounds are curved segments that wrap bubbles. Arounds are needed to represent the curved portion of a wire. The around connects to segments on both ends. Every around has a bubble at its center. Like segments, arounds have a direction from a start point to an end point as shown, in Figure 3.7. Every around starts and ends with a segment. The connected segments are tangent to the around but may have zero length.

Similar to our goal for the definition of a segment, we wanted the around representation to allow quick checking and be consistent with the segment representation. We considered representing the around with end points but rejected the idea due to complexities with geometric design rule checking.

Arounds are represented by a start unit vector $\vec{s}$, an end unit vector $\vec{e}$, a radius R and a center point that is a bubble. The radius R is positive if the arc is counter-clockwise and negative otherwise.

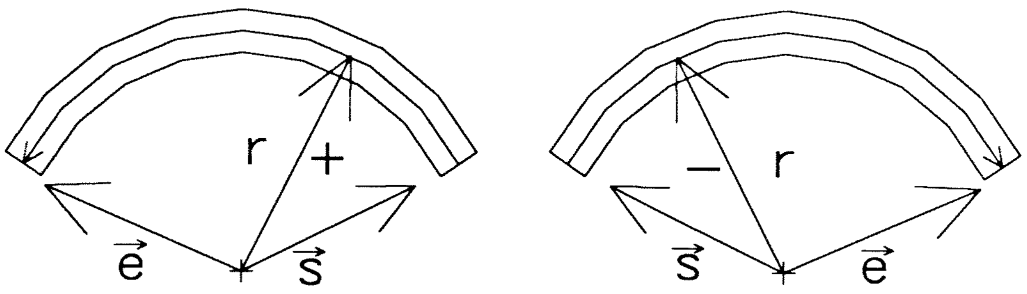


Figure 3.7 Arounds

Observed that the around unit vectors are related to the attached segment line equation vectors. Specifically, the segment line equation vector is equal to the attached around end vector for counter clockwise arounds and the negative otherwise. This representation provides a consistency between the segment and the around. The calculation for an around vector and segment vector is only done once for the pair.

The radius of the around is defined using the geometric separation design rule of the closest primitive that the around wraps. If the around, A, wraps a bubble, B, then the radius of the around, A, is R =MINW(A, B). When the around, A, wraps an around, Q, then the radius would be R =MINW$(A, Q) + |Q.R|$. Notice the arounds radius includes the radius of the around it wraps.

We define an around as a n-tuple:

$$\text{Around}(\boldsymbol{starts}, \boldsymbol{ends}, \boldsymbol{as}, \boldsymbol{bs}, \boldsymbol{ae}, \boldsymbol{be}, \boldsymbol{r}, \boldsymbol{b}, \boldsymbol{link}, \boldsymbol{rule}).$$

Around Attribute	Type	Attribute Description
startseg	N-tuple	The start segment
endseg	N-tuple	The end segment
$\overrightarrow{(as, bs)}$	Reals	Start vector
$\overrightarrow{(ae, be)}$	Reals	End vector
R	Real	Radius
B	N-tuple	Center bubble
link	N-tuple	Around Linked list
rule	N-tuple	The rule for this around

Table 3.3 Around Attributes

This n-tuple does not represent the data structures used for implementation. It is presented to define the segment characteristics used throughout this thesis.

All the arounds that would wrap bubble, B, are in a linked list with reference ***link***.

3.2.3.2.1 Around Initialization

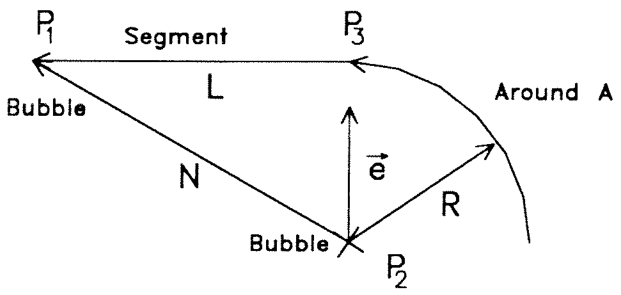


Figure 3.8 Counter Clockwise Around - Bubble End

We will first show how the around representation vector $\vec{e}$ is initialized when the connected segment ends at a bubble and starts at an around as shown in Figure 3.7. The radius R for the around is positive when the around is counter clockwise. We define the unit vector $\vec{e}$ in the direction of the vector $\overrightarrow{P_2P_3}$ as show in the Figure 3.8. The unit vector $\vec{e} = (ae, be)$ is constructed by rotating the normalized vector $\overrightarrow{P_2P_1}$ by the conjugate of a unit vector $\vec{V}$. The unit vector $\vec{V}$ is defined by the vector $\overrightarrow{P_2P_1} * \overrightarrow{P_2P_3}$.

$$\text{Let}\quad \begin{aligned} \Delta x &= P_2.x - P_1.x, \\ \Delta y &= P_2.y - P_1.y, \\ R &= \text{ radius of around}, \\ N &= \sqrt{\Delta x^2 + \Delta y^2}, \\ L &= \sqrt{N^2 - R^2}, \\ \vec{V} &= (|R|/N, R/|R| * L/N). \end{aligned} \tag{3.20}$$

Normalizing vector $\overrightarrow{P_2P_1}$ we get

$$\overrightarrow{P_2P_1} = (\Delta x/N, \Delta y/N). \tag{3.21}$$

Rotating normalized vector $\overrightarrow{P_2P_1}$ by the conjugate of vector $\vec{V}$ is

$$\begin{aligned} \vec{e} &= \overrightarrow{P_2P_1} * \overline{\vec{V}}, \\ \vec{e} &= (ae, be), \\ ae &= (\Delta x * R + R/|R| * \Delta y * L)/N^2, \\ be &= (\Delta y * R - R/|R| * \Delta x * L)/N^2. \end{aligned} \tag{3.22}$$

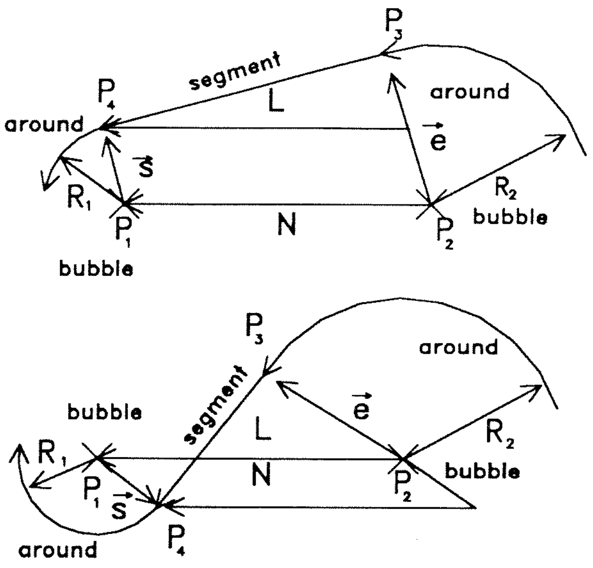


Figure 3.9 Counter Clockwise Around - Around End

We will now show how the around representation vector $\vec{e}$ is initialized when the connected segment ends at an around and starts at an around as shown in Figure 3.10. The radius for the around is positive. We can translate the problem to the previous by moving the line P_2P_1 by the radius of the second around R_2. This translation is done by letting $R' = R_1 - R_2$. We define the unit vector $\vec{e}$ in the direction of the vector $\overrightarrow{P_2P_3}$ as show in Figure 3.8. The unit vector $\vec{e} = (ae, be)$ is constructed by rotating the normalized vector $\overrightarrow{P_2P_1}$ by the conjugate of a unit vector $\vec{V}$. The unit vector $\vec{V}$ is defined by the vector $\overrightarrow{P_2P_1} * \overrightarrow{P_2P_3}$.

Let

$$\begin{aligned}
R_1 &= \text{ radius of first around,} \\
R_2 &= \text{ radius of second around,} \\
R' &= R_2 - R_1, \\
\Delta x &= P_2.x - P_1.x, \\
\Delta y &= P_2.y - P_1.y, \\
N &= \sqrt{\Delta x^2 + \Delta y^2}, \\
L &= \sqrt{N^2 - R'^2}, \\
\vec{V} &= (R'/N, R_2/|R_2| * L/N).
\end{aligned} \tag{3.23}$$

Normalizing vector $\overrightarrow{P_2P_1}$ we get

$$\overrightarrow{P_2P_1} = (\Delta x/N, \Delta y/N). \tag{3.24}$$

Rotating normalized vector $\overrightarrow{P_2P_1}$ by the conjugate of vector $\overrightarrow{V}$ is

$$\begin{aligned} \overrightarrow{e} &= \overrightarrow{P_2P_1} * \overline{\overrightarrow{V}}, \\ \overrightarrow{e} &= (ae, be), \\ ae &= (\Delta x * R' + R_2/|R_2| * \Delta y * L)/N^2, \\ be &= (\Delta y * R' - R_2/|R_2| * \Delta x * L)/N^2. \end{aligned} \tag{3.25}$$

In a similar manner the around representation unit vector $\overrightarrow{s} = (as, bs)$ may be found by remapping the variables and appropriate signs.

Now we will consider initializing the segment when either end of the segment is an around. First let us consider when the start of the segment is an around. Notice that in the around initialization the length of the segment is L and the around vector $-\overrightarrow{e} = (a, b)$ for a positive R or $\overrightarrow{e} = (a, b)$ for a negative R. Therefore we set the values for segment S to

$$\begin{aligned} &S.L = L, \\ &\text{if } R > 0.0 \text{ we set} \\ &\quad S.a = -ae, \\ &\quad S.b = -be. \\ &\text{Otherwise we set} \\ &\quad S.a = ae, \\ &\quad S.b = be. \\ &\text{fi} \\ &S.c = -S.a * P_2.x - S.b * P_2.y - R. \end{aligned} \tag{3.26}$$

Now let us consider when the end of the segment is an around. Notice that in the around initialization the length of the segment is L and the around vector $-\overrightarrow{s} = (a, b)$ for a positive R or $\overrightarrow{s} = (a, b)$ for a negative R. Therefore we set the value for segment S to

$$\begin{aligned} &S.L = L, \\ &\text{if } R > 0.0 \text{ we set} \\ &\quad S.a = -as, \\ &\quad S.b = -bs. \\ &\text{Otherwise we set} \\ &\quad S.a = as, \\ &\quad S.b = bs. \\ &\text{fi} \\ &S.c = -S.a * P_2.x - S.b * P_2.y - R. \end{aligned} \tag{3.27}$$

3.2.3.2.2 Bubble to Around Geometric Design Rule Checking

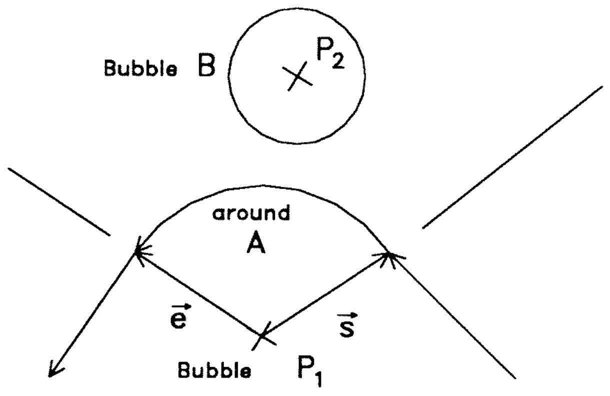


Figure 3.10 Around to Bubble Test

Now we will show how a bubble is checked for minimum separation distance violation to an around A of less than 180° as shown in Figure 3.10. If the bubble at P_2 is within the pie slice defined from the vectors $\vec{s}$ to $\vec{e}$ then the bubble will need to be checked for a minimum distance violation. The segment test would be invoked when the bubble is not within the pie slice. To see if bubble B at P_2 is within the pie slice we can use the vector $\overrightarrow{P_1P_2}$. The bubble is in the pie slice when the vector $\overrightarrow{P_1P_2}$ is in the left half plane of $\vec{s}$ and in the right half plane of $\vec{e}$ for positive R(counter clockwise around) or the vector $\overrightarrow{P_1P_2}$ is in the right half plane of $\vec{s}$ and in the left half plane of $\vec{e}$ for negative R(clockwise around). To see if vector $\overrightarrow{P_1P_2}$ is in the left half plane of $\vec{s}$ we multiply the vector $\overrightarrow{P_1P_2}$ by the conjugate of vector $\vec{s}$ then we take the imaginary part of the resultant vector and test for greater then zero. This is the sine of the angle between $\overrightarrow{P_1P_2}$ and $\vec{s}$ times the magnitude of vector $\overrightarrow{P_1P_2}$ that we will let equal a. If a is positive the vector $\overrightarrow{P_1P_2}$ is in the left half plane of $\vec{s}$. Similarly to see if vector $\overrightarrow{P_1P_2}$ is in the right half plane of $\vec{e}$ we multiply the vector $\vec{e}$ by the conjugate of $\overrightarrow{P_1P_2}$ then take the imaginary part of the resultant vector and test for greater than zero. This is the sine of the angle between $\overrightarrow{P_1P_2}$ and $\vec{e}$ times the magnitude of vector $\overrightarrow{P_1P_2}$ which we will let equal b. If b is positive, then vector $\overrightarrow{P_1P_2}$ is in the right halt of vector $\vec{e}$. We need to know if the angle between $\vec{s}$ and $\vec{e}$ is less than 180°. We will let U equal the sine between $\vec{s}$ and $\vec{e}$ which will be positive if the angle is less than 180° for positive R and negative for negative R. For an angle greater than 180° we would need to logically "OR" the previous test for the left plane of $\vec{s}$ with the test for the right half plane of $\vec{e}$.

We set

$$a = \mathrm{IMG}(\overrightarrow{P_1P_2} * \overline{\vec{s}}), \tag{3.28}$$

$$b = \mathrm{IMG}(\vec{e} * \overline{\overrightarrow{P_1P_2}}), \tag{3.29}$$

$$U = \mathrm{IMG}(\vec{e} * \overline{\vec{s}}) = \sin(\Theta(\vec{s}, \vec{e})). \tag{3.30}$$

If we have

$$\begin{array}{l}
(R \geq 0.0 \ \wedge \ \{\text{counter clockwise around}\} \\
\quad ((U \geq 0.0 \ \wedge \ a \geq 0.0 \ \wedge \ b \geq 0.0) \\
\qquad \vee \\
\quad (U < 0.0 \ \wedge \ (a \geq 0.0 \ \vee \ b \geq 0.0)))) \\
\vee \\
(R < 0.0 \ \wedge \ \{\text{clockwise around}\} \\
\quad ((U \leq 0.0 \ \wedge \ a \leq 0.0 \ \wedge \ b \leq 0.0) \\
\qquad \vee \\
\quad (U > 0.0 \ \wedge \ (a \leq 0.0 \ \vee \ b \leq 0.0)))),
\end{array} \tag{3.31}$$

then the bubble B is within the pie slice and we have

$$\mathrm{MINW}(B, A) \ \leq \ ||\overrightarrow{P_1P_2}| - |R|| \tag{3.32}$$

is true for the Bubble to be within the design rule.

3.3 Layers and Design Rules

The layer concept and geometric design rule representation will be presented in this section. Also the primitive *rule* that was introduced in Section 3.2 will be extended to its full meaning.

3.3.1 Layer Concept

VLSI integrated circuit designs are composed of many layers. In this dissertation each layer will be associated with a color. We will refer to layers as colors.

In the last section we defined the primitives for a single layer or monochrome color. This section will extend the definition of a primitive to many layers. Each primitive will have an associated n–tuple *rule*, that defines the layers for the primitive with the corresponding radii for the layers.

Geometric design rules for VLSI have two types of minimum distance separation relations. There are minimum separation distance rules between objects on the same layer and minimum separation distance rules for objects on different layers. For example there may be a rule between a red object and a green object with some minimum distance. We wish to reduce the interlayer interaction rules to single layer rules. This reduction is accomplished by introducing a new color for each interlayer interaction rule. The interlayer rule is formulated in the new color. For all objects with either layer, the new color will be added to the object and the width of the object for the new color will be the maximum width of the interacting layers for that object. The minimum distance for the color is the interlayer interaction rule. For example we let the color "blah" represent the interlayer rule for red and green. The color blah is added to all objects with colors red or green. The width of blah for the object is set to the maximum of the red or green width. The minimum distance for blah is the interlayer minimum distance for the red and green layers. Thus color blah reduces the red and green interlayer interaction rule to a monochrome rule of color "blah".

There exists a potential problem if the interlayer separation rule is greater that the normal separation layer rule. Two like bubbles with the normal layer rule and such an interlayer rule would be separated by the maximum separation rule. This unusual case is handled by ignoring the interlayer rule if the layer rule is present. The next section on geometric design rules will show how this is done.

Colors are used for many purposes besides fabrication in the VLSI design process. The color names in our system are defined by an array of color names, *colors*[*][1] where the number of color names is |*colors*|. Colors are numbered from 1 to N. We wish to know whether a color is relevant for a specific purpose. Each purpose has a Boolean array that is indexed by a color number. We can test for the relevance of a specific color in the Boolean array by ***arrayname***[***colornumber***] where ***arrayname*** is the name of the Boolean array. A logical operation on Boolean array is defined to be that operation on each of its corresponding elements. For example a logical "AND" on the arrays, A and B, resulting in C would be:

$$C[i] = A[i] \wedge B[i]$$

Each color may have several uses in the design process. For example a CIF color would be used for fabrication. The following table defines uses for global colors. The Boolean type is used to define the presence of a color in an application.

Color Attribute	Type	Attribute Description
Colors[*]	String	Color names
CIF[*]	Boolean	Fabrication
CIFname[*]	String	Fabrication
Mindistance[*]	Real	The minimum distance color

Table 3.4 Global Color Attributes

The real array ***mindistance***[*] is the minimum distance between two primitives with a specific color.

[1] We will consider an array as an n-tuple of similar type object represented by the symbol [*] where the index ranges from 1 to N where $|[*]| = N$.

3.3.2 Primitive Rule Definition

A *Rule* describes the attributes for a primitive. The geometric design rule properties for a primitive are given in the *rule*. Control information for the graphic editor is given. The *rule* describes if this rule may be used for a port. The colors that are displayed to the user in the graphic editor are also described. The *rules* for a specific technology are defined in a technology file.

We define the concept of a rule as introduced in Section 3.2 as a tuple:

```
rule(Name,Presentcolor[*],physical[*],
rulecolor[*],Rulectst[*],Ruleclear[*],
    connection[*],Edplace,Edport,Edwire,
    width[*],relsignificant[*, *],relwidth[*],
    COST)
```

Color Attribute	Type	Attribute Description
Name	String	Name of the rule
Presentcolors[*]	Boolean	Relevant Colors
Physical[*]	Boolean	Graphical Plotting and displaying
Rulecolor[*]	Boolean	Design rule colors
Rulectst[*]	Boolean	Design rule test colors
Ruleclear[*]	Boolean	Design rule clear colors
Width[*]	Real	The radius of the object
Connection[*]	Boolean	Allows connection to primitive
Edplace	Boolean	Allows creating a bubble
Edport	Boolean	Allows creating a port
Edwire	Boolean	Allows connection to primitive
Relsignificant[*, *]	Boolean	Related Significant Rule colors
Relwidth[*, *]	Real	Related width
COST	Real	Cost for a single step move

Table 3.5 Rule Attributes

The *rule* defines a set of properties of a primitive. We extend the number of colors for a rule to be the relevant colors as defined by the Boolean array *presentcolor*[*]. The Boolean array *physical*[*] defines the colors for the primitive that will be plotted and displayed during graphic editing. Not all colors would be displayed. An example would be the colors introduced for the interlayer interaction geometric design rule. The Boolean array *rulecolor*[*] defines the relevant colors for geometric design rules. The Boolean array *rulectst*[*] is used to test for ignore colors and the Boolean array *ruleclear*[*] is the colors to be ignored. The real array *width*[*] defines the radius of the primitive. The arrays *relsignificant*[*, *] and *relwidth*[*, *] will be discussed in Section 3.6 where concept of related is described.

The fields that control graphic editing are *connection*, *edplace*, *edwire*, and *edport*. The Boolean array *connection*[*] allows other primitives to connect to this primitive if the logical "AND" of their respective *connection*[*] has any relevant colors. *Edplace* is a Boolean that allows bubbles to be created with this rule. *Edwire* is a Boolean that allows the rule to be used for wiring. The cell created during graphic editing will have interfaces to the outside world called ports. *Edport* is a Boolean that controls the use of the rule for a port.

The actual names and number of colors are described in a technology description file. The section on nMOS will describe the technology file for nMOS.

3.3.3 Geometric Design Rules

Geometric design rules are a set of inequality constraints that are necessary for a working chip. Minimum allowable width, extensions, overlaps, and separations distances are geometric design rules. We will discuss how each is specified in our system. The designer of the technology file defines the rules for a specific technology he plans to use. We do not have the concept of a minimum width for a primitive but individual widths are specified for each kind of primitive. The width is defined for a primitive in the technology file within a rule. The widths may only vary by defining each desired width as a separate rule. The extension is the minimum distance a layer will extend over another layer. This extension is accomplished by defining the model(see Section 3.4) with a new bubble that is the extension layer with the proper separation constraint. The extension is also sometimes covered by the overlap method. The overlap is the minimum amount that a layer will extend over another layer. This rule can be accomplished by defining the width of the overlap layer equal to the width of the other layer plus the overlap. The separation constraints are defined between two objects where the sum of the two object widths plus a minimum distance is greater than or equal to the distance between the two objects. The constraints may be defined for objects on the same layer or different layers. In the last section we showed how to reduce the interlayer constraints to a monochrome constraint.

In our system the minimum separation rule between two primitives is calculated by the maximum of the separation distances of the relevant colors between the two primitives. The relevant colors are the intersection of the two primitives *rulecolor*[*]. The maximum separation distance is the maximum of the radius of the objects plus the minimum distance for each relevant colors.

With the separation constraints there are exceptions based on connectivity of the objects and other things. We will show how the exception is dealt with in this section and how it is defined in the related section. We call this exception "related". When the two objects are related we intersect the relevant colors as previously described with the union of the significant colors for each primitive. This allows each primitive to specify what colors are significant when it is related to another object. The calculation of maximum separation distance is accomplished as before but with the new relevant colors. In addition to the significant concept we allow the width to be a different value for related primitives.

There are cases when we wish to ignore a specific rule color when another is relevant. We will use the Boolean array *rulectst*[*] to test for this condition. If there are any relevant colors in the logical "and" of array $m[*]$ as previously described and *rulectst*$[*]$ then the colors in array *ruleclear*$[*]$ are set to false in array $m[*]$.

In our system the minimum separation design rule distance between any two primitive objects is defined by the function MINW(O_1, O_2) where O_1 is the first primitive object and O_2 is the second. If the function MINW equals zero then there is no design rule.

We will use a special function, "RELATED(O_1, O_2)" to indicate when there is an exception between O_1 & O_2 to the usual design rule and special handling is needed. The integer functions RELATEDINDEX$_1(O_1, O_2)$ and RELATEDINDEX$_2(O_1, O_2)$ will also be used with the exception. These three function will be defined in Section 3.6.

MINW will use the Boolean array $m[*]$ to indicate the relevant colors for the geometric design. Array $m[*]$ is set to the logical "and" of the first objects *rule.rulecolor*$[*]$ and the second objects *rule.rulecolor*$[*]$ for a non-related rule. The *rulecolor*$[*]$ is a Boolean array of relevant colors for geometric design rule. We now test for presence of any color in $m[*]$ with *rulectst*$[*]$ and clear with *ruleclear*$[*]$ for each object. If the array $m[*]$ does not have any relevant colors then there is no geometric design rule. We let the real array $W_1[*]$ equal the color widths for object one while $W_2[*]$ equals the color width for object two. Now we find the maximum geometric color rule for colors in array $m[*]$ with MINW$(O_1, O_2) = \text{MAX}\,(W_1[*] + W_2[*] + \mathit{mindistance}[*])$.

When there is a related exception, the Boolean array $m[*]$ is set to the logically "AND" of the significant Boolean arrays for each of the objects. The related significant defines the colors that are possible for a related rule. The rule color width of each object may be different than the standard in the case of related. The arrays $W_1[*]$, $W_2[*]$ are set equal to the related widths array. Now we proceed as before.

We precisely define the function MINW as follows:

```
Boolean procedure MINW($O_1$,$O_2$)
begin
    Real array(1 to *) $W_1$;
    Real array(1 to *) $W_2$;
    Boolean array(1 to *) m;
    real r;
    integer i;
    if RELATED($O_1$,$O_2$) then
    begin
        $W_1[*] \leftarrow O_1.rule.relwidth[\mathrm{RELATEDINDEX}_1(O_1,O_2),*]$;
        $W_2[*] \leftarrow O_2.rule.relwidth[\mathrm{RELATEDINDEX}_2(O_1,O_2),*]$;
        $m[*] \leftarrow O_1.rule.rulecolor[*] \wedge O_2.rule.rulecolor[*] \wedge$
            $(O_1.rule.relsignificant[\mathrm{Relatedindex}_1(O_1,O_2),*]$
            $\vee O_2.rule.relsignificant[\mathrm{Relatedindex}_2(O_1,O_2),*])$;
    end else
    begin
        $W_1[*] \leftarrow O_2.rule.width[*]$;
        $W_2[*] \leftarrow O_1.rule.width[*]$;
        $m[*] \leftarrow O_1.rule.rulecolor[*] \wedge O_2.rule.rulecolor[*]$;
    end
    if ANYON($m[*] \vee O_1.rule.ruletst$) then
        $m[*] \leftarrow m[*] \wedge$ NOT $O_1.rule.ruleclear$
    if ANYON($m[*] \vee O_2.rule.ruletst$) then
        $m[*] \leftarrow m[*] \wedge$ NOT $O_2.rule.ruleclear$
    $R \leftarrow 0$;
    for $i \leftarrow 1$ to $|colors[*]|$ do
        if $m[i]$ then
        $R \leftarrow \mathrm{MAX}(R, W_1[i] + W_2[i] + mindistance[i])$;
    return($R$);
end;
```

3.3.4 Invariant Conditions

The invariant condition for a design described by our framework is that the data structures will not violate the geometric design rules as described in the technology file. Initially when a design is created we start from an empty or null data structure. The invariant condition is preserved by definition for the null structure. During the editing of the design, objects are added or deleted to the data structure in transaction. Before the design is updated with the transaction the graphic editor checks the transaction with the design for the possible creation of a violation. If a violation would occur the transaction is rejected. This checking process will be described in the chapter on the graphic editor.

A design may be modified by moving a bubble to a new position. This modification is used by the graphic editor as an editor function for the user and by the compaction process. The next chapter will describe how the invariant condition is preserved for the bubble move process.

Geometric design rules are of two classes as previously described. The first class is preserved by definition. The second class, the minimum separation inequality rule, is preserved by not allowing any two primitive to be close than MINW.

Wires in our design will always have the shortest path for a given topology. When wires are initially created they connect between two points in a straight line which is the shortest possible wire. When a wire path is modified by either moving an end point or bending around an object the shortest path is maintained. A bent wire will wrap an obstacle as close as the geometric design rules allow.

3.4 Abstraction to Structures

We have shown in the previous section the primitive objects for our design. We will now show how they are used to construct transistors, contacts and other structures. A model for each type of transistor is defined in the rules file. An example of a nMOS enhancement transistor model is shown in Figure 3.11. On the left is a mask geometry realization of the device while on the right is a circuit representation. These models are the building blocks that will be instanced in the design by the graphic editor.

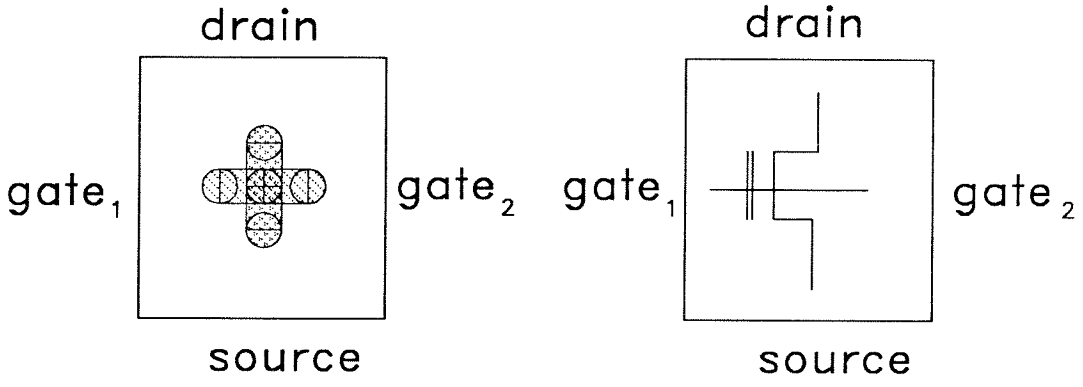


Figure 3.11 Enhancement Transistor Model Example

The purpose of the model is twofold. First the model provides a mechanism to consider a device such as a transistor as a unit that holds additional information not normally carried in the primitives. Functions such as graphic editing and simulation that treat devices as a unit will use the model structure.

The second purpose of the model is to allow the compactor and move algorithm to manipulate the model pieces, not the model as a whole. All previously known compactors as discussed in Chapter 2 do not allow transistors to move in pieces nor rotate. Moving the individual primitives for compaction eliminates any rectilinear constraints and allows the system to take full advantage of the allowed freedom of orientation on silicon.

Considering the devices as composed of pieces complicates the representation of design rules. This is the reason for the elegant method to model design rules.

Manipulating the pieces allows the model to move into regions that could not normally be reached by moving the model as a whole. We allow the device to distort during the move process so the device can "snake" its way into a tighter region during the compaction process. The coming chapters will show how a device, temporarily distorted in the compaction process, is pulled back into a valid shape, and how the amount of distortion is limited.

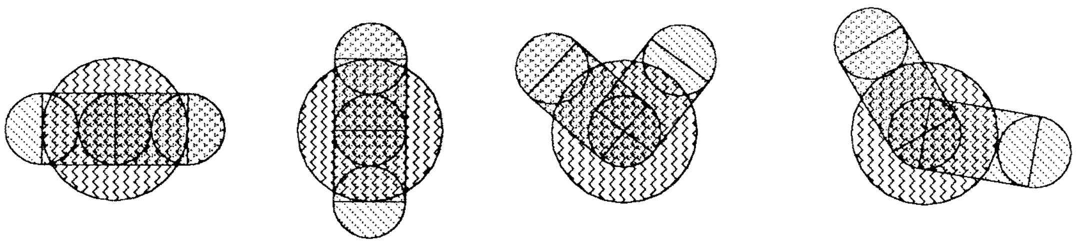


Figure 3.12 Buried Contact Configurations

Specific transistor or device may be of several configurations. For example, consider Figure 3.12 that has several valid configurations for a buried contact. If we only allowed one shape as previous compactors do, we would limit the choice of the compaction space. Our devices can take on a multitude of configurations where the configuration is governed by the geometric design rule mechanism. In Section 3.4.2 we will describe how the model dimensions are preserved while in Section 3.5.1 we will show how angles are considered.

An instance of the model is created for each desired copy of the model in the design. The primitives in an instance contain references to the primitives of the model. The model contains additional information that is not stored in the instance. Some of the information is about related number assignment that will be describe in the next section. Pictorially we can see this in Figure 3.13. Notice that model M_1 has two instances. In the design we may have several instances of transistors. The model is the archetype for the instances.

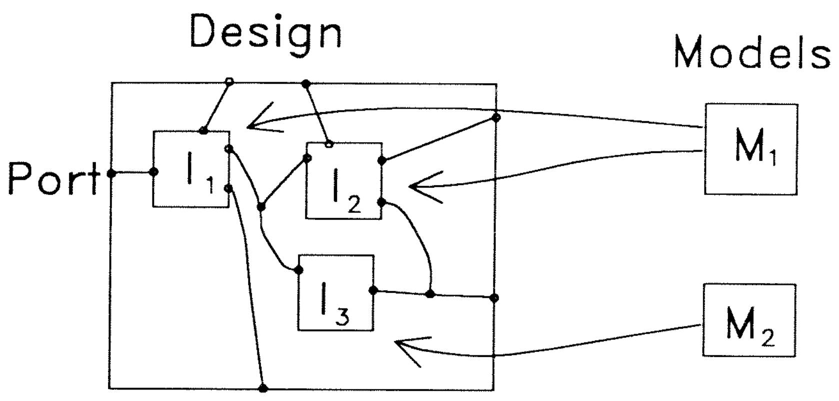


Figure 3.13 Instance Model Relation

Abstractly we can consider our design to contain instances of models, ports, wiring bubbles, and wiring between instances, ports and wiring bubbles. Ports are special bubbles that connect to the outside world. Our design is similar to a circuit schematic in that we have instances that represent devices, and wiring between instances that represent interconnect. In fact we could create a circuit schematic. The information for a interconnect list can easily be derived from our representation at this level. The simulator MOSSIM[Bryant 81] could be run with the information we have although we would need to add more data to the model for special controls to MOSSIM.

At the primitive level our design has no knowledge of transistor. We have only bubbles, segments and arounds. We can extract the instance and wiring level easily by following the links. This lets the modification to the design work at the primitive level without considering higher level structures.

3.4.1 "Model" Definition

A model is a collection of interconnected bubbles and segments that form a standard arrangement for the device that is sometimes referred to as the initial device. The model bubbles and segments are special in they contain additional fields in their n-tuples about the device, "related" information, and a link to the model N-tuple. We call these special primitives "Mbubble" and "Msegments". The model is an n-tuple with references to the elements of the model.

A model is used by creating an instance of the model at a location and orientation. An instance contains the same number of bubbles and segments as the model. The instance bubbles and segments are also special in that they contain an additional links to their corresponding primitives in the model. We call these "Ibubbles" and "Isegments".

There is one more special bubble that is used to interfaces the outside world that we call Ports. The Port has the additional fields of name which is the port name and side which is the side of the perimeter which the port resides.

The following is the list of special types of primitives that are used for models, instances, and ports. They contain all the fields of the generic primitive plus the additional fields as shown.

Mbubble(name,model,relindex[*]),
Ibubble(Mbubble),
Port(name,side).
Msegment(model,relindex[*]),
Isegment(Msegment).

The *relindex*[*] is information about related for a model that will be described in Section 3.6.

The model is an N-tuple of fields

Model(Name,Relnets[*],Relatedcolor[*],Relatedside[*],Pieces[*]),

where the fields are described in the following table.

Model Attribute	Type	Model Attribute Description
Name	String	Name of model
Relnets[*]	Integer	Related number
Relatedcolors[*, *]	Boolean	Related connect colors
Relatedside[*]	Integer	Side related
pieces[*]	Tuple	Primitives for model

Table 3.6 Model Attributes

The model is the archetype for the instances. The name field of the model is its name which is used by the graphic editor. The *relnets*[*] is an array which defines the related mapping number for the model which will be discussed in Section 3.6. The *relatedcolor*[*] and *related*[*] side will be discussed in Section 3.6. The *pieces*[*] is the array of tuples containing the primitives for this model. All models in the rules file are listed in the global array *models*[*].

3.4.2 Specifying Model Dimension

In some technologies it is necessary to control the length or width of a device. This control can be accomplished by two methods or a combination of both. The first method is to design the model to have the proper size bubbles as shown in Figure 3.14 part A for the length and width. Part B also uses this method to control the length while the second method is used to control the width.

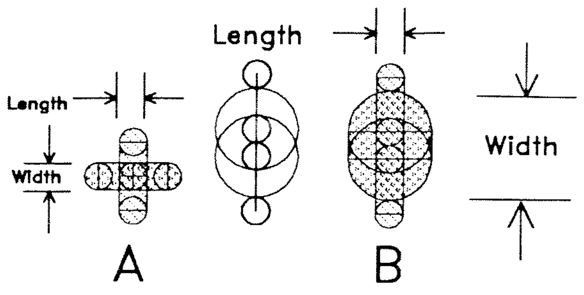


Figure 3.14 Enhancement Transistor

The second method is to create a "related" geometric design rule between the member bubbles in the model. This method is shown in the part B. We create an internal related exception between the member bubbles with the desired exception distance. The internal related exception means that this specific related property is only within the individual instance. This will prevent the two bubbles from getting any closer while preserving the length. The simulated annealing compaction will pull the bubbles together to this minimum distance.

The monochrome layer used for length exception may be a new layer for this purpose or an existing one may be used. We can define the related width for each individual bubble in the rule for the bubble by the ***rule.relwidth***$[*,*]$ and stating that the color is relevant by ***rule.relsignificant***$[*,*]$. The specific row of ***relwidth***$[*,*]$ and ***relsignificant***$[*,*]$ is defined by the position of the related numbers and the model bubbles ***relindex***$[*]$ field. An example of this level of control will be demonstrated in the example on nMOS models.

3.5 Modifying the Design

The design is modified incrementally by moving one bubble at a time. Large modifications are accomplished by moving many bubbles. Initially, devices and wires are entered in a valid state by the graphic editor. We can consider our design to have an energy state E for each configuration of our design. The definition for the energy E will be defined in Chapter 6 on Simulated Annealing. What is important to know at this time is that the energy E is a function of the wire length. The compactor minimizes the objective function E. The compactor modifies the design by moving bubbles first to a high energy state and then reduces the energy to a final configuration.

In the high energy state models are allowed to distort to a possible invalid shape although the invariant condition is preserved. As the system energy is reduced we would like the models to return to a valid shape. The control we use is that the energy E for a valid shape must be less than the energy E for an invalid shape.

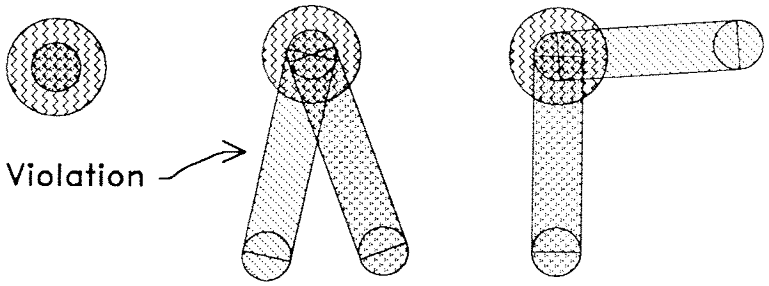


Figure 3.15 Invalid Buried Contact

Consider the nMOS buried contact in Figure 3.15 with the model on the left and a hypothetical portion of a design on the right. The connection to the model is directly to the contact itself. With this model there is no means to control the angle between the attached wires. As you can see this is a clear violation between the connected diffusion and polysilicon wires. If we keep the length of the attached segments constant the energy E is constant for any angle between the attached segments. We need a mechanism to control the angle between connected wires.

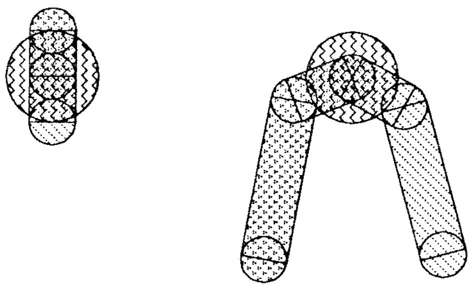


Figure 3.16 Valid Buried Contact

Consider the nMOS buried contact in Figure 3.16 with the model on the left and a hypothetical portion of a design on the right. Notice that the model has two fingers protruding from the contact. These fingers are the connection points. The geometric design rule between the bubbles on the fingers holds them apart. The related mechanism is used to allow the finger bubbles to be closer to the contact then normal geometric rules would allow. The normal rules do apply to the finger bubbles that hold them apart. This limits the angle between the segments attached to the buried contact proper. By using the related mechanism the distance between the finger bubbles can be controlled. This allows complete control between the angle of the two attached segments.

The energy necessary to move the finger bubbles away from the contact is costly compared to the cost of a normal segment. The relationship of model segment cost to normal segment cost will be discussed in Chapter 6 on simulated annealing. This preserves their shape at a low energy states. We now have a buried contact that preserve the angle design rules.

Another possible solution for angle control would be to add a test to see if the angle was exceeded. At first this seems the simplest. However this method limits the configuration space. Notice the configuration in Figure 3.15 is correctly represented in Figure 3.16. If we had a true angle constraint this configuration would not be possible without adding intermediate bubbles. In addition we would limit the compaction space.

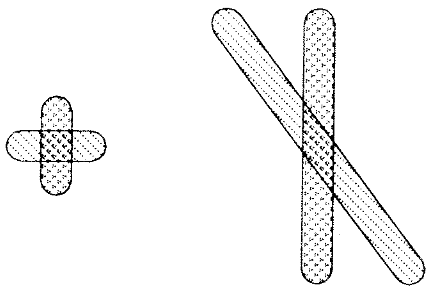


Figure 3.17 Invalid Enhancement Transistor

Now let us look at a possible model for an nMOS enhancement transistor as shown in Figure 3.17 on the left and an use of it on the right. The model is composed of a single bubble with a red, green and gate layer. Now look at the use of the model that has a acute angle. This may or may not be a violation depending on the design. This will affect the transistor behavior since the gate area has been increased. We would like the minimum angle between the polysilicon and diffusion to be close to 90° as possible.

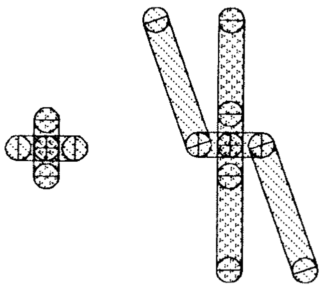


Figure 3.18 Valid Enhancement Transistor

Now consider the model in Figure 3.18 that has added finger bubbles. The finger bubbles would tent to hold the model in an orthogonal state for low energy states which is our desired goal. Notice the use of the model on the right that has an equivalent positions as in Figure 3.17 however it does not have the previous problems.

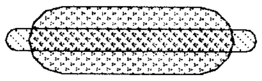


Figure 3.19 Wide Enhancement Transistor

We would like to be able create a wide transistor or device that is allowed to snake or bend as shown in Figure 3.19. The model is created by adding several intermediate bubbles that are the bending point as in Figure 3.20. The number of bends can be controlled by number of bubbles in the device. The length can be controlled as described previously with the related exception. The total length of the transistor is the sum of distances between the individual bubbles as shown. All we need to do to control the total length is to control the pieces.

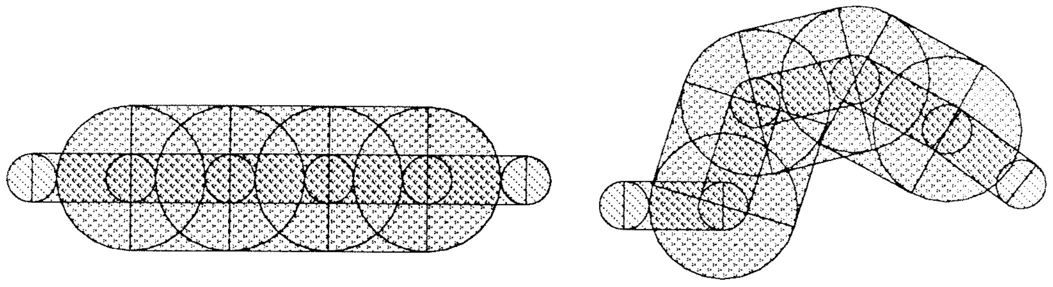


Figure 3.20 Snaky Enhancement Transistor

Now we may wish to control the amount of bending in the snaky device. This control can easily be accomplished by adding another length rule to limit the distance between alternate bubbles in the device. This minimum distance would limit the angle of the segments attached to the bubble between the alternate bubbles. Assume L is the minimum distance between bubbles in the device. Then the new rule should be no greater than $2 * L$ which would limit the angle to 180°. Suppose we wished to limit the angle to 90°. Then we would let the new rule be $L * \sqrt{2.0}$.

The new distance rule is created by adding a new monochrome layer for this purpose. We will use the related exception handling capability to exercise the new rule. The ***rule.width*** and mindistance should be set to zero so non–related rules do not apply. We let alternate bubbles ***rule.relwidth**[**model related**, *]* for the new monochrome layer be equal to one half of the desired distance rule. The rule.relsignificant[model related, *] should be also set relevant for the alternate bubbles. The device should have an internal related number which will be use to state that the alternate bubbles are related.

3.5.1 Preserving Angle Constraints

From the previous examples with angle problems and their solutions we can present a general mechanism to control angles within a model. First we need to know the minimum allowed angle θ between the two segments that we wish to limit as shown in Figure 3.21. The bubbles B_1, B_3 at the ends must be part of the model which can easily be added if they are not. The distance L_1 is the minimum geometric design rule between bubbles B_1 and B_3. Likewise L_2 is the minimum geometric design rule between B_1 and B_2. We now create a geometric design rule of distance D between bubble B_3 and B_2 if there is not one by using the exception handling method. The distance D is defined by

$$D = \sqrt{L_1^2 + L_2^2 - 2 * L_1 * L_2 * \cos(\theta)}.$$

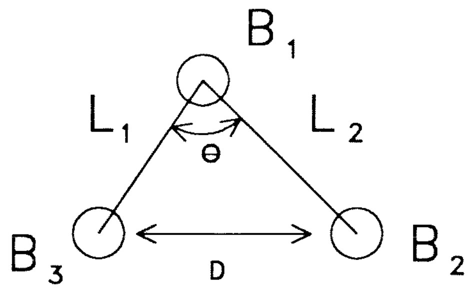


Figure 3.21 Angle Control

Now let us apply the method to an example. In the Figure 3.22 we wish to limit angle between the segment from B_2B_3 and B_2B_1 to be greater than or equal to 135° and less than or equal to 225°. Using the method we would add a geometric design rule with the exception handling mechanism between bubble B_1 and B_3. The distance between B_1 and B_2 is 4λ. The distance between B_2 and B_3 is 4λ. The minimum distance between B_1 and B_3 would be 7.4λ. We could add a new color for this rule and give each bubble one half of the distance with the minimum separation of zero. We use the "related" mechanism to confine the exception rule to these two bubbles. In a similar manner the other end may be controller.

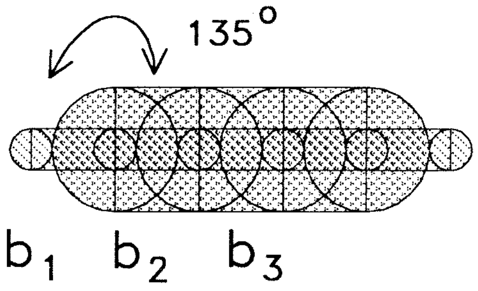


Figure 3.22 End Bubble Angle Control

3.6 Concept of "Related" Primitives

Let us first review how the geometric design rule between two bubbles or objects is determined. The geometric design rule is the maximum of the sum of each object's widths, plus minimum separation for each relevant color, where the relevant colors is the intersection of the objects rule colors. The objects may have none, one or more colors in common. If there are no colors in common there is no geometric design rule between the two objects.

Within our framework we will have bubbles that are connected by wires and not connected as shown in Figure 3.23. In the top of Figure 3.23 there are two bubbles that are not connected. In this case we would like the normal geometric design rules to apply. In the bottom of the figure we have the same two bubbles connected by a wire. In this exception case we would like the bubbles not to have a geometric design rule. We could accomplish this by marking each bubble that is in common with a unique integer. Then if two bubbles had the same integer we would say there is no geometric design rule between them.

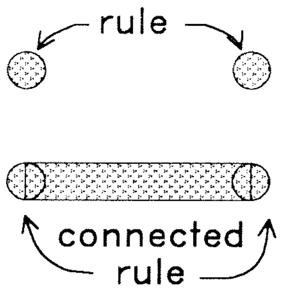


Figure 3.23 Unconnected and Connected Bubbles

We could extend this method by using a separate width and color set for each object rule for cases where the object is connected to another object. When two objects are connected we would use the separate width and color sets for each object. This would allow altering the geometric rule between the two connect objects. In our example we would use the normal rules for the unconnected bubbles and the separate width and color sets for the exception when they are connected.

In the nMOS enhancement transistor described in Section 3.4 we have several bubbles that have exception rules. Our new method for using a separate width and color set for the member bubbles would give the desired affect.

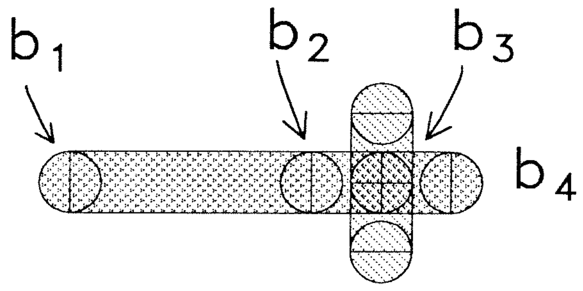


Figure 3.24 Bubbles and Transistor Connection

With this method we would have difficulty with a bubble connected to the transistor as shown in Figure 3.24. Bubbles b_1 and b_2 have the same colors and widths while b_3 has the colors of the previous bubbles plus a gate type color. We would like the design rule between bubble b_1 to b_2, to be zero and the design rule between bubble b_2 to b_3, and b_1 to b_3 to be the desired distance for the model. For the connected bubbles, b_1 and b_2, we need the separate color set to be null for the geometric design rule to be zero. However, then we could have no rule between bubbles b_2 and b_3 or between bubbles b_1 and b_3.

We could add an extra color for the purpose of holding the bubble b_2 away from bubble b_3. However if we had two of the models connected by a wire between bubbles b_4 to b_2 the models could not come as close as desired because of the extra color. This method does not function properly for bubbles connected to models.

Instead of using an alternate rule we will use an alternate set of widths as before and a set of "significant" colors. A significant color is one that is significant in the rule definition for a connected object. We will define the relevant colors for the connected rule as the intersection of the object rules colors intersected with the union of the significant colors. Assume *rcolor* is the rule color and *scolor* is the significant color. Then

$$(rcolor_1 \wedge rcolor_2) \wedge (scolor_1 \vee scolor_2)$$

would be the relevant colors for two connect objects. Notice that if a color, say green, is present in both $rcolor_1$ and $rcolor_2$ than either *scolor* may be present to specify green as a relevant color.

Applying the method of "significant" color to the case in Figure 3.24 bubble b_3 would have a significant color and bubble b_1 and b_2 would not. We would not have a rule between bubbles b_1 and b_2 because there are no significant colors. However there would be a rule between bubbles b_1 to b_3 and b_2 to b_3 because bubble b_3 has a significant color.

In the models we would like to be able to add a color that is only relevant when the objects are connected. If we alter the relevant expression as follows

$$\begin{array}{l}((rcolor_1 \vee scolor_1) \wedge (rcolor_2 \vee scolor_2)) \\ \qquad\qquad \wedge \\ (scolor_1 \vee scolor_2)\end{array}$$

so that significant colors is the union with the rule color, we can introduce a color that is not normally a rule color. This allows us to add internal colors to a model. We now have a general method to define geometric design rules for connected objects.

Notice in Figure 3.24 that bubble b_4 is connected to b_2 through b_3 but is not electrically in common with b_2. We would like b_2 to be connected to b_4 so that they can be separated by a distance less than a normal design rule. Any non-model bubbles connected to bubble b_4, we would consider connected to b_3, and b_2 but not b_1. If bubble b_1 was connected to bubbles beyond bubble b_4 they could generate design rules errors in the compaction process.

We can use the general method of "related" to solve the previously described problem. We will let each wiring object have a single unique related number where any reachable wire and wiring bubbles from the wire without going through a model will also have the same number. A model object will have a set of related numbers. Two objects are related when they have an equal related number.

This solves the previous problem by having bubbles b_1, b_2, b_3 and b_4 related where bubbles connected to bubble b_4 and not in the model will not be related to bubble b_1. Then we would have a separate related number for b_2, b_3 and b_4 and for objects connect to b_4 and not in the model.

We will extend our method of "related" by allowing each model object to have a width and significant set for each unique related number. This allows an alternate rule for each kind of related.

How are related numbers defined? We can't anticipate all the possible combinations of related number assignment. We will let the definition of the model in the rules file define the equivalence mapping for related. We now have a general method for handling geometric design rule exceptions.

3.6.1 "Related" Definition

The "related" mechanism provides exception handling capability in determining geometric design rules as previously described. Two primitives are related when they have an equal non-zero related number in array "reln". All primitives except for arounds have reln number arrays. Arounds use the start segment for testing for related numbers. This is done because each primitive of a wire will have the same related number. Instance primitives may have a set of related numbers while all others will have one. Recall from the section on abstraction to structures that we can consider our design to have wires, wiring bubbles and instances. Wires and wiring bubbles will have a single reln number while an instance may have many. The reason for the array of reln numbers is that instance will have many more related exceptions then wires.

Related differs from connection in that connection refers to a Netlist type of interconnection structure while related is used for design rules exception handling. Connection defines how devices are connected. Generally devices like contacts do not pass related information for different layers while the different layers are electrically in common.

The function related(O_1,O_2) set the temporary arrays N_1 and N_2 to the respective reln arrays for object one and two. Then the function looks for a non-zero match in the two arrays. We precisely define the function related as follows.

```
**Boolean function** RELATED($O_1$,$O_2$)
**begin**
    **Integer array(1 to** $*$**)** $N_1$, $N_2$;
    **Integer** i, j;
    $N_1[*] \leftarrow$**if** $O_1$ **is** around **then** $O_1$.*starts.reln*$[*]$ **else** $O_1$.reln$[*]$;
    $N_2[*] \leftarrow$**if** $O_2$ **is** around **then** $O_2$.*starts.reln*$[*]$ **else** $O_2$.reln$[*]$;
    **for** $i \leftarrow 1$ **to MAX**$(|N_1[*]|, N_2[*]|)$ **do**
    **for** $j \leftarrow 1$ **to** $i$ **do**
    **begin**
        **if** $N_1[i] = N_2[j] \land N_1[i] \neq 0$ **then**
        **return**(TRUE);
    **end**
    **return**(FALSE);
**end**;
```

In a similar manner function relatedindex$_1$(O_1,O_2) set the temporary arrays N_1 and N_2 to the respective reln arrays for object one and two. However, it first check O_1 to see if it is an instance type. Otherwise a 1 is returned. Then the function looks for a non-zero match between the two arrays. If a match is found the relindex of the model object is return. We precisely define the function as follows.

```
integer function RELATEDINDEX$_1(O_1, O_2)$
begin
    Integer array(1 to *) N$_1$, N$_2$;
    Integer i, j;
    if $O_1$ is around then
        if not $O_1$.starts is Instance then return(1)
    else
        if not $O_1$ is Instance then return(1);
    if $O_1$ is around then $N_1[*] \leftarrow O_1$.starts.reln$[*]$
    else $N_1[*] \leftarrow O_1$.reln$[*]$;
    if $O_2$ is around then $N_2[*] \leftarrow O_2$.starts.reln$[*]$
    else $N_2[*] \leftarrow O_2$.reln$[*]$;
    for $i \leftarrow 1$ to MAX$(|N_1[*]|, N_2[*]|)$ do
    for $j \leftarrow 1$ to $i$ do
    begin
        if $N_1[i] = N_2[j] \wedge N_1[i] \neq 0$ then
        return($O_1$.MP.RELINDEX$[i]$);
    end;
    return(1);
end;
```

In a similar manner function relatedindex$_2(O_1, O_2)$ returns the relindex except for object two. We precisely define the function as follows.

```
Integer function RELATEDINDEX$_2(O_1, O_2)$
begin
    Integer array(1 to *) N$_1$, N$_2$;
    Integer i, j;
    if $O_2$ is around then
        if not $O_2$.starts is Instance then return(1)
    else
        if not $O_1$ is Instance then return(1);
    if $O_1$ is around then $N_1[*] \leftarrow O_1$.starts.reln$[*]$
    else $N_1[*] \leftarrow O_1$.reln$[*]$;
    if $O_2$ is around then $N_2[*] \leftarrow O_2$.starts.reln$[*]$
    else $N_2[*] \leftarrow O_2$.reln$[*]$;
    for $i \leftarrow 1$ to MAX$(|N_1[*]|, N_2[*]|)$ do
    for $j \leftarrow 1$ to $i$ do
    begin
        if $N_1[i] = N_2[j] \wedge N_1[i] \neq 0$ then
        return($O_2$.MOBJECT.RELINDEX$[i]$);
    end
    return(1);
end;
```

3.6.2 "Related" Number Assignment

We will now describe how the related numbers are assigned to our design from the technology file. Consider our design as wires, wiring bubbles, and instances. Individual wires will have the same related number. All paths that can be followed along a wire through any wiring bubbles will have the same related number. The model defines the equivalence mapping of the related number for the instance. The model has related numbers that have been defined in the technology file that describe how the instances and connecting wires related numbers are assigned.

The model.relnets[*] is the array of related numbers for the model that define the equivalence mapping in the instances. Each primitive of the model may have one or more of these related numbers. Ibubbles connecting to Nsegments may pass one related number since Nsegments are normal wires. The model.relatedcolors[*, *] define a Boolean array of colors for each specific model related number. When there is a relevant color in relatedcolors[i, *] and in the presentcolor[*] an equivalence is performed for related index i. For an Ibubble to define an equivalence between one of its related numbers and a related number of an Nsegment the following array must have at least one true element for the model related number index by i.

$$\text{Nsegment.rule.presentcolor}[*] \wedge \text{model.relatedcolor}[i, *]$$

For example consider the following nMOS enhancement transistor in Figure 3.25. The model would have 3 related numbers consisting of 1 for color red, and 2 and 3 for green. In the skeleton in Figure 3.25, the model primitives are numbered with the related number. Notice the ordering of the related numbers on the bubbles, that is the order in the array in the model primitive. The equivalence of an Nsegment to a Ibubble will take place on the first valid related number. If an Nsegment connects to the top or bottom it would be equivalent to one. If a green Nsegment connects on the left it would be equivalent to 2 while on the right it would be equivalent to 3.

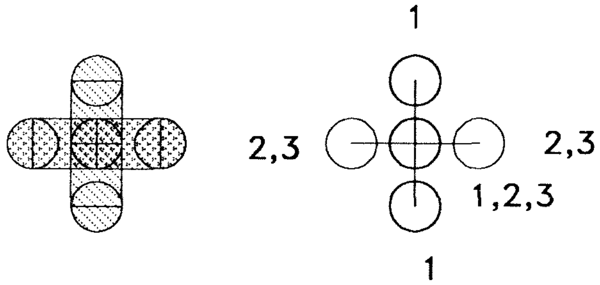


Figure 3.25 Reln Enhancement Transistor

We can define an internal model related number by not having any relevant colors for the related number in array relatedcolor[*, *].

The related side further refines the equivalence mapping. A side of zero allows connection to any side. A side of one allows a mapping if the Nsegment $\overrightarrow{S_t}$ connects between the Isegment $\overrightarrow{S_1}$ and $\overrightarrow{S_2}$ as shown in Figure 3.26. The instance segment order in the instance bubble array, segs[*] is preserved from the definition in the model. Therefore the model defines segment one and segment two. A side of two allows a mapping if the Nsegment $\overrightarrow{S_t}$ connect between the Isegment $\overrightarrow{S_1}$ and $\overrightarrow{S_2}$ as shown.

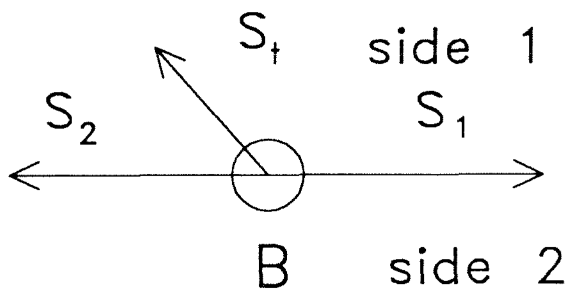


Figure 3.26 Related Side Definition

For an angle of less than or equal to 180° between $\overrightarrow{S_1}$ and $\overrightarrow{S_2}$, we can easily test for an Nsegment, $\overrightarrow{S_t}$, on side one by testing for a positive sine between Isegment $\overrightarrow{S_1}$ and $\overrightarrow{S_t}$, and a positive sine between $\overrightarrow{S_t}$ and $\overrightarrow{S_2}$. For an angle between $\overrightarrow{S_1}$ and $\overrightarrow{S_2}$ of greater than 180° we would do a logical "or" instead of a logical "and".

We will define the function testside(B,NS,z) that will return true if the Nsegment, NS, is on side Z of Ibubble B. We will let the scratch variables x_1, y_1 equal the components of a vector pointing along segment S_1. Similarly we will let the scratch variables x_2, y_2 equal the components of a vector pointing along segment S_2. Similarly we will let the scratch variables x_t, y_t equal the components of a vector pointing along segment S_t. We will let scratch variable u equal the sine between $\overrightarrow{S_1}$ and $\overrightarrow{S_2}$ by taking the imaginary component of the vector $\overrightarrow{S_2}$ multiplied by the conjugate of $\overrightarrow{S_1}$. Similarly we will let scratch variable a equal the sine between $\overrightarrow{S_t}$ and $\overrightarrow{S_t}$ and b equal the sine between $\overrightarrow{S_t}$ and $\overrightarrow{S_t}$. We precisely define the function as follows.

```
Boolean function testside(IB,NS, $s$)
begin
    real $x_1, y_1,\ x_2, y_2,\ x_t, y_t$;
    real $U, a, b$;
    if IB.$segs$[1].$starto$ =IB then
        $x_1 \leftarrow -$IB.$segs$[1].$b$, $y_1 \leftarrow$IB.$segs$[1].$a$
    else
        $x_1 \leftarrow$IB.$segs$[1].$b$, $y_1 \leftarrow -$IB.$segs$[1].$a$;
    if IB.$segs$[1].$starto$ =IB then
        $x_2 \leftarrow -$IB.$segs$[2].$b$, $y_2 \leftarrow$IB.$segs$[2].$a$
    else
        $x_2 \leftarrow$IB.$segs$[2].$b$, $y_2 \leftarrow -$IB.$segs$[2].$a$;
    if NS=IB then
        $x_t \leftarrow -$NS.$b$, $y_t \leftarrow$NS.$a$
    else
        $x_t \leftarrow$NS.$b$, $y_t \leftarrow -$NS.$a$;
    $U \leftarrow -y_1 * x_2 + y_2 * x_1$;
    $a \leftarrow -y_1 * x_t + y_t * x_1$;
    $b \leftarrow -y_t * x_2 + y_2 * x_t$;
    if $s = 1$ then
    return(($U \geq 0.0 \wedge a \geq 0.0 \wedge b \geq 0.0) \vee$
        ($U < 0.0 \wedge a \geq 0.0 \vee b \geq 0.0$));
    else
    if $s = 2$ then
    return( not
        (($U \geq 0.0 \wedge a \geq 0.0 \wedge b \geq 0.0) \vee$
        ($U < 0.0 \wedge a \geq 0.0 \vee b \geq 0.0$)));
    else return(true);
end;
```

We will now define how the related numbers are assigned. Consider a design that has all the related numbers set to zero. This is easily accomplished by visiting each bubble and segment while setting their related numbers to zero. We now proceed by visiting each bubble. We process the bubbles as follows. If the bubble has a zero related number, a new related number, say r, is assigned to the bubble and we trace all the attached wires while setting their related number to r. We now process the bubbles at the ends of the wires just visited while setting their related number to r. This continues until all the reachable primitives have been assigned. This is a simplified description. We will now precisely define the setting of the related numbers by four recursive algorithms.

The procedure "assignrelatedb" (B) will assign related numbers to the bubble B. We will use the function "newrelatednumber" that will return a new unique related number. Newrelatednumber can easily be created by using a global variable that is increased by one for each call of newrelatednumber. If the bubble B is an Nbubble we will check the Nbubble's related number for zero and if it is zero we will call the procedure "setnbubble" with a new related number and bubble B. Otherwise the bubble must be a Ibubble. An Ibubble may have 1 to |reln[*]| related numbers. We will use a for loop, indexed by i, to step through all the possible related numbers for the Ibubble. If the related number index by i is zero then we will call setibubble. But before we do so, we will locate the model related number by using a while loop with index j. Notice in the procedure, we are looping until we find the model relnets[j] equal to the Ibubbles mbubbles reln[j]. We now call the procedure setibubble with a new related number, the bubble B, the model equivalence related number for index j, the model related colors for index j, and the model size for index j. We define procedure assignrelatedb as follows.

```
procedure assignrelatedb(B)
begin
    if B is Nbubble then
    begin
        if B.reln[1] = 0 then
        setnbubble(newrelatednumber,B);
    end else
    begin
        Integer i, j;
        for $i \leftarrow 1$ to |B.reln[*]| do if B.reln[$i$] = 0 then
        begin
            $j \leftarrow 0$;
            while B.mbubble.model.relnets[$j \leftarrow j + 1$] $\neq$
                B.mbubble.reln[$i$] do;
            setibubble(newrelatednumber,B,B.mbubble.reln[$i$],
                B.mbubble.model.relatedcolor[$j, *$],
                B.mbubble.model.relatedside[$j$]);
        end;
    end;
end;
```

The procedure "setnbubble"(r,B) will assign the related number r to Nbubble B if the bubble B has a zero related number. This check is performed to prevent infinite recursion, if there is a loop in the data structure. The procedure will step through all the Nsegments of bubble B with index i while checking for an Nsegment zero related number. If the Nsegment related number is zero we will call procedure tracensegment that will trace the wire path while setting the wires primitives related number. We will pass the related number r, the Nsegment index by i and the direction of the segment. A true direction is where the segment starts at the bubble. We define the procedure setnbubble as follows.

```
Procedure setnbubble($r$,B)
begin
     Integer i;
     if B.reln[1] $\neq$ 0 then return
     B.reln[1] $\leftarrow r$;
     for $i \leftarrow 1$ to $|B.\text{segs}[*]|$ do if B.segs[$i$].reln[1] = 0 then
     tracensegment($r$,B.segs[$i$],B.segs[$i$].starto=B);
end;
```

The procedure tracensegment and traceisegment will use the procedure "next"(O, *dir*) that returns the next primitive from object O. The direction *dir* is true if we are proceeding from start to end. We define the procedure next as follows.

```
Primitive procedure next(O,$dir$)
begin
     if O is segment then
        return(if $dir$ then O.endo else O.starto)
     else
     if O is around then
        return(if $dir$ then O.ends else O.starts)
     else
        return(NULL);
end;
```

The procedure "tracensegment"(r,S, *dir*) is used to follow Nsegments and arounds in direction *dir* to a bubble B while setting the related number to r. If the bubble B is an Nbubble we will call setnbubble with the related number r and bubble B. Otherwise, the bubble B must be an Ibubble. We will need to find the equivalence number of the Ibubbles model to remap. This is accomplished by looping through the list of related numbers for the Ibubble with index i and looking for the proper match. For mbubble related number indexed by i we need to find the model relatedcolor and related side. This is accomplished with a while loop with index j. The function "anyon" will return a true if there is a true in the passed Boolean array. We can test for a proper match by checking the segments rule presentcolor with the models related color for index j. If there is any relevant colors and the segment is on the proper side we will call setibubble with related number r, the bubble B, the model equivalence related number, the model related colors, and the model related side. We define the procedure tracensegment as follows.

```
procedure tracensegment($r$,S,$dir$)
begin
    Bubble B;
    S.reln[1] $\leftarrow r$;
    while next(S,$dir$) is around do
    begin
        S$\leftarrow$next(next(S,$dir$),$dir$);
        S.reln[1] $\leftarrow r$;
    end;
    B$\leftarrow$next(S,$dir$);
    if $B$ is Nbubble then setnbubble($r$,B)else
    begin
        Integer i, j;
        for $i \leftarrow 1$ to $|B$.reln[$*$]$|$ do
        begin
            $j \leftarrow 0$;
            while B.mbubble.model.relnets[$j \leftarrow j+1$] $\neq$
                B.mbubble.reln[$i$] do;
            if anyon(B.mbubble.model.relatedcolor[$j,*$] $\wedge$
                    S.rule.presentcolor[$*$]) $\wedge$
                testside(B,S,B.mbubble.model.relatedside[$j$])then
            begin
                setibubble($r$,B,B.mbubble.reln[$i$],
                    B.mbubble.model.relatedcolor[$j,*$],
                    B.mbubble.model.relatedside[$j$]);
                return;
            end;
        end;
    end;
end;
```

The procedure "setibubble"(r,B,e,$m[*]$,z) will set the Ibubbles related number to r for model equivalence related number e. The Boolean array $m[*]$ is the model related color, and the integer z is the side for the equivalence number e. The procedure first locates the position in the bubbles related number array reln[$*$] with index l for the equivalence related number e. If there is not one or the related number index by l is not zero we are done. Next the related number indexed by l is set to r. We will now proceed to check each of the segments of bubble B. We will let the pointer or link q equal the current segment. There can be two types of segments attached to the Ibubble. We will first consider when the segment q is an Nsegment. We will first check to see if the segment can be equivalent to the bubble for the related equivalence number e by checking for proper color match and side. If this is true, then we must make sure there is no better match by checking for any valid equivalence before the index l. Finally if there is none and the Nsegments related number is zero we call tracensegment with related number r, Nsegment q, and the direction. For the case where the segment q is an Isegment, we first find the equivalence related number indexed by j. If the Isegment related number is non zero we call traceisegment with the related number r , Isegment q, the direction, the model equivalence related number e, the model related colors, and the model related side z. We define the procedure setibubble as follows.

```
procedure setibubble($r$,B,$e$,$m[*]$,$z$)
begin
    Segment $q$;
    Integer $i,j,k,l$;
    $l \leftarrow$ |B.mbubble.reln[*]|;
    while $l > 0 \wedge$ B.mbubble.reln[$l$] $\neq e$ do $l \leftarrow l+1$;
    if $l = 0 \vee$ B.reln[$l$] $\neq 0$ then return;
    B.reln[$l$] $\leftarrow r$;
    for $i \leftarrow 1$ to |$B$.segs[*]| do
    if B.segs[$i$] is nsegment then
    begin
        $q \leftarrow$B.segs[$i$];
        if anyon($m[*]$ $\wedge$ $q$.rule.presentcolor[*]) $\wedge$
        testside(B,$q$,$z$)then
        begin
            $k \leftarrow l$;
            while $k > 0$ do
            begin
                $j \leftarrow 0$;
                while B.mbubble.model.relnets[$j \leftarrow j+1$] $\neq$
                    B.mbubble.reln[$k$] do;
                if anyon(B.mbubble.model.relatedcolor[$j$, *] $\wedge$
                    q.rule.presentcolor[*]) $\wedge$
                    testside(B,q,B.mbubble.model.relatedside[$j$])then
                Done;
            end;
            if $k = 0 \wedge$ B.segs[$i$].reln[1] $= 0$ then
            tracensegment($r$,B.segs[$i$],B.segs[$i$].starto=B);
        end;
    end else
    begin
        $q \leftarrow$B.segs[$i$];
        $j \leftarrow$ |$q$.msegment.reln[*]|
        while $j > 0$ $\wedge$ $q$.msegment.reln[$j$] $\neq e$ do $j \leftarrow j+1$;
        if $j \neq 0$ $\vee$ $q$.reln[$j$] $= 0$ then
        traceisegment($r$, $q$,B= $q$.starto, $e$, $m[*]$, $z$);
    end;
end;
```

The procedure "traceisegment"$(r, \mathrm{S}, dir, e, m[*], z)$ is used to follow Isegments and arounds in direction *dir* to an Ibubble B while setting the related number to r. We first need to find the position in the Isegment's related number array for the equivalence number e. This is accomplished with a loop with index i. If there is no equivalence related number or the Isegment related number indexed by i is non zero we are done. We then set the related number indexed by i to r. We now proceed through the list of Isegments and arounds setting the related number as previously described. We then call setibubble with the related number r, the Ibubble B, the equivalence related number e, the model related colors $m[*]$, and the side z. We define the procedure traceisegment as follows.

```
procedure traceisegment($r$,S, $dir, e, m[*], z$)
begin
    integer $i$;
    Bubble B;
    $i \leftarrow$ |S.msegment.reln[*]|;
    while $i > 0 \wedge$ B.msegment.reln[$i$] $\neq e$ do $i \leftarrow i + 1$;
    if $i = 0 \vee$ s.reln[$i$] $\neq 0$ then return;
    S.reln[$i$] $\leftarrow r$;
    while next(S,$dir$) is around do
    begin
        S$\leftarrow$next(next(S,$dir$), $dir$);
        $i \leftarrow$ |s.msegment.reln[*]|;
        while $i > 0 \wedge$ B.msegment.reln[$i$] $\neq e$ do $i \leftarrow i + 1$;
        S.reln[$i$] $\leftarrow r$;
    end
    B$\leftarrow$next(S,$dir$);
    setibubble($r$,B, $e, m[*], z$);
end;
```

We have now shown how the related numbers are defined with four simple recursive procedures. Notice that we will visit each bubble twice and each segment and around once. We can consider the complexity to be approximately order N, $O(N)$ where N is the number of primitives.

3.7 Specifying a Technology

The design of a technology file need not be fully completed before it can be used. A design using a partial file can only use the models and wiring medium that has been specified. The file may be updated with new models and wiring medium, or existing models may have limited update. The geometry design rules can be updated for correction to the design or alteration for a fabrication process. A technology file is sometimes designed in parts, which allows debugging. First a small version is created and then it is updated to the desired version.

The first step in specify a technology, assuming you have decided on a technology, is to gather the information about the geometric design rules. The information generally is available from the fabricator as in nMOS Mosis[Mosis 84], or in a text book such as in Mead and Conway[Mead 80].

The basic unit and the grid size is determined next. The basic unit should be a number which is meaningful to you. Design rules are specified in a basic unit as in λ in Mead and Conway[Mead 80]. For example we will use 100 units as the base for the nMOS description which corresponds to λ. The basic unit may not be easily changed for a design so it is important to make a wise choice here. The grid size is a multiple of the basic unit. Each bubble will reside on a grid unit. This is generally a multiplication of a basic unit as in $1/4\lambda$. The grid size may be changed for an existing design. The system will automatically realign the bubbles to the new grid.

The next step is to determine what colors are to be used and what they represent. Recall that a color may be any of the following: rule, physical, and CIF. Look at the table in the next section for an example of the nMOS rules. Colors may be added, deleted, or modified to an existing design. During the description of the models it will probably be necessary to come back to the color description and add special rules colors.

Next the rules for the wiring are defined. This is accomplished by defining a rule with the appropriate colors for each wire. Allowed bubble connection to the wire based on colors is described next. A rule needs to be defined for each separate width wire. For example if you want a 200 and a 400 width blue wire then two rules are necessary. Colors may be needed for interaction between wires where there exists a geometric design rule. For example in nMOS there is a red to green rule. In the nMOS example a color blah was added as a rule color to represent the red - green geometric design rule. The name of the rule will appear in the menu in the graphic editor.

The next step is to define the models. A model can represent a contact, a transistor, or device such as a diode. A contact is a passive device that is used to establish a connection between different layers. The contacts are the next item to be defined. At this time a decision needs to be made whether contacts pass related numbers for dissimilar rules or not. In the nMOS example they do not pass related information. Only similar wires connected to a contact will. Contacts with angle rules are addressed next. They can be defined using the technique outlined in Section 3.5.1 on preserving angles. New contacts can always be added to a technology file for an existing design.

The transistor and devices are defined next. The first decision is how much bending in the devices will be allowed. This will determine the number of bubbles for a device. Then a mapping of segments and bubbles to the device is performed to create the desired shape of the device. Along with the mapping, rules are created for the appropriate bubbles and segments. Next the bubbles that are to be used for connection points are identified and marked as such. Last, the related and significant mapping is defined. The amount of deformation of the model can be controlled by using the method outlined in Section 3.5.1.

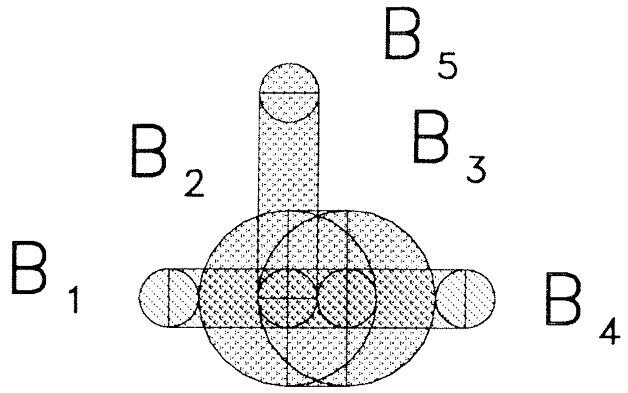


Figure 3.27 Consistency Problem Example

One of the problems in designing a model is to overlook being consistent within the model. For example in Figure 3.27 the segment B_5B_3 geometric rule to bubble B_3 should be less than or equal to the geometric rule between bubble B_2 to B_3. If the rule for the segment was greater, the move algorithm would think there needs to be a bulge in the segment that would cause the algorithm to fault. We define the consistency of a model as any segment S_j which is a member of the model or can connect to the model and the segment end bubble B_e which is a member of the model and any other bubble in the model B_i. Then following is true for a model to be consistent -

$$\mathrm{MINW}(S_j, B_i) <= \mathrm{MINW}(B_e, B_i).$$

The final item to define in the technology is to define the cost for each of the rules. This will be discussed in the chapter on annealing. Things to consider are what wires are to be shorter than others. Are there any special wires that require short routes?

3.7.1 An Example: nMOS Models

The nMOS technology file definition is taken from Mead and Conway[Mead 80] and augmented by the design rules for the Computer Science VLSI Class class at Caltech for 1984. We may use any units we wish in our system. In designing a technology file we will decide on a basic unit. We will use the unit of λ as described in Mead and Conway. The reason we chose λ as our unit was to be able to compare cells we compact with cells done by hand and by other automated compactors. Most comparable test cases are done in λ units. We will assign one λ unit to be 100.0 internal units. For example a polysilicon wire would be 200.0 units. All Bubbles in our system will reside on some minimum grid. For the nMOS rules we will use a minimum grid of 25.0 units which is 1/4 of a λ. The following table defines the colors that we will be using for nMOS. The first column is the color name while the second is the nMOS name. Blank nMOS name means that this is a rules color or a color for some other purpose. The third color is the standard or minimum width for the color. The forth column is the minimum distance between objects of that specific color. The fifth column is the use for the color.

Color	nMOS	Width	Mindistance	Use
Blue	Metal	300.0	300.0	Rule,physical,CIF
Green	Diffusion	200.0	300.0	Rule,physical,CIF
Red	Polysilicon	200.0	200.0	Rule,physical,CIF
Blah		200.0	100.0	Rule
Gate		200.0	200.0	Rule
Black	Cut	200.0		Physical,CIF
Yellow	Implant	200.0		Physical,CIF
Brown	Buried	200.0		Physical,CIF

Table 3.7 nMOS color Attributes

The first three colors in the table are the standard nMOS colors. The color "blah" is a monochrome color that is used to represent the design rule between red and green. The color "gate" is used for the nMOS gate rule. The last three colors are non rule–colors and are used for fabrication. The Figure 3.27 defines the patterns that will be used to represent the colors in our cells.

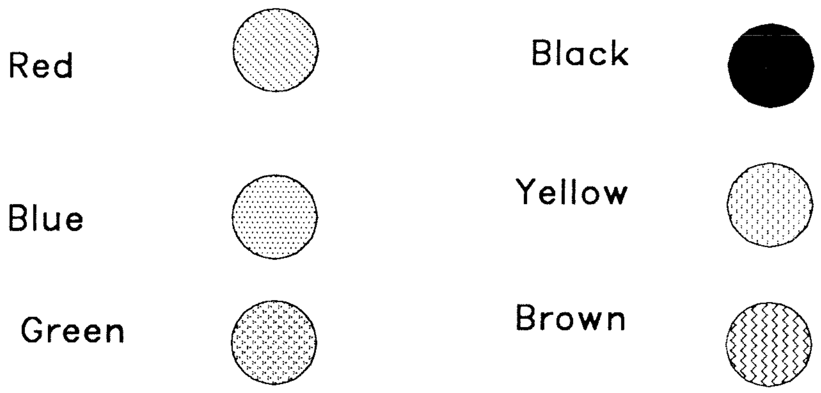


Figure 3.28 nMOS Color Pattern Representation

Appendix A will present the syntax for the technology file and the nMOS example. We will only present a subset of the technology file in this section. Appendix A will present the full definition in the technology file format.

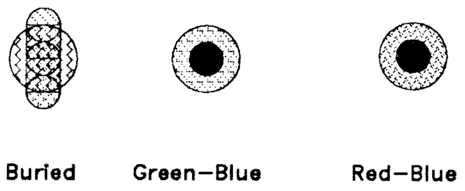


Figure 3.29 nMOS Contacts

The three contacts that we will be using are shown in Figure 3.29. In general the contacts provide a path from one layer to the next. We will not allow the contact to pass related information for unlike layers. For the green-blue contact we will let the blue color have an equivalent related number of 1 while the green color has a equivalent number of 2. In a similar manner we will let the blue color have an equivalent related number of 1 for the red-blue contact and the red color have an equivalent number of 2. We will impose no restriction on related red, green, or blue of these two contacts. The center cut color will also have a gate color of the same width to prevent related contacts from getting closer than 4λ. We will allow a green and blue connection to the green-blue contact while we will allow a red and blue connection to the red-blue contact.

The buried contact has red and green protruding fingers that are used to connect to the buried device and to prevent the angle between the attached wires from getting less then 90°. The red colored finger bubble is used to connect a red wire while the green colored finger bubble is used to connect a green wire. The fingers are used to prevent the end bubbles from getting closer than 3λ from center to center while we allow the finger bubbles to be 2λ from the center. We will let the red finger have an equivalent related number of 1 and the green have an equivalent related number of 2 while the center bubble has both. The related exception will be used to hold the finger bubbles away from the center bubble. This is accomplished by reducing the related width for the center bubble to the green and red finger bubbles. The center bubble has a gate layer to prevent other contact and devices from violating the cut to cut, and gate to cut separation rule.

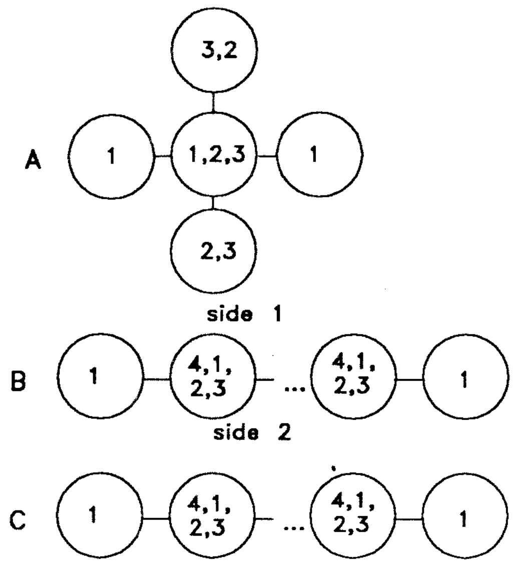


Figure 3.30 nMOS Generic Equivalence Related Numbers

All the transistors will use one of three types of equivalent related number assignments as shown in Figure 3.30. Equivalent related number 1 is for red, 2, and 3 are for green and 4 is the internal equivalent related number. Four is used to hold the many bubbled transistors apart. Type A is used for the square transistors. Notice the ordering for the numbers 2, and 3 in type A. On the top it is 3, and 2 while on the bottom its 2, and 3. This allows green connections at the top to use number 3 for the equivalence while 2 is used for bottom. Type B is for the long red transistors. We will use 2 as side 1 and 3 as side 2. Notice that the number 4 is the first number in the center bubbles. This is so the internal related number will take precedence in the related exception handling. Type C is used for the long green transistors. Notice that type B and C may be extended for any desired length.

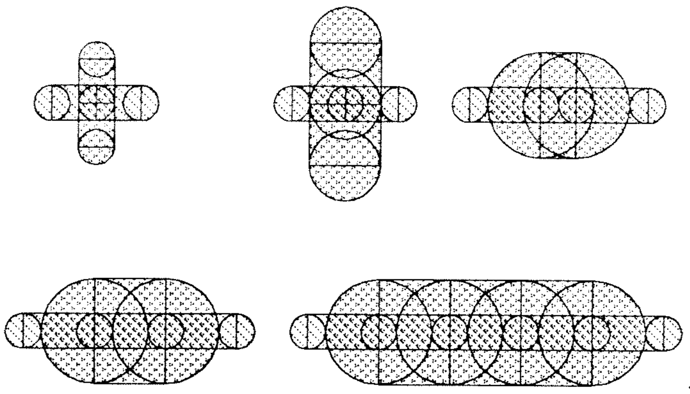


Figure 3.31 nMOS Enhancement Transistor

The top two enhancement transistors on the left will use equivalent related numbers assignment as previously described of type A. The center bubble effective radius for the related exception will be reduced from the standard to a value that will let finger bubbles come in toward the center as shown in Figure 3.31. We will define the color Blah as significant. Notice that the related numbers are the same for the end two green bubbles numbers 2 and 3 as shown in Figure 3.30. If this was not allowed, the green bubbles could not be close as the design rules would allow since the green bubble at the opposite would hold it out.

The remaining three enhancement transistors use the equivalent related number assignment of type B. Notice the use of the internal related number 4 which holds the primitives of the transistor apart. We can create any desired length of transistor by adding extra bubbles.

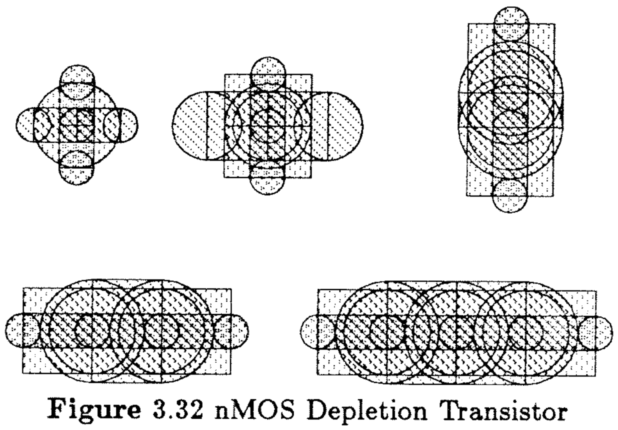


Figure 3.32 nMOS Depletion Transistor

Similarly the two top depletion transistors on the left will use equivalent related numbers assignment as previously described of type A for their model. The center bubble effective radius for the related exception will be reduced from the standard to a value that will let finger bubbles come in toward the center as shown. We will define the color Blah as significant.

The remaining three depletion transistors use the equivalent related number assignment of type C as shown in Figure 3.33. Notice the use of the internal related number 4 that holds the primitives of the transistor apart. We can create any desired length of transistor by adding more center bubbles.

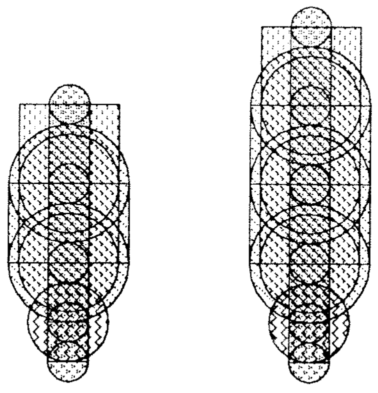


Figure 3.33 nMOS Pullups

The Pullups use the equivalent related number assignment of type C. We can create a pullup of any desired length by adding extra bubbles to the newly created model and added the new model to the rules file.

3.7.2 Modify Geometric Design Rules

Many different fabrication lines are available for a given technology. Given an existing design we would like to be able to modify the technology file for variations in processes lines. This is very important since we may find several fabrication houses that can produce our chip but each may have variation in their design rules. We would like to use the fabrication house which is the most cost effective.

There are two possible cases the design rules file is modified. The first case is where the modification to the technology file will introduce no design rule violation in an existing design. Generally this is where we reduce the width or separation rules. All that is necessary is to modify the technology file and do a partial compaction. A full compaction is not necessary since we assume the design has been previously compacted. The second case is where the modification would introduce a design rule violation. Generally this is where we increase a width or a separation rule. In this case we would project or scale our design by the maximum change in the technology file. We would then modify the technology file while not introducing any design rule violations. The projection process would increase the arounds radius which may or may not be the proper radius. We would next collapse the arounds as described in Section 4.2.1 and partially compact.

Another modification we may wish to do is to replace an existing device with another. An example of this would be to replace an nMOS Butting contact with a nMOS buried. Model replacement would require manual intervention. In the example a butting contact is capable of accepting connection from blue, green and red wires while the buried only accepts green and red wires. Therefore if a butting contact has blue connecting wires, two instances must be added to replace the one. What is required is first to add the new model to the technology file. Next use the graphic editor(see Chapter 5 on graphic editor) to delete the old instance and add the new ones while making necessary connections. Finally delete the unwanted models from the technology file.

3.8 Post Processing and CIF Output Generation

The framework for a design is oriented toward compaction and does not have efficient data storage for CIF format. CIF[Mead 80] is an interchange mediate to represent geometric object on layers. We will translate designs to a CIF format using the CIF primitive polygon which is a closed region defined by a set of points and a flash which is a circular object.

The process of translating a cell design to CIF is accomplished in a two step process. First the design is translated to a set of colored wires and flashes where a wire is a set of points which has a radius and a flash is a circle with a radius. Flashes are only generated when they are not an intermediate point in a wire with the same radius and color. The minimum number of wires are generated. The final step is to translate the wires to polygons and proceed with the sliver process. The sliver process will be discussed later on in this section. The polygons and flashes are then output to a CIF file.

The generation of wires and flashes is as follows. For each CIF color c and each unprocessed bubble b, do the following. Trace and generate wires for each different width colored c segment through all directly connected bubbles. Generate flashes for each bubble just visited which is not a member of a wire and mark the bubbles as processed.

A problem known as the "Sliver" problem is that of generating a sliver which is very narrow. During the fabrication process the sliver can break off and land somewhere else on the chip which could cause a faulty chip. This problem is not mentioned in the design rules but comes from folklore[Seitz 85b]. Two examples of sliver problems are shown in Figure 3.34. There are two type of possible sliver generation. The first is shown on the left which is generated from two wires with a sharp angle. The second is from two related objects that have partially closed together while leaving a small gap.

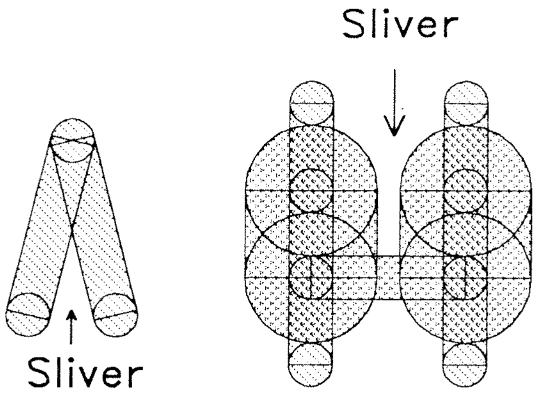


Figure 3.34 Wire and Transistor Sliver

The first sliver problem is corrected by adding a curved fillet in the gap between the wires with a radius of the minimum distance for the color. The solution is shown in Figure 3.35. This fillet is added during the translation of wires to polygons.

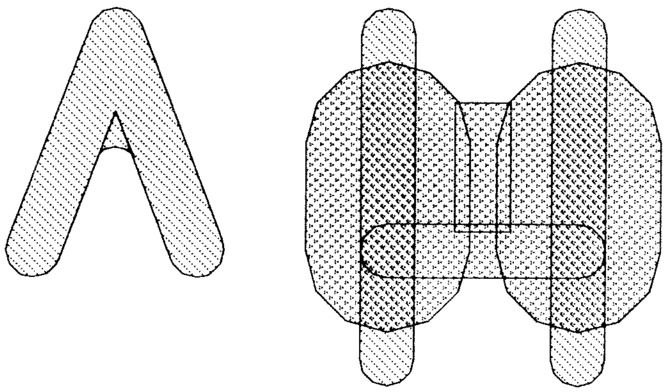


Figure 3.35 Wire and Transistor Sliver Solution

The second problem requires the use of a general polygon package[Sutherland 78, Barton 80]. A general polygon package can intersect, union, expand and shrink polygons, where a polygon can contain arcs and arbitrary angles. The second sliver process takes advantage of the fact that an expand and shrink are not symmetrical. The second sliver process is as follows. For each CIF layer, *c*, do the following. Expand the polygons for layer *c* by an amount smaller than one half of the minimum separation geometric design rule. Next merge all intersecting polygons for layer *c*. Finally shrink the resultant polygons by the amount expanded. The solution for the sliver problem is shown in Figure 3.35.

3.9 Cell Definition

Our design is contained in cells. A cell is a polygonal bounded region. The interface to the outside world from within the cell is through bubbles that we call "ports" which rest on the perimeter of the cell. With our structure we are not limited to any particular shape for the cell. The cell boundary is defined by a polygon starting at a point P_1 as shown in Figure 3.36 and preceding counter clockwise through all the points P_i.

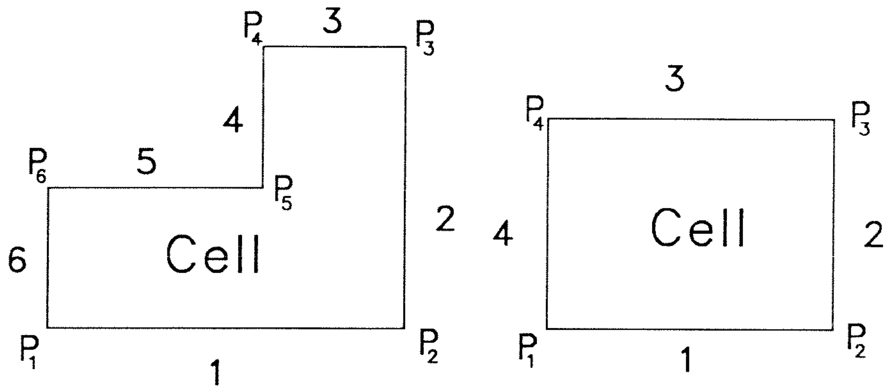


Figure 3.36 Cell Representation

The perimeter of our cell is described by an array of points $P_i(x,y)$ named sides[*] which are numbered around the cell in a counter clockwise fashion starting with 1 and ending with the number of sides in the cell. We number the side of the cell defined from P_i to P_{i+1} as side i. The port contains the integer side number which defines the side on which the port resides. The ports are allowed to move only along their side and not change sides.

A property of this type of cell is that it can be constructed to fit in any polygonal region. Of course the primitives in the cell must be able to compact to the region. We could design our chip to use any tiling pattern we wish. For now we will limit the cells to rectangles so that we will be able to compare our cells with other cells.

A cell contains the primitives of the cell, the technology file as data structures, and the shape of the cell. The primitives are contained in sets defined by the symbols {} as in the set of all bubbles in the cell. We define a cell as:

```
Cell(Bubbles{},Segments{},Arounds{},
     Colors[*],mindistance[*],
     Rules[*],Models[*],
     sides[*],
     CIF[*],cifnames[*],CIFA,CIFB).
```

3.9.1 Minimum Bounding Area

The "Minimum Bounding Area", A_{mb}, for a cell is the smallest area the cell could reside in, if the ports are allowed to move along the cell edge. This area is used as a measure to compare compactness of cells. A way to visualize the minimum bounding area is to move each cell side in until it hits a bubble or an around edge that is not a member of a wire that connects to a port on the edge being moved. The minimum bounding area is the area of the cell when the sides are moved in.

For comparison purposes we will be using rectangular cells as previously described. The minimum bounding area will sometimes be referred to as the minimum bounding box. The method that we use to calculate the minimum bounding box is to find the maximum bounding box of the bubbles excluding the ports. This minimum bounding box is then increased by side, for each around that is not a member of a wire to that side being increased and an outward perpendicular vector to that side is within in the around.

3.9.2 Ideal Bounding Area

The "Ideal Bounding Area", A_{ib}, is a measure of the area required for the cell excluding the wiring. The area is calculated by summing the area required for each model instance in the cell and non-instance bubbles. The calculation includes the fact that some models can overlap others. additionally, the wiring between connected bubbles that can not overlap is accounted for.

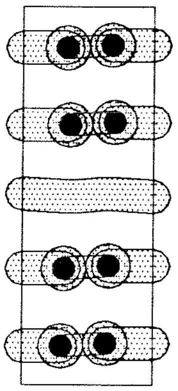


Figure 3.37 Sample Cell One

The cell shown in Figure 3.37 has the same ideal bounding area as the cell in Figure 3.38. Notice that the wiring in cell Figure 3.38 has the effect of increasing cell minimum bound area.

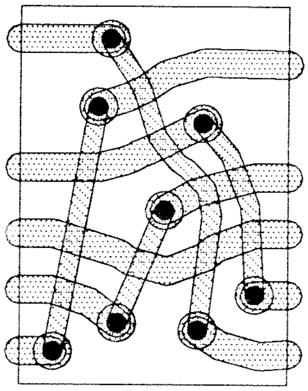


Figure 3.38 Sample cell Two

A measure that we will use to relate minimum bounding area to ideal area is A_{mb}/A_{ib} which gives us an idea how much of the cell is wiring. For the first cell the ratio is approximately 1 while the second is approximately 1.5. This measure will be used in the analysis in Chapter 7.

A measure for the minimum length, L_s, for a side of a cell which comes from the ideal area will be used for the compaction process. The L_s for a side is calculated by taking the maximum of the sum of the separation distance between port plus the maximum radius of the end ports. For the cell that is a rectangle we will calculate the minimum horizontal length H_m for the cell by using the maximum of the top and bottom side. For the vertical, V_m we will use the maximum of the left and right side.

We now have minimum limits for a cell. The minimum horizontal length is H_m, the vertical is V_m , and the approximate minimum area is M_{iba}. These constants for a cell will be used in the compaction process.

3.10 Conclusion

In this chapter we have defined a new representation for a VLSI design based on three simple primitives. We have shown how these primitives are composed to make wires and devices. The devices called "models" are described in an independent manner in a technology file. We have shown how we can abstract the circuit devices and interconnect from a design using these primitives. This level of abstraction can be used for a circuit schematic and design simulation.

We defined wires in our system as connected segments and arounds that progress from a bubble to bubble. Wires can be considered as tightly stretched bands that wrap bubbles or other arounds in the shortest possible distance.

We presented a new method to represent design rules in monochrome. This greatly simplifies the geometric design rules checking and the geometric design rules representation. A method of representing the exceptions in design rules called "related" was presented. This new method allows a simple technology file to describe complex geometric design rules.

4
Modifying the Structures

We would like to be able to compact our design to a smaller or different shape while preserving the topology and invariant condition as described in Section 3.3.4. The design can be modified in a sequence of small steps to produce a large modification. A small step is a move of a single bubble to a new position where the invariant condition is preserved. This chapter will present a move algorithm which can move a single bubble to a new location while preserving the invariant condition.

A bubble move will be an atomic action from which other modifications will be constructed. We will allow only one bubble to be processed at a time. The bubble may be moved only if the move preserves the invariant condition. We will not allow a bubble to push or move another bubble. An example of a cell before and after a bubble move is shown in Figure 4.1.

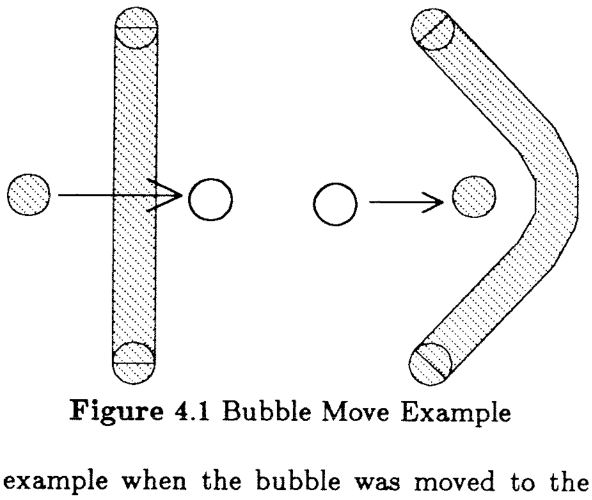


Figure 4.1 Bubble Move Example

Notice in the example when the bubble was moved to the new position the wire was pushed out by the bubble. A simple way to visualize the bubble move is by considering the bubble as a solid object that can slide on the board or plane. We can move only one bubble at a time and other bubbles will appear as blocks.

Visualize wires as rubber bands with a thickness. Now look at the example. Notice the bubble is pushed along the path to a new position. The intervening wire was pushed out by the bubble while being tautly stretched. We will call this process the bubble move. This chapter will present how the bubble move is accomplished.

The description of the bubble move algorithm will first be presented with an example that has been carefully chosen to contain most of the special cases, which is described in Section 4.1. The example presents a clear picture on how the algorithm works. Detailed description of the algorithm's two parts is presented in "clear path" described in Section 4.1.1 and in "queue construction" described in Section 4.1.2. These two sections go into detail about the algorithm and may be skipped.

All changes to our design will be processed through two sets which are the set of primitives to be added called "addq"{} and the set of primitives to be deleted called "deleteq"{}. We wish to be able to update our design with these two sets while preserving the invariant condition. We define the function "update"(U,addq{},deleteq{}) which returns a new design, U′ where all the primitives in the deleteq{} have been deleted from U and all the primitives in the addq{} have added to U, that is

$$U' \leftarrow \text{update}(U,\text{addq}\{\},\text{deleteq}\{\}).$$

We can consider the function update to happen in instantaneous time. A nice property of the function update is that of symmetry. A change can be reversed or undone by interchanging the sets, addq{} and deleteq{}, when the function update is called. This property is extremely important in the interactive environment. If we have constructed several changes which we are not happy with, the changes can easily be backed out by calling update with the proper sets.

Portions of the bubble move algorithm are used as utilities by the functions absorb bubble and around collapse. A design may have an intermediate bubble between the path of two wires. The intermediate bubbles may have been used as aids during the editing process or as a by product of the cell expansion process or other processes. The algorithm for intermediate bubble absorption will be presented in this chapter. During the editing processes bubbles may be deleted for a variety of reasons. Wires may wrap or go around the bubble to be deleted. The collapse of this type of around will be presented in this chapter.

This chapter will describe a uniform method for expanding or constricting the perimeter of the cell. The relative positions for the ports is maintained for both the expansion or constriction process. During the cell generation process in the graphic editor the case arises where we would like to be able to expand the cell to add more circuits. We call this process cell expansion. One edge of the cell is expanded at a time. The whole cell may be expanded by expanding each edge one at a time. To go along with the cell expansion we have the opposite of cell contraction. The cell contraction also proceeds one edge at a time.

4.1 Bubble Move Algorithm

The bubble move algorithm is best described with an example first. The example as shown in Figure 4.2 has been chosen with a variety of cases that should answer most of the questions about the algorithm. The example will be explained in monochrome as will the entire algorithm. We would like to move the bubble B_1 to the new position at $B_{1'} = (B_1.x + \Delta x, B_1.y + \Delta y)$, where $\overrightarrow{(\Delta x, \Delta y)}$ is the vector from the old position to the new. The possible new position is shown in the Figure 4.2 with an unshaded circle.

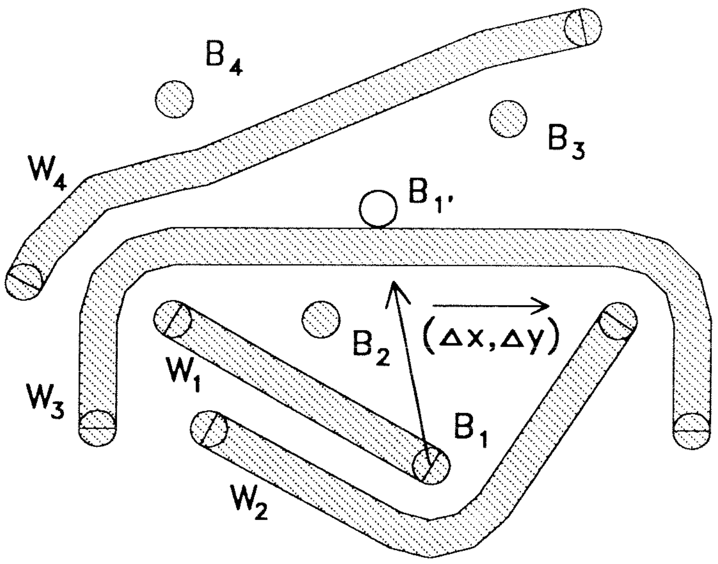


Figure 4.2 Bubble Move Example, Before Move

The example shows some of the interesting deformation to the wires that are necessary for the example move with the invariant condition preserved. Notice in Figure 4.3 that the bubble B_1 has an attached wire w_1 which will need to wrap bubble B_2 in the new position. The around of wire w_2 that wraps bubble B_1 may need to be unwrapped. This wire may need to wrap other primitives which will be discussed later. Now look at wire w_3 which will be pushed by the bubble and will need to be wrapped around B_1. In a similar manner wire w_4 will need to be wrapped. There is also an around that will possibly need to be unwrapped which is the around that wraps bubble B_3.

Before we can move the bubble B_1 to a new position, we need to know if the bubble can be moved directly - *i.e.*, in a straight line - to that position without violating the invariant conditions and without moving any other bubble. If we can move the bubble with no errors, then we can perform the move. This process naturally divides the move algorithm into two parts. The first is the "clear path" algorithm that determines whether we can accomplish the move with no geometric design rule errors and "queue construction", which constructs the addq{} and deleteq{}. In this example we will describe the queue construction as actually taking place. It is straight forward to translate this process into creating the add and delete queue.

Let's consider bubble B_1 as a solid ball that is moved along a straight line to the position $B_{1'}$. As the bubble moves along the path a region is defined by the radius of the bubble plus the minimum separation distance. Any bubble that intersects this region will violate the invariant condition. Notice in Figure 4.3 the region R_1 is defined by the lines parallel to the path. If we hit a wire, as shown with wire w_3, we need to increase the radius of the path above the wire by the radius of the wire when it wraps the bubble plus the radius of the wire, plus the minimum separation distance. We will save the wire with its wrapping radius in an array called "forwardP"[*]. Now if a bubble is in region R_2 it would need to be further away than a bubble in region R_1. As we move along, another wire, w_4, will be hit which we will add to "forwardP"[*] with the radius that w_4 would have if it wrap bubble B_1 in the new position. Notice that the radius of w_4 wrapping bubble $B_{1'}$ would need to be the previous wrap wires radius plus the radius of the two wires plus the minimum separation distance.

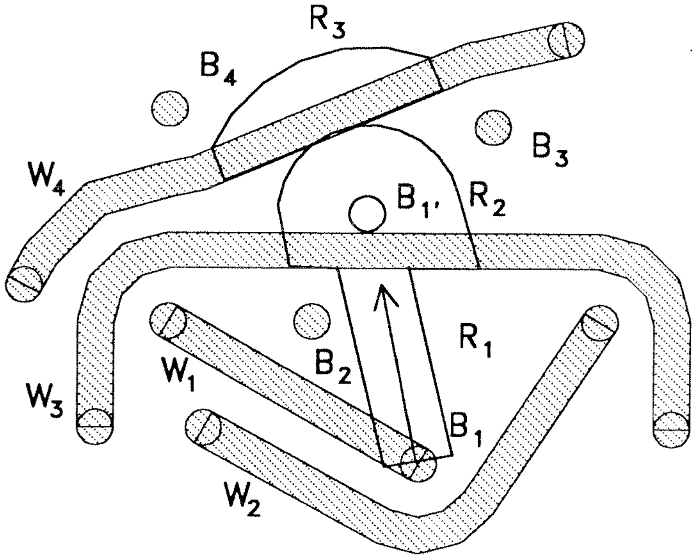


Figure 4.3 Bubble Move Clear Path Example, Before Move

Notice that we have defined an ordering in the array "forwardP"[*]. The array is ordered with the closer wire first. For our example "forwardP"[*] would contain w_3 followed by w_4. Also recall the array contains the radius that would need to be created for the wire when it wraps bubble $B_{1'}$. This property will be used in the queue construction algorithm.

Let's consider what would happen, if the bubble hit an around along the path. There are two cases for the around. The first is where the bubble B_1 is inside of the around. If the bubble B_1 is inside the around, then we will treat the around as a wire and processes as previously described. If the bubble is outside the around then the around must be treated as a bubble which is a solid object which would possibly cause the clear path to fail.

In a manner similar to the creation of "forwardP"[*], we will create an array "backwardP"[*] containing the wires that would need to be unwrapped. The ordering is closest to bubble B_1 as before. For our example this array would hold wire w_2. This array will be used later on by the queue construction algorithm.

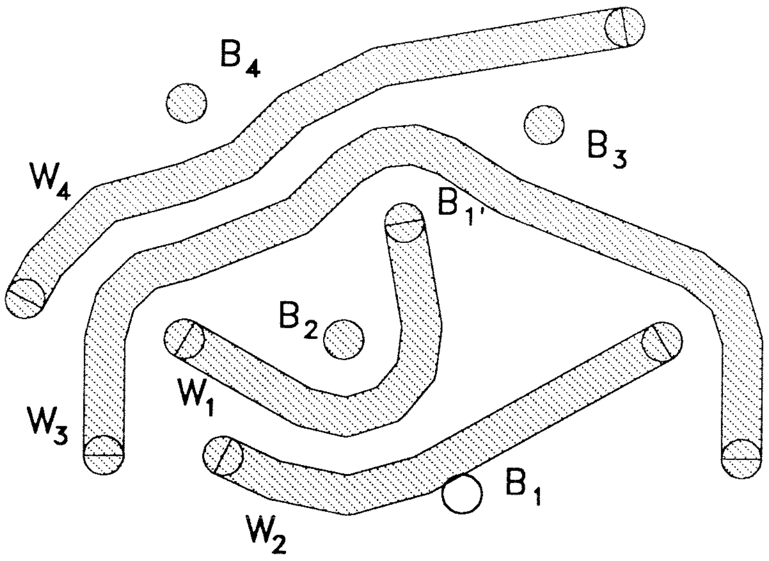


Figure 4.4 Bubble Move Clear Path Example, After Move

We have now shown how clear path works. The processing of the move by queue construction will now be explained. The queue construction takes advantage of the arrays that were constructed by clear path. Notice in the Figure 4.4 that the move has been performed. Now notice that we need to move wire w_4 first. If we did not, and moved some other wire, say w_3, there would not be the around of wire w_4 for wire w_3 to wrap. So, there is an ordering of processing. We need to process w_4, followed by w_3 and then move the bubble with attached wires, and finally move wire w_2. Notice this ordering is from the end of array "forwardP"[*] to the start, the bubble, followed by array "backwardP"[*] to the end. This order is the same order queue construction does the processing.

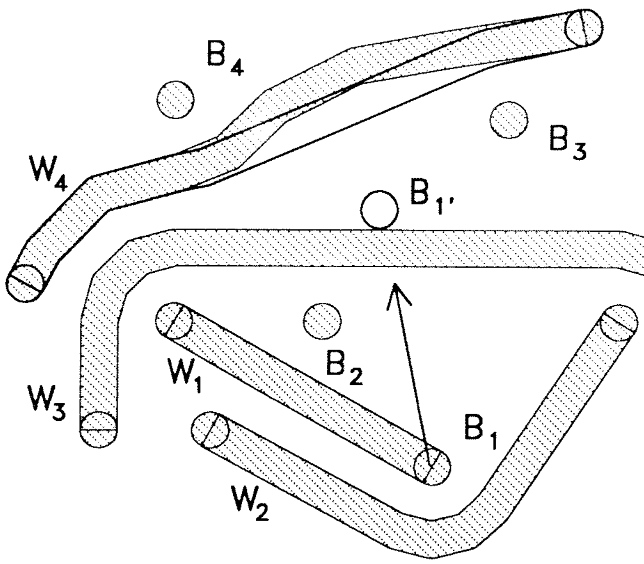


Figure 4.5 Bubble Move Clear Path Example, Partial Move

The first wire to be processed by queue construction is wire w_4 which is the last wire in "forwardP"[*]. The old position is shown as an outline in Figure 4.5. The new position is shown filled in. The old wire has two arounds where the first around wraps the end bubble of wire w_1 and the second around wraps bubble B_3. As we move the wire forward we create an around with a radius defined by "forwardP"[*] which will wrap bubble $B_{1'}$. The wire can be considered as a taught band that snaps, wrapping and unwrapping solid objects. In this case, wire w_4 would wrap bubble B_4 and unwrap the around for bubble B_3.

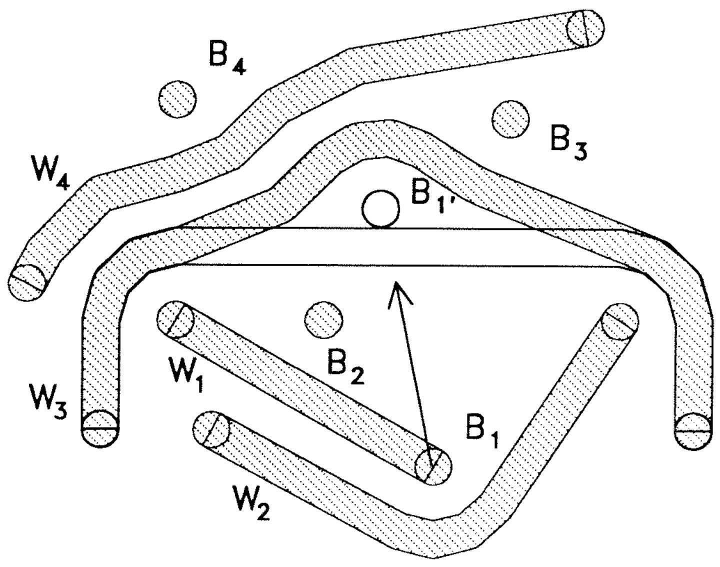


Figure 4.6 Bubble Move Clear Path Example, Partial Move

Wire w_3 is the next wire to be processed from the "forwardP"[*] array. We will create an around that wraps bubble $B_{1'}$ with the radius from "forwardP"[*] as shown in Figure 4.6. We will let the wire snap up to its new position. Notice that it wraps the around created previously within wire w_4 and wrap bubble B_3.

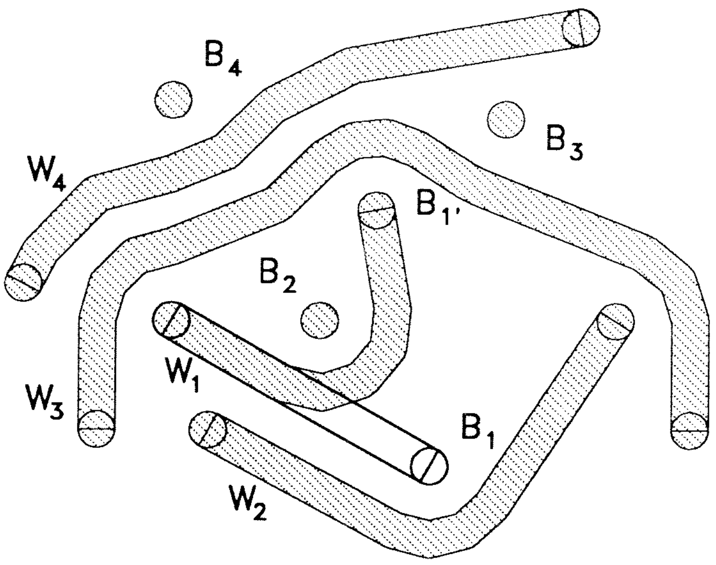


Figure 4.7 Bubble Move Clear Path Example, Partial Move

The array "forwardP"[*] is now exhausted and we will proceed by moving bubble B_1 to $B_{1'}$ as shown in Figure 4.7. As with the wires just processed, wire w_1 is dragged to its new position wrapping bubble B_2.

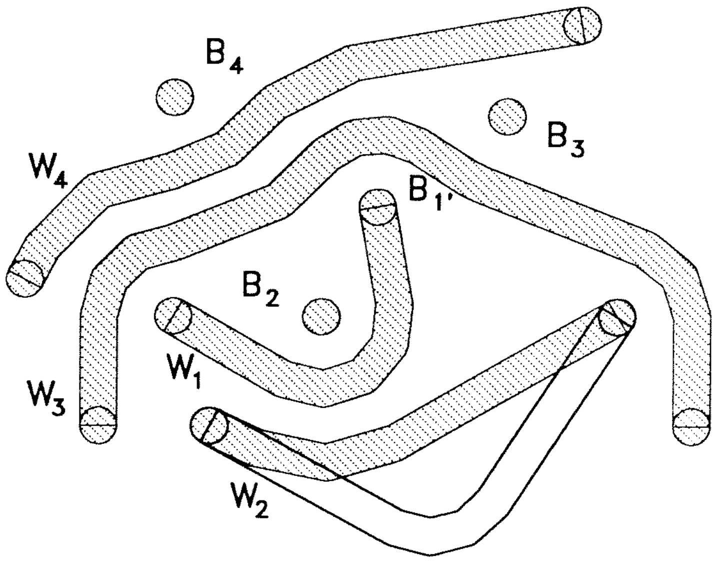


Figure 4.8 Bubble Move Clear Path Example, Partial Move

The last step of queue construction is to process the wires in array "backwardP"[*] from start to the end. For our example there is only one wire, w_2. Consider the wire as being held by bubble B_1. We remove bubble B_1 and allow the wire to snap into place. Notice that wire w_2 wraps the previously created around of wire w_1.

We have just demonstrated how the bubble move algorithm works for a complex example. All of the possible bubble moves can be reduced to parts of the example. Notice that before the move, the wires were tightly stretched, and after the move they are tightly stretched. This is a property of the algorithm.

The Boolean procedure "bubblemove" is used to construct a single bubble move in the addq{} and deleteq{} sets. The procedure will return a true if it was successful. The procedure "bubblemove" takes as parameters, the cell U, the bubble to move B, the delta position $(\Delta x, \Delta y)$, the addq{} and deleteq{} for the resultant move. The procedure uses two temporary arrays "forwardP"[*] and "backwardP"[*] for passing the wire wrap information created by clear path procedure to the queue construction procedure. We define the procedure as follows.

```
Boolean Procedure bubblemove(U,b,$\Delta x$, $\Delta y$,addq{},deleteq{})
begin
    array forwardP[*];
    array backwardp[*];
    if clearpath(U,b,$\Delta x$, $\Delta y$,forwardp[*],backwardp[*]) then
    begin
        qconstruct(U,b,$\Delta x$, $\Delta y$,forwardp[*],backwardp[*],addq{},deleteq{});
        return(true);
    end;
        return(false);
end;
```

4.1.1 Clear Path Algorithm

The clear path algorithm determines if we can move a bubble B to a new location defined by a delta position, $(\Delta x, \Delta y)$, in our cell U. Clear path also defines the array "forwardP"[*] which contains the list of primitives and their respective radius that will need to wrap bubble B at its new position. The array "forwardp"[*] is ordered by closest wrap primitive first. The array "backwardp"[*] is also defined which contains all the primitives that would need to be unwrapped when bubble B is moved. Array "backwardp"[*] is ordered by closest unwrap first. The clear path algorithm is defined by procedure "clearpath"(U,B,Δx, Δy,"forwardP"[*],"backwardP"[*]).

The variables U, B, Δx, Δy, "forwardP"[*], and "backwardP"[*] will be assumed to be global throughout the procedures and functions defined within a clear path.

The clear path algorithm proceeds in two steps. First, the "forwardP"[*] and "backwardP"[*] are constructed. This construction first examines the arounds attached to bubble B in B.arnds[*]. These arounds are then added to "forwardP"[*] or "backwardP"[*] depending if they would continue to wrap B in its new position or would possibly be unwrapped. The order of the arrays "forwardP"[*] and "backwardP"[*] is by the increasing absolute value of the radius of the primitive that would wrap bubble B. Next, our cell, U, is searched for any segment or around that would wrap bubble B in the new positions or any segment or around pushed by bubble B. These primitives are added to "forwardP"[*] in the proper order. The second step in the clear path algorithm is to check for any bubble or around in our cell that would violate the invariant condition if the bubble B was moved to the new position. Clear path would fail if any bubble or around would violate the invariant condition when bubble B is moved along the path to the new position.

The second step of the clear path algorithm will be described first which checks if the bubble B can be moved without geometric design rule violation. The construction of the "forwardP"[*] and "backwardP"[*] arrays will be discussed later on in Section 4.1.1.2. We wish to move the bubble B_1 as shown in Figure 4.9 to a new position defined by a delta position, $(\Delta x, \Delta y)$, to a new position at $B_{1'} = (B_1.x + \Delta x, B_1.y + \Delta y)$. Each bubble, B_i, in our cell U, will be tested with the path of the bubble B_1 to the new position at $B_{1'}$ for a geometric design rule violate. In addition, each around A_i which is outside the bubble B_1 is tested for a possible geometric design violation. We will use the normalized line equation passing through point $P = (B_1.x + \Delta x, B_1.y + \Delta y)$ to bubble B_1 and the perpendicular normalized line equation passing through bubble B_1 for this testing.

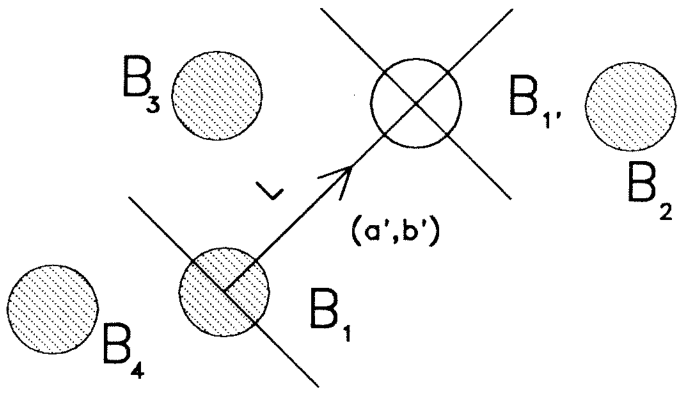


Figure 4.9 Clear Path Move Position

We let the length of the move be defined as

$$L = \sqrt{(\Delta x)^2 + (\Delta y)^2}. \tag{4.1}$$

For the normalized line equation,

$$a * x + b * y + c = 0 \tag{4.2}$$

passing through point P to bubble B_1 we will let

$$\begin{aligned} a &= (\Delta y)/L, \\ b &= -(\Delta x)/L, \\ c &= -a * B_1.x - b * B_1.y. \end{aligned} \tag{4.3}$$

For the perpendicular normalized line equation

$$a' * x + b' * y + nc = 0 \tag{4.4}$$

passing through bubble B_1 we will let

$$\begin{aligned} a' &= -b, \\ b' &= a, \\ nc &= -a' * B_1.x - b' * B_1.y. \end{aligned} \tag{4.5}$$

Notice the vector $\overrightarrow{(a', b')}$ points along the path move vector $\overrightarrow{PB}$ as shown in Figure 4.9. This property will be used later on.

We will use the procedure "Coveredprimitive"(p, w, i, j) described in Section 4.1.1.1. The procedure returns via the variable w, the minimum separation distance between bubble B_1 and the primitive p which includes any wire which may be between bubble B_1 and primitive p. Covered primitive uses the array "forwardP"[*] that contains the tuple (r, p) where r is the signed radius of wire primitive p which will wrap bubble B_1 in its new position. The procedure searches from the last primitive in "forwardP"[*] looking for a primitive where the minw [1] is greater than zero and the primitive is between the new position at P and bubble B_1. If one is found the absolute value of the radius in "forwardP"[*] plus minw is returned. Otherwise minw for the primitive p and the bubble B is returned.

To test if there is a clear path for bubble B_1 to move to point P we will test each bubble B_i in our cell for an invariant condition violation with the path bubble B_1. In addition we will test each around A_i which bubble B_1 is outside for an invariant condition violation. If any bubble B_i or around A_i fails the tests, clear path will fail. For each bubble B_i in our cell we will do the following. First, "coveredprimitive" is called that returns w which is the minimum separation distance of bubble B_1 to bubble B_i including any primitive between the two. If w is equal to zero there is no separation rule and we will proceed with the next bubble.

We will let

$$r = |a * B_i.x * b * B_i.y + c| \tag{4.6}$$

which is the distance from the line to bubble B_i. Also let

$$d = a' * B_i.x * b' * B_i.y + nc \tag{4.7}$$

[1] Minw is a procedure which returns the minimum design rule separation between two passed primitives. See Section 3.3.3.

which is the directed distance from the perpendicular line to bubble B_i.

If $d > L$ then the bubble is above point P in the region of bubble B_2 as shown in Figure 4.9. For this case the clear path fails if the following is true.

$$w > \sqrt{(d-L)^2 + r^2}. \tag{4.8}$$

If $d > 0 \wedge d <= L$ then the bubble is below point P in the region of bubble B_3 as shown in Figure 4.9. For this case if the following is true then the clear path fails, in which case we will return with a false.

$$w > r. \tag{4.9}$$

If $d \leq 0$ then the bubble is below bubble B_1 in the region of bubble B_4 as shown in Figure 4.9. This case can be skipped since a bubble violating a rule here would violate the invariant condition.

We will treat arounds as if they are bubbles for the around test. We will only check arounds that do not have the bubble B_1 inside. The reason for this can be seen from the previous example. Arounds that bubble B_1 is not inside can not be pushed since the around wraps another bubble and acts like a hard wall. For each around A_i in our cell we will do the following. First we will test if the bubble B_1 is inside around A_i and if so we will continue with the next around. Next we call "coveredprimitive" with primitive A_i which will return w which is the minimum separation of bubble B_1 to around A_i including any primitive between the two. If w is equal to zero there is no separation rule and we will proceed with the next around. Next we add the radius of around A_i to w. We will let

$$r = |a * A_i.b.x * b * A_i.b.y + c| \tag{4.10}$$

which is the distance from the line to the center of around A_i. We will let

$$d = a' * A_i.b.x * b' * A_i.b.y + nc \tag{4.11}$$

which is the directed distance from the perpendicular line to the center of around A_i.

If $d > L$ then the around is above point P in the region of bubble B_2 as shown in Figure 4.9. For this case clear path fails if the following is true.

$$w > \sqrt{(d-L)^2 + r^2}. \tag{4.12}$$

If $d > 0 \wedge d <= L$ then the around is below point P in the region of bubble B_3 as shown in Figure 4.9. For this case if the following is true then the clear path fails in which case we will return with a false.

$$w > r. \tag{4.13}$$

If $d \leq 0$ then the around is below bubble B_1 in the region of bubble B_4 as shown in Figure 4.9. This case can be skipped since a around violating a rule here would violate the invariant condition.

4.1.1.1 Covered Primitive Definition

The procedure "Coveredprimitive"(p, w, i, j) determines the minimum separation distance between passed primitive p and bubble B_1 including any primitive between the two which is in array "forwardP"[*]. The returned variable w is the minimum separation distance. The returned variable i is the index in "forwardP"[*] of the highest covered primitive. The variable j is the starting index in "forwardP"[*]. How variables i and j are used will be described in Section 4.1.1.2.

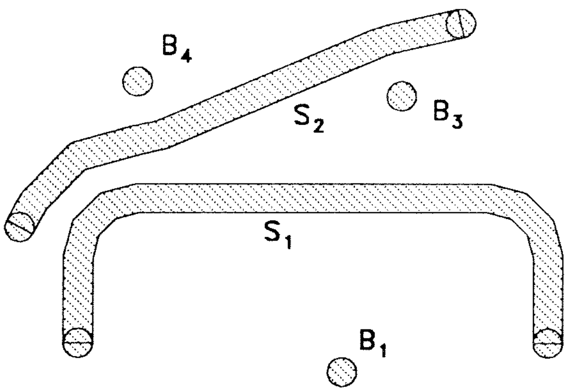


Figure 4.10 Covered Primitive Example

The array "forwardp"[*] contains tuples of (q, r) where q is a primitive and r is the radius of the primitive when it would wrap bubble B_1 in its new position. The sign of the radius r defines the wrapping as clockwise for negative and positive for counter clockwise. The primitives in the Figure 4.10 would be organized in "forwardP"[*] by (S_1, r_1) followed by (S_2, r_2).

We would like the procedure to include the effect of segment s_1 in considering bubble B_3 as shown in Figure 4.10. For the case of the example in Figure 4.10 when called with bubble B_3 the procedure would start by checking the last entry in "forwardP"[*] first. If it is not between bubble B_3 and B_1 the entry would be skipped and the next tried. Notice that segment s_2 is not between bubble B_3 and B_1. The next to be tried would be segment s_1 which is between bubble B_3 and B_1. The procedure would return the minimum separation distance between bubble B_3 and B_1 which includes the effect of segment s_1.

The algorithm starts with the last primitive in "forwardP"[*] when passed variable $j = 0$ otherwise, j is used as the starting position. The minw is calculated for the primitive p and primitive q and if it is zero this entry is skipped. Next the primitive q is tested to see if it is between bubble B_1 and primitive p. If it is not this primitive is skipped. The minimum separation design rule, w, is the sum of the previously calculated minw plus the radius r. The index i is set to the first valid primitive in "forwardp"[*]. The maximum value of w is calculated. If no primitive was found the minw for primitive p and bubble B_1 is return in w with index i equal to zero.

We call the testing of a primitive, q, between a primitive p and bubble B_1 "covered". There are two types of primitives in "forwardp"[*] which makeup a wire. They are a segment and an around which bubble B is inside. The possible combinations of covered primitives are six which is the number of primitives times the two types of wire primitives in "forwardP"[*]. Each case will be covered next.

Before we show how the test is performed for each of the six cases an important fact about the primitives in "forwardP"[*] will be pointed out. All primitives in "forwardP"[*] will intersect the path of the bubble to be moved. We will take advantage of this in the testing for "covered". Also, we know that the two primitives, *p* and *q*, have a minw greater than zero.

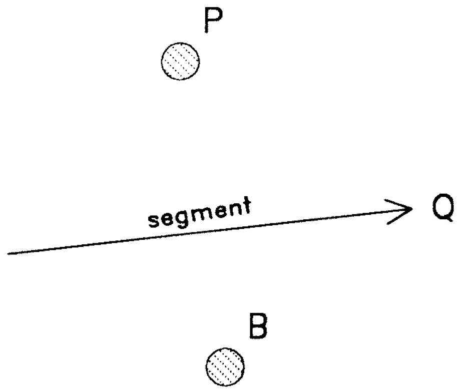


Figure 4.11 Bubble to Segment Covered Example

The first case is that of a bubble *p* testing if a segment *q* is between it and bubble *b* as shown in Figure 4.11. We know which side bubble *b* is on in relation to segment *q* from the signed radius *r* in the tuple in "forwardp"[*]. The test is to simply check if bubble *p* is on the other side.

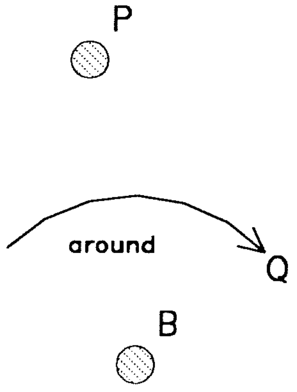


Figure 4.12 Bubble to Around Covered Example

For the case where *q* is an around and *p* is a bubble as shown in Figure 4.12 we know the bubble *B* must be inside of the around *q*. The test is simply to test if bubble *p* is not inside of the around *q*.

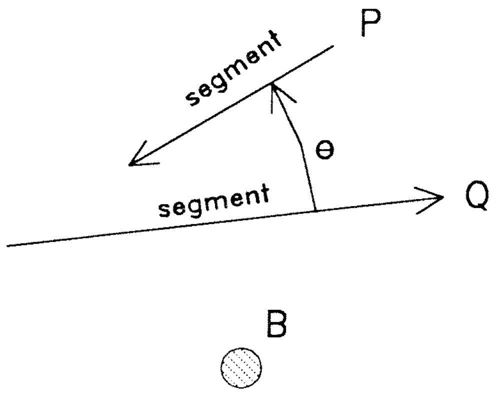


Figure 4.13 Segment to Segment Covered Example

When we have *p* and *q* as segments shown in Figure 4.13 we know the segment bodies do not intersect since the minw is greater than zero and the invariant condition is preserved. If the two segments are parallel we check for segment *q* to be closer to bubble B than segment *p* and we are done. Otherwise we compute the intersection point and test to see if the point is within the body of segment *q* . If so, we only need to check for one end point of segment *p* to be above segment *q* and we are done. If the intersection point is not within the body we compute angle θ as shown which is the angle above segment *q* to segment *p* about the intersect point. If angle *theta* is less than 90° then segment *p* covers segment *q*; otherwise it is not covered. We can use this test since we know that segment *q* is in the path of bubble B_1.

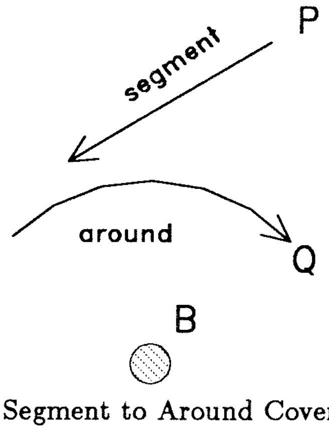


Figure 4.14 Segment to Around Covered Example

For the case where *p* is a segment and *q* is an around as shown in Figure 4.14 we test for covered by checking for the ends of segment *p* to be outside of around *q*.

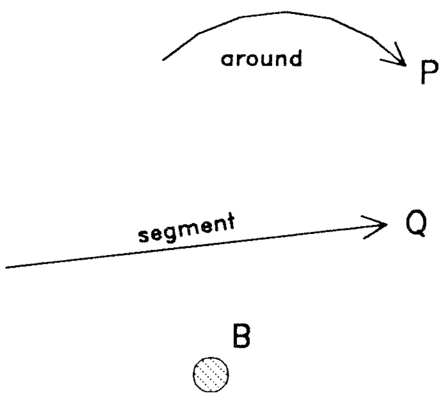


Figure 4.15 Around to Segment Covered Example

When p is an around and q is a segment as shown in Figure 4.15 we test the end points of around p with segment q. If either end point is above segment q the around covers the segment.

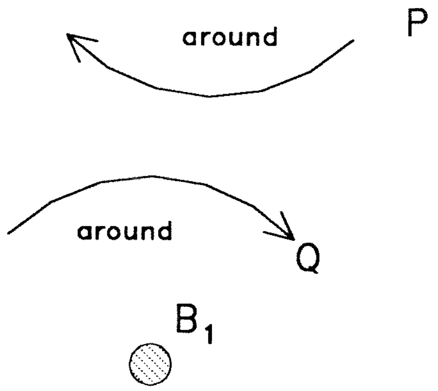


Figure 4.16 Around to Around Covered Example

The last case is when p and q are arounds as shown in Figure 4.16. If either end point of around p is outside of around q the around p covers around q.

4.1.1.2 "forwardP"[*] and "backwardP"[*] Construction

We will now describe how the "forwardp"[*] array and "backwardp"[*] are constructed. First the arounds that wrap bubble B_1 are processed by adding them to the appropriate "forwardp"[*] or "backwardp"[*] array. Next the arounds and segments in our cell u are checked to see if they intersect the path of bubble B_1. If any do they are added to the "forwardp"[*] array in the proper position.

We will now show how the arounds attached to bubble B are placed in either the "forwardP"[*] or "backwardP"[*] arrays. Notice in Figure 4.17 that if the bubble B is moved in the direction between the segment vector $\overrightarrow{S_1}$ and segment vector $\overrightarrow{S_2}$ the around A would be unwrapped. This determination can be visualized by assuming the bubble is a marble and the segments and around are hard walls. Now if the marble is moved in the direction between the segment walls, the around would be unwrapped. However, anywhere else the walls would be hit and the around would need to be moved outward. We will use this fact to add the around to the appropriate array.

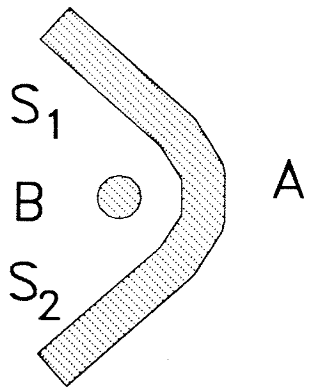


Figure 4.17 Attached Around

We will use the Boolean function "betweenaseg"(A, x, y) (see Section B.2) to determine whether to place the around in the "forwardP"[∗] or "backwardP"[∗] arrays. The function is logically true if the vector defined by the pair (x, y) is between the vector parallel to and pointing outward from the end segment of around A and the vector parallel to and pointing outward from the start segment of A. We will add the attached around to the "backwardP"[∗] array when a call on the function betweenaseg is logically true and add the around to the "forwardP"[∗] array when the function is logically false. The arounds are added by increasing radius.

Next we will test each around and segment in our cell to see whether either would cause a minimum separation design rules violation with the path of bubble B_1. If the around or segment would it is added to "forwardp"[∗] at the proper position. The search of the segments and arounds is accomplished by a loop through each around and a loop through each segment. If an intersection was found the search is restarted. The search loop is best described with a pseudo code as follows.

```
hit←True;
while not  hit do "search"
begin
     hit←False;
     for each around $a$ of U do
     if  $a$ not in forwardp[*] then
     if  insidearound($b_1, a$) then
     if  $a$ intersect path then
     begin
          add $a$ to forwardp[*]
          hit←True;
          continue search;
     end
     for each segment $s$ of U do
     if  $s$ not in forwardp[*] then
     if  $s$ intersect path then
     begin
          add $s$ to forwardp[*]
          hit←True;
          continue search;
     end
end;
```

Notice in the code that when an intersection is found the search loop is restarted. This reduces some of the search run time. Also arounds that bubble B_1 is inside of are only tested. The reason for this is that these arounds may be lifted or pushed by bubble B_1 while the others act as blocks. When the search loop has finished we have constructed the "forwardp"[*] and "backwardP"[*] arrays.

The test for an around, a, path intersection is described next. First we call Coveredprimitive(a, w, i, j)(see Section 4.1.1.1) to determine if there is a possible design rule between the bubble B_1 with the wires in "forwardp"[*] added and around a. If w is zero there is no design rule and around a can be skipped. We will save the index i for later use if the around a intersects the path.

We will first compute the intersection point x_i, y_i of the path line equation (4.6) and the around a as shown in Figure 4.18. There are two intersection points for a circle and a line equation. We will choose the furthest point from bubble B since we know the path intersect on the inside of the around. Recall that bubble is on the inside of the around a.

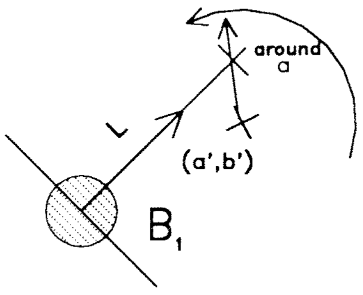


Figure 4.18 Around Path Test

We will let

$$s = a.b.x * a + a.b.y * b + c \tag{4.14}$$

which is the distance from the path to the around center. We will let

$$t = a.r^2 - s^2. \tag{4.15}$$

If t is less than zero we will continue with the next object since for this case the path would not intersect the around. We will let

$$\begin{aligned} x_i &= a.b.x - a * s - b * \sqrt{t}, \text{ and} \\ y_i &= a.b.y - b * s - a * \sqrt{t}. \end{aligned} \tag{4.16}$$

We now will check the intersection point x_i, y_i to see if it is less than the minimum separation distance w and the intersection point is within the arc of the around. We will let

$$d = a' * x_i * b' * y_i + nc \tag{4.17}$$

which is the directed distance from the perpendicular line to the intersection point. If $d < 0$ then we can skip this around since it is below the bubble B_1. If $d - l < w$ and the intersection point is within the around we have a hit.

If we do not have a minimum separation intersection from the previous test we need to check the shortest point from the around to the end point of the path. If we construct a line as shown in Figure 4.18 from the center of the around through the end point of the path to the around the shortest distance will be from the path end point to the around. We will let

$$h = abs(a.r) - \sqrt{(a.b.x - b_1.x + \Delta x)^2 + (a.b.y - b_1.y + \Delta y)^2} \tag{4.18}$$

equal this distance. IF h less than w and the intersection point of the constructed line and the around is within the around then we have a path intersection.

There is one last check to make if we do not have a path intersection. We need to check each end point of the around using the same test as was done in the clear path for the case of a bubble. We can do this since we can treat each end as a bubble.

If we have an around path intersection we will add the around to the "forwardp"[*] array. Recall the index i we saved when coveredprimitive was called. We will insert around a and its radius r which has been set positive or negative depending how around a would wrap bubble B_1 in "forwardp"[*] at i plus one. We will adjust the primitives radius above $i+1$ by using coveredprimitive with the appropriate values.

Now we will describe how the test is accomplished for the segment path intersection as shown in Figure 4.19. First we call Coveredprimitive(a, w, i, j)(see Section 4.1.1.1) to determine if there is a possible design rule between the bubble B_1 and segment s. If w is zero there is no design rule and the segment is skipped. We will save the index i for later use if the segment s intersects the path. We will next compute the sin of the angle q between segment s and the path in preparation for computing the intersection point x_i, y_j. We will let

$$q = a * s.b - s.a * b. \tag{4.19}$$

If q equals 0 than the lines are parallel and we do not have an intersection from the invariant condition. Next we compute the intersection point x_i, y_i by

$$\begin{aligned} x_i &= (b * s.c - s.b * c)/q, \\ y_i &= (c * s.a - s.c * a)/q, \end{aligned} \tag{4.20}$$

and

$$d = a' * x_i + b' * y_i + nc \tag{4.21}$$

which is the directed distance from the perpendicular line to the intersection point. If d is less than zero the segment is below bubble which can not intersect so we are done.

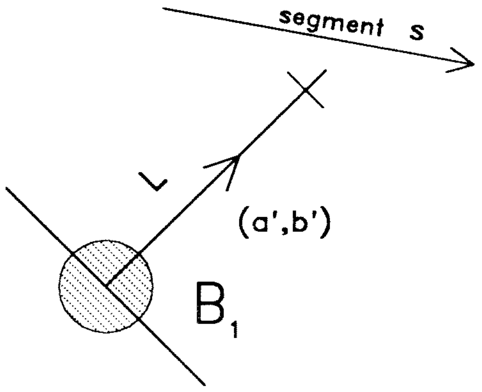


Figure 4.19 Segment Path Test

If $d > L$ then the path and segment do not intersect so we will check the final desired position of bubble B_1 with the segment. We will let

$$\begin{aligned} s &= s.a * (b_1.x + \Delta x) + s.b * (b_1.y + \Delta y) + s.c, \text{ and} \\ h &= s.a' * (b_1.x + \Delta x) + s.b' * (b_1.y + \Delta y) + s.nc \end{aligned} \tag{4.22}$$

where s is the shortest distance from the final desired position of B_1 and h is the distance from perpendicular line which goes through the final desired position to the center of the segment. If $s < w$ and $|h| <= s.l/2$ we have an intersection.

The final case is where d is less than L and greater than zero. In this case we will let

$$h = s.a' * x_i + s.b' * y_i + s.nc \tag{4.23}$$

where h is the distance from intersection point to the center of the segment. If $|h| <= s.l/2$ we have an intersection. Otherwise we test the segment end points with the path.

If we have a segment intersection we will add the segment to "forwardP"[*] as we did for the around. The sign of the radius w is determined from the side of the segment in which bubble B_1 resides.

4.1.2 Queue Construction Algorithm

At this point in time we have all the needed information in "forwardP"[*] and "backwardp"[*] for the wires that need to be pushed out and wires that are allowed to collapse. We also have the bubble B and its new position which has the attached wires that need to be moved. We also know we have a clear path. Wires in "forwardP"[*] will be processed first as waves from the last wire in "forwardp"[*] to the first wire. Next the bubble B will be moved to the new location and attached wires will be processed. Finally the wires in "backwardp"[*] from first to last will be processed.

Moving a wire can be considered as a wave sliding out from its old position to its new position wrapping any obstacle. Each wave may fall on the previous wave. Therefore we need the search list of primitives to contain the cell U with all the update being adds or deletes of primitives. Recall that we do not actually update our cell but add the changes to the addq{} and deleteq{} queues. We will define our search cell to be

$$U' = U \cap \neg \text{deleteq}\{\} \cup \text{addq}\{\}$$

which has all the primitives in the cell U with the deleteq{} excluded and all the primitives in the addq{} added. We will use U' for our processing in the queue construction algorithm. The invariant condition is preserved by the cell U'.

We will first process the wires in "forwardp"[*] from last to first. There can be two types of wire elements in the array which are segments and arounds. Recall that each element in the array contains a wire primitive and a radius. The radius defines the radius of the around to be created and the direction for the wire in its new position where the around center is bubble B at its new position. We first create a bubble B′ which is a copy of bubble B with its location at $(B.x + \Delta x, B.y + \Delta y)$. Any around we create from "forwardp"[*] will have its center at bubble B′. The newly created bubble B' is not added to the addq{} at this time since it would possibly violate the invariant condition for the cell U'.

We will first show how a segment is processed in "forwardp"[*]. In Figure 4.20 we have a segment s being moved forward to create a new segments s_1,s_2 and around

a where the radius of a is defined in "forwardp"[*]. We will divide the processing into a left and right half since we know there are no interfering objects in the center channel where the around is created from clear path. We will first show how the left side is processed where the new segment s_1 resides.

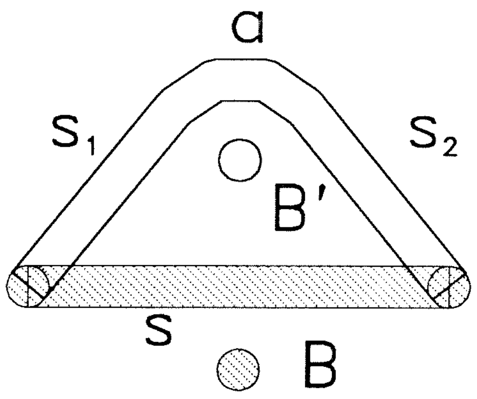


Figure 4.20 Wire Process Segment Example

We will first identify any primitives that the wire defined by segment s_1 and around a will need to wrap. Notice in the Figure 4.20 that there will not be any interfering primitive closer to the path of bubble B to B$'$ with the radius of around a. Any primitive that intersects segment s_1 or is in the region defined by segment s, segment s_1 and the path of bubble B to B$'$ shifted left by the radius of around a would need to be wrapped. Of course the primitive's minw between itself and segment s would need to be greater than zero. All others would be skipped. Notice that this region defines a triangle as shown in Figure 4.21.

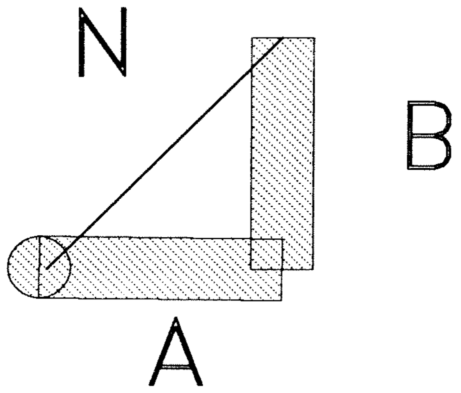


Figure 4.21 Wire Process Triangle Example

We will reduce all the processing in the queue construction to an primitive identification process and processing of the identified primitives for a pseudo triangle region as shown in Figure 4.21. The triangle will have one side which is a new segment shown as N with each end being an around or bubble. The other two sides define interior regions of the triangle. They have some width as shown. The triangle has a general form where one side N is a new segment with possible attached arounds and the other two, a and b are walls. The identification process involves searching the cell U' for any bubble or around which intersects the triangle and storing the result in a queue $q\{\}$ for processing.

The next part of the triangle process is to construct the new wire which will wrap the primitives in $Q\{\}$. This construction is accomplished by first finding the primitive in $Q\{\}$ with the smallest angle θ between a line drawn from the intersection of side n and a to the primitive extended outward by minw as shown in Figure 4.22. Next the new segment and around is created to go around the obstacle. The primitives in $Q\{\}$ which are inside the new around and not intersecting the new segment are culled from the queue. The process is then repeated until the queue is empty. We will call this process of identification and processing of primitives that intersect the triangle the "Triangle Process" which will be described in Section 4.1.2.1.

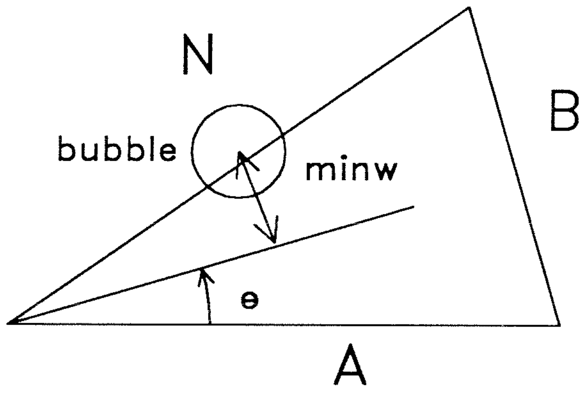


Figure 4.22 Triangle Example

You should now have a general idea of how segment s is processed in "forwardp"[*]. The actual steps are as follows: First the segment s is added to the delete queue. Next copies of s are created for segment s_1 and s_2. The around a is created which wraps bubble B' using the rule of s. The reason that the segments are copied is to preserve the related and rule information. If the ends of either s_1 and s_2, which is not connected to around a, is an around, the around is added to the delete queue, copied and linked to the appropriate segment. This copying is so that when the segments are initialized they only effect new arounds. The segments and around are initialized as defined in Section 3.2. The triangle process is then perform which adds the primitives to the add queue. This process is done for the left and right triangles.

Now you may be wondering what happens when the end is an around and needs to be unwrapped. This case is interesting since we will use what we have already built - the "Triangle Process".

Let's consider the case as shown in Figure 4.23 where we have the old segment s and the new segment s_1 where the start of the segments is an around q. We can easily detect whether this around q possibly needs to be unwrapped from the direction of s to s_1 and the direction of the around q. A necessary condition for the around to be able to be unwrapped is that the around is less than 180°. For the example shown we have met both of these conditions.

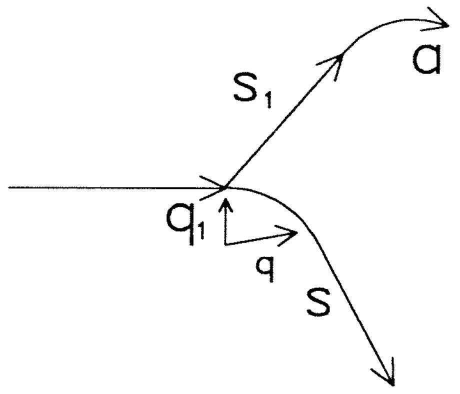


Figure 4.23 Around Setup Unwrap Example

We will proceed as before with the segment; however if the new around q_1 is unwrapped the start point for the segment s_1 will be the start point of the around as shown in Figure 4.23. We can test for the unwrap case by noting whether the new around is greater than 180°. We will set the new around's start and end point equal. This also means the angle of the new around q_1 is zero. We will now proceed as before with the triangle process for segment s_1.

At the completion of the triangle process for segment s_1 we will test the around q_1 for zero angle. If the angle is zero we will unwrap the around q_1. This unwrapping is accomplished in a manner similar to the previous example by creating a new segment s_n as shown in Figure 4.24. We now call the triangle process using the new segment S_n, s and s_1. This unwrapping process is recursive and will unwrap any arounds that needs to be unwrapped.

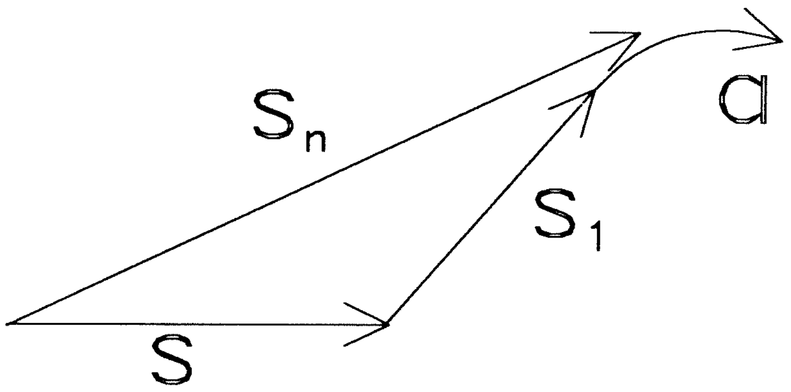


Figure 4.24 Around Unwrap Example

We will now explain how the unwrapping algorithm operates. The unwrapping algorithm is given an around q with a zero angle that will be unwrapped. The around q is culled from the add queue. If the segments, s_s and s_e, on each end of the around are on the add queue they are culled from the add queue otherwise they are added to the delete queue. A new segment s_n is created with the ends being the start of s_s and the other being the end of s_e. When the end of the new segment s_n is an around, the around is deleted from the add queue if present otherwise it is added to the delete queue. A copy of the around is created and connected to the proper end of s_n. The segment s_n is initialized as described in Section 3.2. The triangle process is used next. If either end of s_n needs to be unwrapped the algorithm is called for each.

This is a good point to summarize the algorithms we have constructed so far. We have developed a method to transform a segment into a segment -around-segment triad when the "forwardp"[*] contains a segment. We have shown how we wrap any primitives in the triad with the general method of the triangle process. We have shown a general algorithm for unwrapping an around.

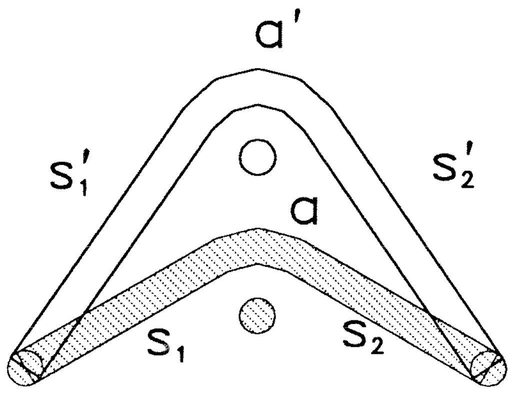


Figure 4.25 Wire Process Around Example

We will now show how an around a is processed from the "forwardp"[*] with the radius r'. We will be moving the around a to a new position which wraps bubble B' which is bubble b in its new position. All we need to do is show how the triangles are constructed for the processing of an around. Notice in Figure 4.25 we are moving around a to a new around at a'. The radius of a' is defined in "forwardp"[*]. We will first consider the cases where the radius is greater than or equal to the radius of a. The construction of the triangle for the left side will be described first. Lets assume the direction of the wire is from s_1, to a, to s_2. The triangle is defined by segment s_1, the line from the start point of around, a, to the start point of a' and segment s_1'. We can use the line from from two arounds, since we know the path of the around from a to a' is not obstructed from clear path. In a similar manner the triangle is created for the right side.

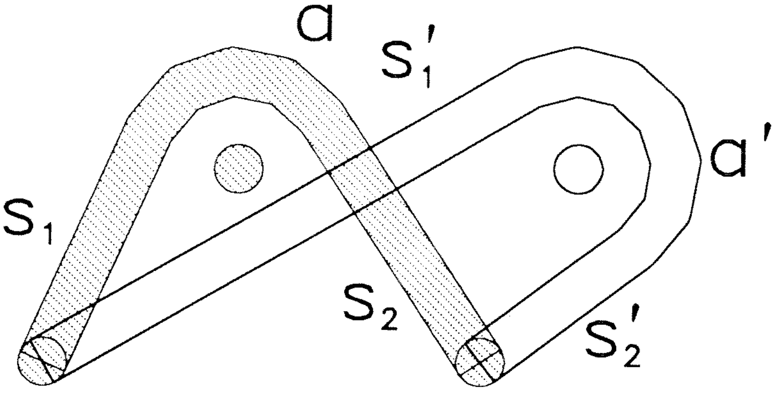


Figure 4.26 Wire Process Around Side Move Problem

This process works fine as long as the around moves up and away from bubble B. However consider the example in Figure 4.26 in which the bubble is moving to the side. If we create a triangle as previously described we will leave a hole under the old around a. In a sense we are not moving a wave but jumping the wire. Assume the direction of the wire is from s_1 to a , to s_2. We can test for this condition by checking for the end of s_1 being outside of s_1'.

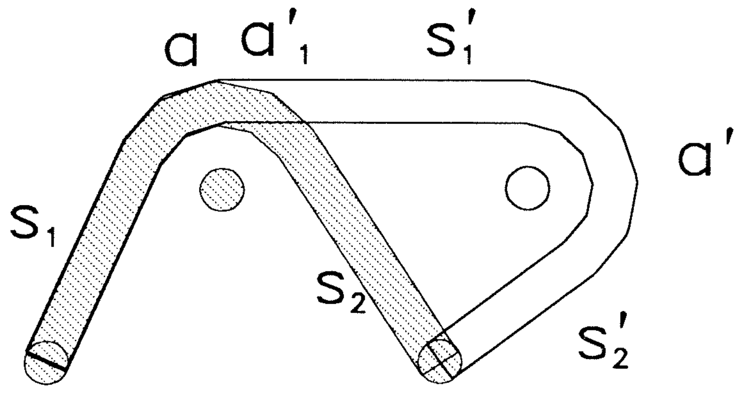


Figure 4.27 Wire Process Around Side Move Example

If the test is true we will copy around a to a_1'. We will link the start of s_1' to the end of around a_1'. This can be seen in Figure 4.27. We can do this and preserve the invariant condition from clear path. We will then add around a_1' to the "backwardp"[*] array in the proper position for later processing. Notice that now the wire will act as a wave. We will process the right side as previously described.

We will now summarize how an around a is processed from "forwardp[*] when the new radius is greater than or equal to the around a radius. First the around a, its start segment s_1 and its end segment S_2 are added to the delete queue. Next if the start of s_1 is an around it is added to the delete queue, copied and linked to s_1. Next s_1 is initialed. In a similar fashion s_2 is processed. We will first show how the left side is process. The test is performed to see if the end of s_1 is outside of the segment s_1'. If it is a copy of around a is created and linked to s_1'. The around a' is added to the "backwardp"[*] in the proper position. Otherwise the left triangle is processed and any arounds on the start of s_1 are unwrapped. In a similar manner the right side is processed.

But what about an around a in "forwardp"[*] whose radius is greater that the new around's radius? This case is a bulge out of the old around as shown in Figure 4.28. The figure shows the new around a' as a finger which is valid because of clear path. The around a' will have the radius defined in "forwardp"[*]. We will show how the left side is processed. First we test to see if the left side of the finger is inside of the around a. The test is simply to test if the end of s_1 is on the left of the path of B to b' translated to the left by the radius of a'. If it is not we proceed as we did before where the radius of a is less than or equal to the new radius.

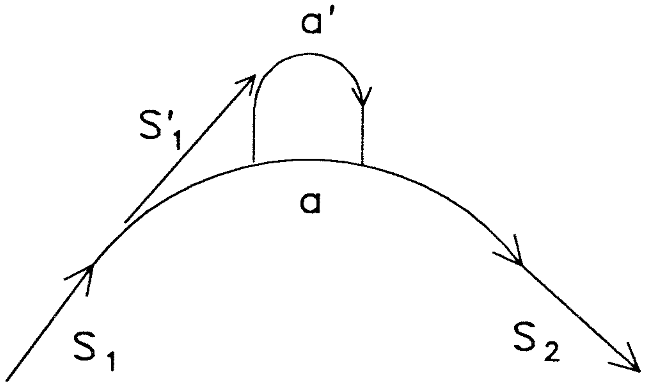


Figure 4.28 Wire Process Around Finger Problem

If the test is true the following is performed. First the around a' is created from a with new radius r'. Next segment s'_1 is copied from segment s_1. Now here is the difference. We next copy around a'_1 from a and link the start of s'_1 to the end of a'. Notice that the start of a' connects to the old s_1. We initialize the segment s_1 with the attached around a'. The triangle process is then performed using the path of B to B' as one side, the around a as the other, and segment s_1 as the last. Around a'_1 is unwrapped if necessary.

After the "forwardp"[*] we have all the wire ahead of the bubble b processed. Notice that our cell U' preservers the invariant condition. Next we process the bubble B and the attached wires.

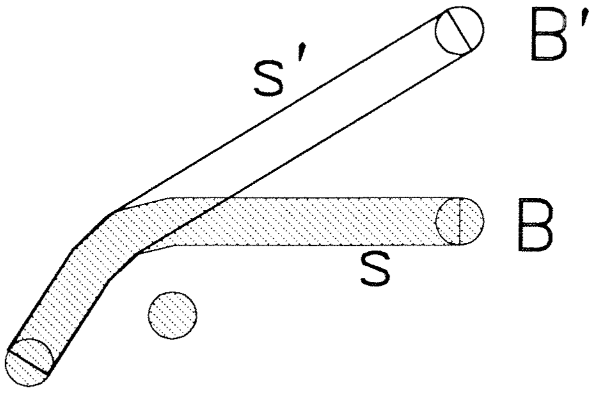


Figure 4.29 Attached Segment Process

Next the bubble B is added to the delete queue and bubble B' is added to the add queue. We will process each segment s_i which is attached to bubble B as follows. Assume the end of s_i connects to bubble B as shown in Figure 4.29. First the segment s_i is added to the delete queue and s'_i is copied from segment s_i. If the start of s_i is an around it is copied and linked to s'_i. The segment s'_i is initialized with any attached arounds. The triangle process is performed with sides s_i, s'_i and the path from B to B'. Any necessary unwrapping is performed.

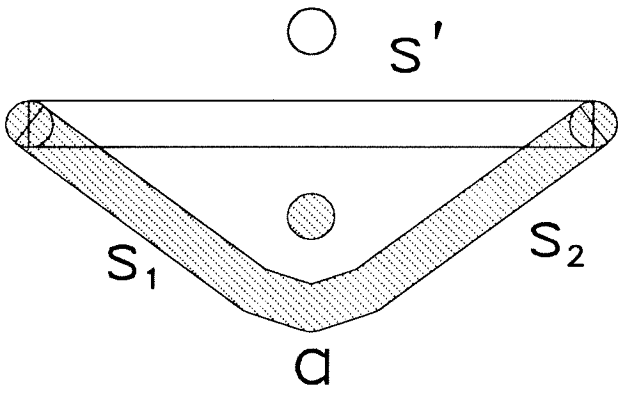


Figure 4.30 Around Collapse Example

Finally the "backwardp"[*] array is processed. We will process each around a in "backwardp"[*] from first to last. Consider the example in Figure 4.30 where a is to be unwrapped. Notice how the figure looks like the old triangle. Assume the start of the around connects to segment s_1 and the end connects to segment s_2. The steps are as follows. First the around a, segment s_1 and segment s_2 are added to the delete queue. The segment s' is copied from segment s_1. The end of s' is linked to what the end of s_2 points to. If either end of segment s' is an around it is copied, added to the delete queue, and the new around is linked to segment s'. The triangle process is then performed using the sides defined by around a and segment $4s'$. Next any necessary unwrapping is performed. Notice that if the bubble B' does not extend out of the triangle it will be wrapped.

That completes the description of the queue construction process. Notice that the cell U' preservers the invariant condition.

4.1.2.1 Triangle Process

The function of the Triangle Process is to determine if any primitives intersect the triangle and to modify the given wire to wrap the intersecting primitives. The process is naturally divided into two parts where the first is the identification process and the second is the wire modification. The given wire is in the form of a segment s_n where the segment is a new segment and not on the add queue. The only primitives that we will consider has a minw greater than zero between the primitive p and the segment s_n. All sides of the triangle can be considered to have a rule relation defined by the primitive being checked and segment s_n. The reason for this is that the sides originally were either composed of the rules of segment s_n or clear path insures there is no rule of that type.

The identification process tests each bubble b and each around a in cell U' for intersection of the triangle and storing the result in list $Q\{\}$. The test is first to determine the minw for the primitive and if it is zero the primitive is skipped. Next the primitive is checked for intersection to the triangle and if so it is added to $Q\{\}$.

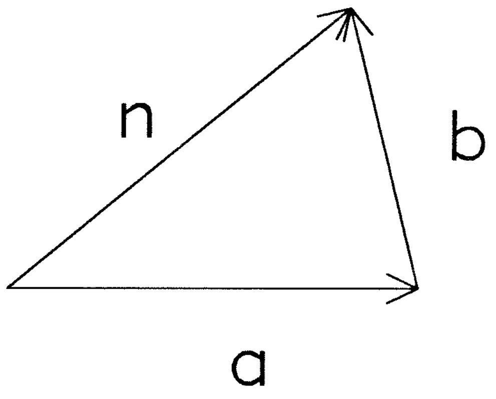


Figure 4.31 Simple Triangle

Let's take a close look at a simple triangle as shown in Figure 4.31. Without loss of generally We will assume the direction are as shown. The side n is the given segment s_n. Side a and b may be either a segment or a given line equation. The test is simple to check to see if the primitive p intersect the segment s_n or the primitive is inside of the triangle. The test for inside the triangle is a simple check with the line equations.

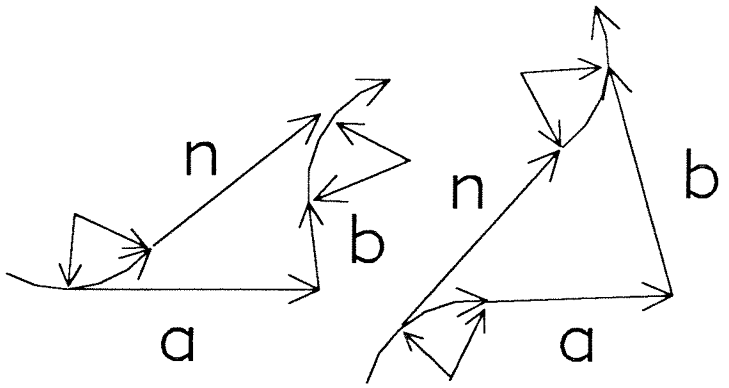


Figure 4.32 Complex Triangles

The normal case is the complex triangle as shown in Figure 4.32 where the sides may be composed of a segment and an around. This does not complicate the test since we can use the function "inside around" described in Section B.1 to test for inside of the triangle. The sides a and b may be composed of a single segment or line equation, or a segment and an around. The side n may be a single segment or start and/or end with an around. The definition of the sides of the triangle is from the caller.

We will now describe how the triangle process constructs the list $Q\{\}$. First the list is set to null. We now check each primitive p in U' which is a bubble or an around. The variable w is set to the minw between primitive p and segment s_n. If w equal zero we proceed with the next primitive. Now we test the primitive p for intersection with the segment s_n as described in Section 3.2. Also if segment s_n starts with an around it is also checked and so is the end if the segment s_n ends in an around. If we have an intersection with the segment, the primitive p and w are added to list $Q\{\}$. The list $Q\{\}$ actually contains tuples of (w,p). The distance w is the radius that the new wire will need to be away from primitive p. If we do not have an intersection we will test to see if the primitive is inside of the triangle.

We will now describe how we test if a primitive p, which is either an around or a bubble is inside the triangle. First we know the primitive is either totally outside or totally inside of the triangle. This is because from clear path and the initial invariant condition it does not intersect sides a and b. Also we have just tested for side n. Therefore we only need to check if one point of the primitive is inside of the triangle. If the primitive p is a bubble we will let the point $v = (x,y)$ equal the center of the bubble; otherwise we will let the point $v = (x,y)$ equal one end of the around p. Now we only need to test if the point v is inside of the triangle by testing with each side. When we test with a side that has an around the function "inside around" is used. If the following tests for each side is true the primitive p and distance w are added to the list $Q\{\}$. The test for each side is as follows.

Side n) If the segment s_n starts with an around the point v will need to be outside of the around. If the segment s_n ends with an around the point v will need to be outside of the around. If neither end is an around the distance from the segment s_n to the point v will need to be positive.

Side a) If side a is a segment and starts with an around the point v will need to be outside of the around; otherwise the distance from the directed line a will need to be negative.

Side b) If side b is a segment and starts with an around the point v will need to be outside of the around; otherwise the distance from the directed line b will need to be negative.

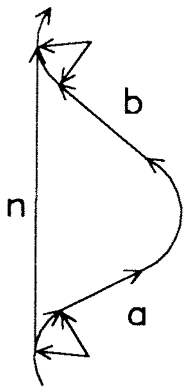


Figure 4.33 Special Triangle

There is a special triangle that comes from the processing of the "backwardp" [*] array where the intersection of the sides a and b is an around as shown in Figure 4.33. In this case the testing of sides a and b is modified to test if the point v is inside of the around of intersection of a and b. There is a possible hole if side a starts with an around and/or side b ends with an around as shown. We will only describe one side since they are similar. If the point v is outside of the intersection around and side a has a start around we test distance from the point v to the directed side a for greater than or equal to zero and the point v to be outside of side a start around.

If the list $Q\{\}$ is empty we are done. If the list has one primitive p we copy segment s_n as s'_n. We create a new around a. If primitive p is a bubble the radius of a is w with the center at bubble p. Otherwise the primitive p is an around in which case the radius of a is w plus the radius of the around $|p.r|$ where the center is bubble $p.b$. We then link the end of s_n to around a, and the end of a to s'_n. Initialize the segments and around. Finally we add them to the add queue.

For the case where the list $Q\{\}$ has more than one primitive it may be necessary to wrap one or more primitives. The algorithm that we will describe next is repeated until the list $Q\{\}$ is empty. First let the primitive p with distance w be the first primitive with the minimum pivot angle. We will describe how the pivot angle is computed after the description of the algorithm. We create a new around a. If primitive p is a bubble the radius of a is w with the center at bubble p. Otherwise the primitive p is an around in which case the radius of a is w plus the radius of the around $|p.r|$ where the center is bubble $p.b$. The sign of the radius of a for the triangle shown will be positive; if the triangles are mirrored then the sign will be negative. We then link the end of s_n to around a, and the end of a to s'_n. Initialize the segments and around. We add s_n and around a to the add queue. We next let s_n be s'_n. The elements in list $Q\{\}$ culled if they are inside of the around a and do not intersect segment s_n nor around a. If the list $Q\{\}$ is empty we are done, in which case we add s_n to the add queue. Otherwise we repeat.

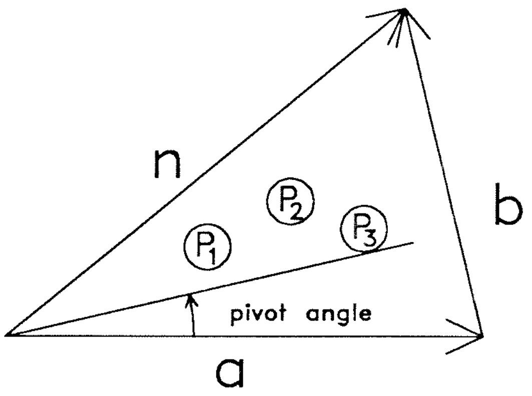


Figure 4.34 Pivot Angle

Now the question is, what is the pivot angle and why should the primitive with the minimum pivot angle be the first primitive to wrap? Consider the Figure 4.34 where we have possible wrapping candidates P_1, P_2 and P_3. We can assume the walls a and b are made of the rule of s_n. Now let's slowly let wall a rotate in about the intersection of side n and a. Notice the first primitive to be touched will be p_3. The pivot angle is the angle between side a and the rotating side. The primitive P_3 will have the minimum pivot angle. We could use either of the sides for rotation.

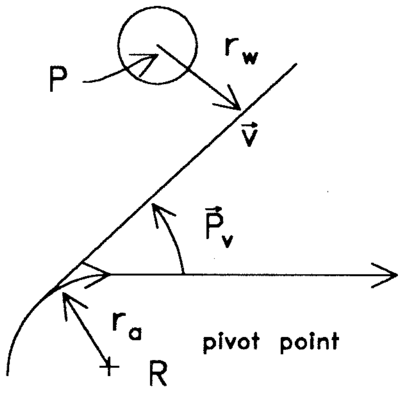


Figure 4.35 Pivot Angle Computation

In the previous example the rotating side was about a point; however the end of the side could be an around where the side is rotating along an around. The pivot angle $\vec{p_v}$ is the angle between the resting side shown as s and the side rotated out until it is r_w away from the primitive P. The pivot angle will be represented by a vector. The distance r_w is the clearance w plus the radius of the primitive if it is an around. We will let

$$\vec{p_v} = \vec{v} * \overline{\vec{s}}. \tag{4.24}$$

The vector $\overrightarrow{s}$ is easily calculated by rotating the line equation vector of side s by 90°.

$$
\text{We let}\quad
\begin{aligned}
\Delta x &= P.x - R.x, \\
\Delta y &= P.y - R.y, \\
r_w &= w + \text{plus radius of P if an around}, \\
r &= r_w + r_a, \\
N &= \sqrt{\Delta x^2 + \Delta y^2}, \\
P_d &= \sqrt{N^2 - r^2}, \\
\overrightarrow{q} &= (p_d/N, r/N).
\end{aligned}
\tag{4.25}
$$

The distance p_d is the distance from the point of rotation to the intersection of the rotating side and the primitive P. The pivot point R may be either an around bubble where r_a is the radius of the around which may be positive or negative or a bubble where r_a is zero. The vector $\overrightarrow{q}$ is the inside angle of the the vector $\overrightarrow{RP}$ and vector $\overrightarrow{v}$. We will let the pivot angle be

$$
\begin{aligned}
\overrightarrow{v} &= \overrightarrow{RP} * \overline{\overrightarrow{q}}, \\
\overrightarrow{p_v} &= \overrightarrow{v} * \overline{\overrightarrow{s}}.
\end{aligned}
\tag{4.26}
$$

Now the question may arise, what if the pivot angles are equal for two primitives? Then we use the distance P_d as the qualifier. We will take the primitive with the maximum p_d first.

4.2 Utility Routines

The utility routines are two functions that are constructed from portions of the bubble move algorithm. The routines do not modify the data structure proper but modify the add and delete queue. The routines will be used in the graphic editor and other functions.

4.2.1 Around Collapsed

The around collapsed algorithm allows a wire being held by a bubble to be released and snap inward wrapping any object in its path as shown in Figure 4.36. In this example the filled in wires are before the around collapsed and the outline wires are after. In the example we removed the bubble which allowed the wires to collapse. We have just demonstrated a use for the algorithm in the deletion of a bubble. Another of the uses of the algorithm is when the geometric design rules are modified as described in Section 3.7.2 and the around radius needs to be altered. If we delete a wire with an around wrapping the wire we would also need to unwrap the wrapping wire.

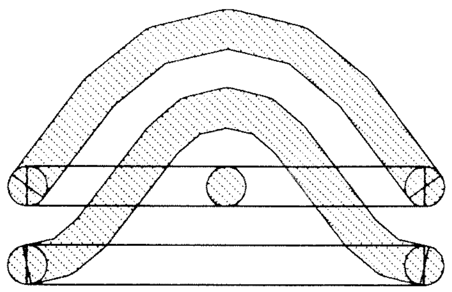


Figure 4.36 Around Collapsed Example

The algorithm for the around collapse is straight forward in that it will use the algorithm for processing of the "backwardp"[*] as described in Section 4.1.2. The first step is to load the "backwardp"[*] with the desired arounds to be collapsed. The arounds are stored in the proper order in the array. Next the "backwardp"[*] array is processes as described in Section 4.1.2.

4.2.2 Bubble Absorption

The bubble absorption algorithm will remove a bubble that is an intermediate point in a wire. The bubble can not be a model bubble and the two connected wires must have the same rule as the bubble. An example of a bubble absorb is shown in Figure 4.37. The filled in wires are the before absorption where the outlines are the after. Notice that there is a bubble below the bubble to be absorbed. Also notice that there is a wire wrapping the bubble.

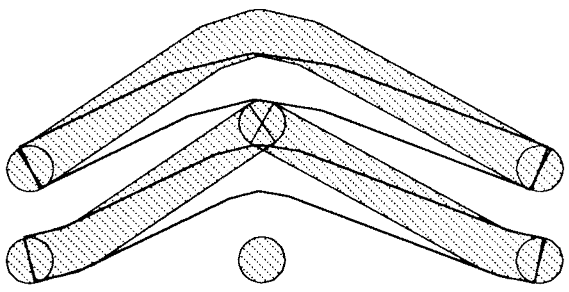


Figure 4.37 Bubble Absorption Example

The algorithm first checks if the bubble is not a model bubble and has two connecting wires with the same rule as the bubble. Next the "backwardp"[*] arrays is loaded with the arounds attached to the bubble in the proper order. The bubble is added to the delete queue. The unwrapping algorithm described in Section 4.1.2 is used to merge and collapse the two attached wires. This algorithm can be used since the intersection of the bubble with the two wires is the same as a zero degree around. Then the arounds in "backwardp"[*] are collapsed as previously described.

4.3 Cell Perimeter Modification

The cell is enclosed in a bounded polygon with the ports on the edge of the polygon. Let's assume that the polygon is a rectangle although the algorithm presented will work on a polygon with the proper checks. We would like to be able to expand or contract the cell. This process is accomplished by moving one edge at a time. The edge will move parallel to its old position. The ports are maintained at the same position on the edge. Figure 4.38 show a cell with expansion and contraction.

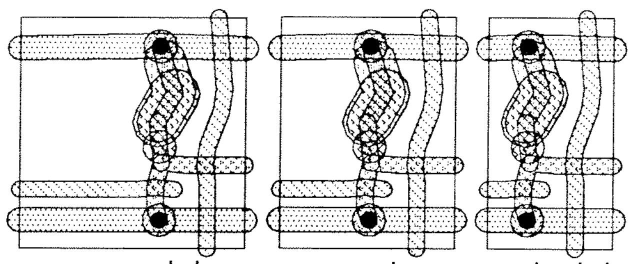


Figure 4.38 Example of Cell Expansion or Contraction

The reason the ports are maintained at their position of the edge is to be consistent with expansion and contraction, and aid in cell composition. Suppose we wish to compose two cells A and B. Assume cell A connects to cell B on the right and cell B connects on the left. Let's also assume there is a one to one port match with location and rule. Therefore we can compose cell A and B on there right and left edge respectively. We would like to add some more logic to cell B while being able to preserve the ability to compose with cell A. This addition is easily accomplished by expanding cells B on the left, adding the logic, and contracting the cell to minimum size.

Another use of the expansion and contraction is to create a routing cell. We could create a routing cell R with wires from port to port where the ports are positioned for proper composition. We could than reduce the size of the cell by contraction to minimum size.

4.3.1 Cell Edge Expansion

The expansion process algorithm is accomplished by creating a new side S_{new} as shown in Figure 4.39 and deleting the old side. Next new ports are copied from the old port to preserve rule and other information. The new ports are placed on the new side on a line perpendicular to the new side and intersecting the old port at the center. Next a wire is connected from each old port to the copied new port. The old ports are then absorbed as described in Section 4.2.2.

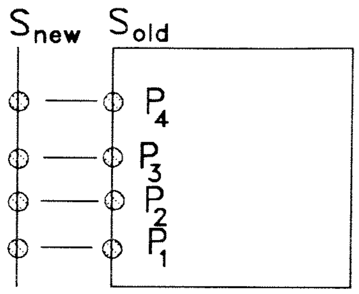


Figure 4.39 Cell Expansion

4.3.2 Cell Edge Contraction

The cell contraction will use the bubble move algorithm as previously described. First we create a new side S_{new} as shown in Figure 4.40 at the desired contraction position parallel to the old. Generally this position is at the minimum cell boundary which also limits the position of the new side. We label one end of the side as bottom as shown. Next we create an ordered list $Q\{\}$ of tuples (P, s, d) where P is a port on the side, s is the distance from the port to bottom, and d is the distance of intersection point of the connected wire to the port and the desired new side with bottom. The list is ordered by the increasing distance of the port to bottom.

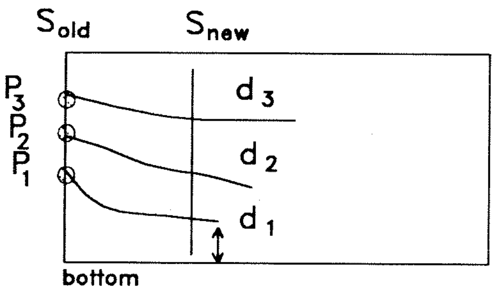


Figure 4.40 Cell Contraction

We will use P_i as the i port in the list, s_i as the i port distance in the list and d_i as the i distance in the list. We start with the lowest order port P_i which is on the old side where $d_i + 1$ is greater than d_i by the minw of P_i to $P_i + 1$ or the last port. This choice is so there is room to move the port without the bubble move algorithm failing on the above ports wire. The port is then moved to S_{new} if possible or as close as possible with bubble move. If the port is not able to contact the side S_{new} the new side is moved to the port and the list is adjusted appropriately. Next all the bubbles in $Q\{\}$ below i are move similarly starting with $i - 1$. This process is repeated until all the ports have been moved or the new side S_{new} is coincides with S_{old}.

At this time we have all the ports moved to or beyond the new side S_{new}. Notice that we have moved the new side as close as possible to the desired new side position. Next we proceed like the expansion for the ports that are not on the new side by copying the port to the new side, adding a strap, and absorbing the old port.

It is important to realize if the desired new side is on the minimum cell boundary the algorithm just presented will put the new side as close as possible, to the desired side without violating the invariant condition. This method is the way a cells boundary is reduced to minimal size while preserving relative port position.

4.4 Implementation Considerations

As you may suspect a large portion of time is spent in searching for primitives. We would like to minimize the search time. We can compute a bounding box for the triangles and clear path search area and only test those primitives that intersect the bounding box. This would eliminate some checking but the order would be $O(N)$ where N is the number of primitives.

The problem of reducing the search is very simple if we have a fixed object size and there are fixed locations. However objects can be varying sizes and at any location in the cell. In addition we are searching for all the primitives within a variable size area. We will use a ternary tree with a left L, center C and right R links. The top node will divide the cell into two part where objects on the left of the center cut line would be held in the tree in the left link, objects intersecting the cut line would be store in the center link, and objects on the right of the cut line would be in the right link. Each link will point to another tree which would again split the area in half or a list of objects. Assume we have a uniform distribution of objects, and the ternary tree partition down to the average grain size. We could search for an area of average grain size in order $O(\log(A))$ where A is the area of the cell.

Each tree node will be a tuple of $(S_d, V, L, C, R, C_l, C_h)$ where S_d defines the cut axis, V defines the cut ordinate, L,C, R are the node links, and C_l,C_h are bounds on the center node. The bounds on the center node limit the searching of the center tree to those objects that intersect the center channel. Each node may divide the cell area in the x-axis or y-axis. Each node splits the remaining area orthogonal to the distance of the maximum axis.

The actual search could be for a small area as a grain size or a much larger area. Assume the search area is a_s and the total cell area is A. The total number of elements is N. Then the search time would be $O(\log(A)+k*n)$ where n is $N*a_s/A$.

The implementation consists of two search routines. The first is for initializing the search with a search bounding box or a null box for traversing the whole tree. The second routine returns the next element in the search space or null for exhausted.

One question that is asked is what does it cost for an update on the tree? The elements are stored as doubly linked objects. For a delete the links only need to be adjusted. For an add it's order $O\log(A)$.

5

Graphic Editor

This chapter presents an interactive graphic editor for cell design. The editor is a physical editor in that the placement of objects relative to one another has real geometric meaning. The invariant condition as described in Section 3.3.4 is preserved for the editing of cells. Direct immediate feedback is given to the user upon an error condition. The editor has only a few flexible and powerful commands.

Generally most cell designers start with a hand sketch of the cell they are working on. This sketch is usually done on grid paper in scale using the technology design rules. The sketch serves as an editing medium. Even though there are graphic editors available to the designer they usually start with a paper laid out cell. The reason is that the graphic editors used do not have the commands nor the editing capabilities that pencil and paper do. We would like our graphic editor flexible enough that designers will choose to start with the editor rather then pencil and paper.

Before we discuss the classes of editors it is important to consider how the design process is done for cells. First an initial design is created. Next the design is updated many times until the desired cell is achieved. Notice that it is not too important to optimize the initial cell design process. However the update process needs to be optimized since it is done many times and requires the larger amount of user time. The cost of an update should be proportional to the amount of change and the editing functions should be tuned for updating the design.

There are two general classes of cell editors. The first is a symbolic relative graphic editor. With this class of editor the user enters the cell symbolically specifying the topology. Generally this editing is done with a **stick**[Williams 77] type of representation. Thin stylized lines are used to represent the wires where transistors are defined by the crossing of the appropriate type of wires. The editor digests the stick input and creates an internal representation which then can be compacted. The advantage of this class of editor is in the update process which is the most important. Circuit elements can easily be added between existing elements without consideration of design rule errors.

One of the early symbolic editors to combine the editing function with compaction is **Rest**[Mosteller 81]. With this editor a user can enter or alter a cell, compact the cell, and get immediate feedback on the compactness. Appropriate adjustments to the cell are then done and process is repeated. The advantage of this type of editor is the user is coupled with the compaction process. The **MULGA**[Weste 81] system uses a virtual grid for the editing specification. With this system the cell is laid out on a virtual grid. The compaction process can then be utilized. The advantage of symbolic editors is that the user is freed from the task of considering design rule and compacting the cell. Updates to an existing design is very easy.

Cell editors of the second class of cell are called physical editors. Example of early editors of this class are **ICARUS**[Fairbairn 78], **Cell Design System**[Franco 81], and **Caesar**[Ousterhoust 81]. With these editors mask geometry is entered in physical form with meaningful dimensions. Each mask layer must be individually entered in the $2\frac{1}{2}$ D–plane. Updates to geometric elements are by each object on a mask layer. The geometric objects are in the form of **CIF**[Sproull 79] where objects are specified by boxes, wires, and polygons. The geometric objects are limited to orthogonal shapes. The Caesar system only allows boxes. The Cell Design System has the concept of a transistor path where transistors are specified like a wire with a series of points. The drawback with this class of editors is that there is no design rules checking. These editors are used by first editing a cell, running a time consuming batch process design rule checker, and then correcting the mistakes. This process is iterated until the desired cell is reached. Another drawback is that the update process is tedious at best. Insertion of additional logic or modification to a hand compacted cell requires extensive movement of elements.

The next generation of physical editors couple the design rule checking process with the editor. This technique allows feedback on design rule errors when they are created. Some of the frequently used circuit elements are combined into a single symbol or a logical layer for easy manipulation. The **Magic**[Ousterhoust 84] system is a primitive element editor of this type. The Magic system provides feedback during the editing process where the design rule errors are displayed. Then it is up to the user to correct the errors. The editor is similar to the Caesar editor in that all modification is via single layered boxes. This editor provides a function for moving obstacles called "plowing" which is similar to pushing a group of objects out of the way. The **Tigger**[Whitney 85] editor is similar to Magic in that it provides feedback on design rules violation. Unlike Magic, Tigger does not allow Design rule violations and modifications are accomplished at the symbolic level. Tiggers input is in the form of paths for wires and transistors. Contacts are created with a simple command.

We are limited by the nature of our **structure** and the invariant condition described in Section 3.3.4 to a physical editor. This restriction is not a drawback like previous editors since we will provide immediate feedback during editing functions that would produce a design rule error while not allowing the error. The extremely powerful **bubble move algorithm** described in Section 4.1 will be provided as an editing function and can also be used to construct additional powerful commands. Unlike previous physical editors that need to describe individual mask geometry pieces, we will only allow the higher level primitives of wires, bubbles and models. Coupled with a compactor we have an extremely powerful editor.

5.1 Editing Frame

The editing frame provides the user "viewport" as shown in Figure 5.1. The implementation uses a device independent graphic package with multiple widow and menu capabilities. The frame is divided into two parts consisting of the editing view of the cell and the menu's. There are two scrollable menus where the scroll bar shows the visible portion of the menu. The top menu contains the editing functions while the bottom menu contains the rules and models defined from the technology file. The order of the items in the menu and the function that the rules and models can be used for is described in the technology file.

The graphic terminal has a pointing device with three buttons usually called a mouse. We will call the left button the ***select*** button which is used for selecting objects, the center button is called the do it button for causing action and the right button is called the ***mark*** button for marking a point.

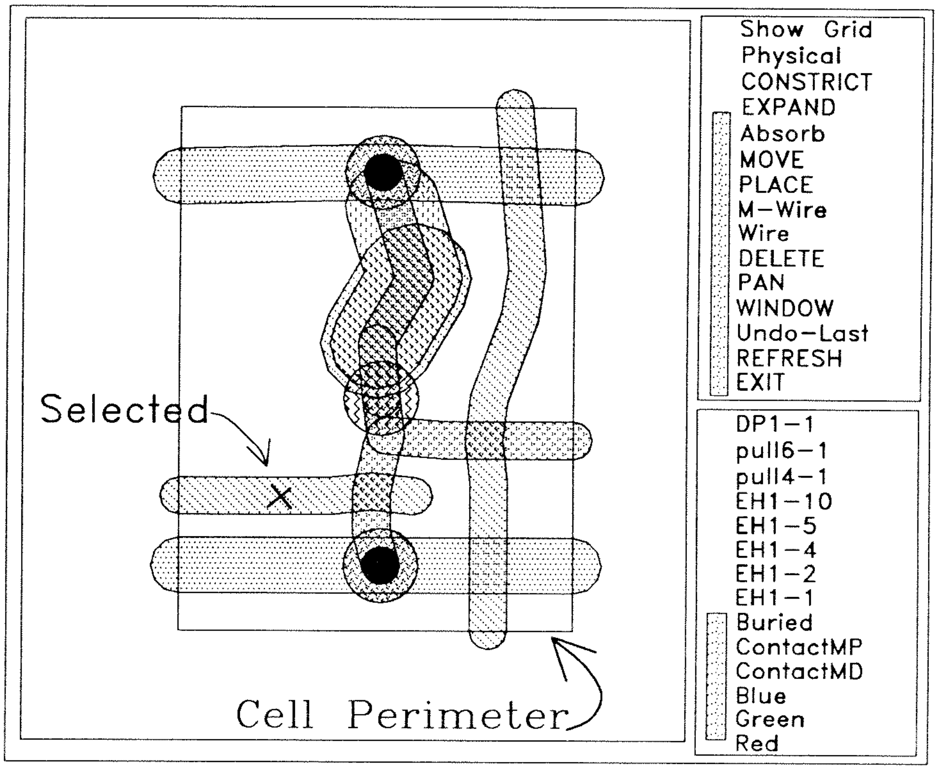


Figure 5.1 Editing Frame

Items in the menu are selected by pointing at one and pressing any mouse button. The selected menu item is then highlighted while the previous one is shown as a normal item. A menu may contain sets of functions where only one item may be highlighted in a set. The menu may also contain Boolean items where the Boolean is highlighted in the true state. The menu is scrolled by using the center button pointing within the scroll bar area at a new position for the scroll bar. This action causes the menu to scroll. The processing of the menus is handled by the graphic package.

The editing cell is shown to the user in a window where the perimeter of the cell is highlighted. This can be seen in Figure 5.1. Editing can only occur inside of the cell perimeter. Ports are the only exception. The models and bubbles are shown in a physical form where the connecting wire may be shown as lines or as fleshed out wires. This control is an option in the top menu called *physical.* The colors displayed for each primitive are defined in the rule for the primitive as the physical colors see Section 3.3.2.

There are three Booleans for controlling the response within the cell window. The first, we have just talked about which is *physical.* The second is *snap-to-grid* which causes any input point to be snapped to a user definable coarse grid which is greater than the fine grid defined in the technology file. The window can have a grid which appears as dots on the screen. The grid is turned on or off with *showgrid.*

We have the concept of a selected primitive. A primitive is selected by positioning the cursor on top of the desired primitive to be selected and pressing the select button. An χ will then appear on the selected primitive. An example is the selected wire in Figure 5.1. The selected primitive has the primitive p and the selection point. The point is the center of the χ. For bubbles the selected point is the center of the bubble. For wires the selected point is on the path of the wire. The selected primitive is used by the editing functions. The allowed set of selectable primitives is controlled by the editing function.

There are two viewing functions that the user can use to get a closer look at portions of the cell or to move the window across the cell. The functions use a stack of windows so that the previous one may be popped back to. The window function is used to defined an area within the current editing window for the next editing window with a box defined by the *mark* and *do it* which of course causes the windowing to take place. The pan function allows moving the current window across the cell. The previous window is returned to by the use of the select button.

An important point about the coupling of a graphic editor and a graphic package is the updating of the screen. Only the portion of the cell that has changed should be updated on the screen. This technique is done to give the user quick response. We would not want to refresh the entire screen each time a change is made. This screen update is easily accomplished with the aid of the graphic package and the uniform transaction process discussed in the next section.

5.2 Uniform Transaction Process

We would like all of the modifications to the cell to be processed by a common function. The reason for this is to have a uniform error handler and allow for ease of extensibility. This process would need to do a check for an invariant condition violation with the objects to be updated. We call this process the uniform transaction process since updates are presented as transactions in the add and delete queue described in the bubble move algorithm.

The cost to accomplish the update and checking should be proportional to the amount of change. We will only need to check the new items in the add queue with the cell, U, and any possible broken related network defined by objects in the delete queue. Any detected invariant condition errors will be presented to the user is a reasonable manner.

To be able to perform the checks we need a cell, U', that has been updated with the changes. The reason for this is so that the related networks will reflect the update when minw is used for the design rule check. We will first initialize a queue $C\{\}$ called the checking queue with all of the primitives in the add queue. Next each end bubble B of a wire in the delete queue is processed as follows. If the bubble B is in $C\{\}$ or in the delete queue it is skipped. Next B is added to $C\{\}$ and each wire w connected to B not in $C\{\}$ nor in the delete queue with the same related information is added to $C\{\}$. End bubbles of the wire w are processed in a similar manner. We now have a complete check list.

The editing cell, U, is then updated with the add and delete queue to create U'. The update process alters the related information by clearing the related networks in the delete queue and then updating all uninitialized related networks. This update is a linear process on the number of changes in the add and delete queue.

The invariant condition checking is now performed for each primitive p in the queue $C\{\}$ and the cell, U', by using the method outlined in Section 3.2. When and if an error is detected, a triangle is placed on the primitives that caused the error. The size of the triangle is based on the widths of the primitive. The location on the triangle for a bubble is in the center. A segment will have the triangle on the centerline of the segment closest to the error. The around will have the triangle on the centerline closest to the error. This display gives the uses a clear perspective where the errors are. The computation time is proportional to the size of the change where the order is $O(n * \log(A))$ where n is the number of changes and A is the area of the initialized cell. The searching of the cell for primitives uses the method outlined in Section 4.4.

If no errors were detected the cell U becomes U'. The screen is updated in the region of the changes. The add and delete queue are saved on a *whoops* stack. This method is extremely important since most of us are not perfect and we sometimes would like to undo the last few changes. The size of the stack is user definable.

If there were errors the cell U' is restored to U by undoing the add and delete queue and discarding the queues. The error triangles are left on the screen until a select or a new function is used.

The bubble move algorithm does not need to go though the checking phase since the invariant condition is preserved by the algorithm. The triangle error flags are produced for the move. The *whoops* stack also contains the move history.

5.3 Editing Functions

The cell is modified with the editing functions. A function is selected by placing the cursor on the desired function in the menu and pressing any button. The desired function is then highlighted and ready for use.

An editing function may require a rule or model where an example is the place function. The rule or model that is used is the currently selected rule or model in the technology menu. An item in the menu is selected by placing the cursor on the desired one and pressing any button. How the selected rule may be used is defined in the technology file described in Section 3.3.2.

The controlling of the selectable primitive is governed by the functions. Normally all primitives are selectable. Any exception will be described in the description of the function. For example when the wire function is used only primitives that are wirable are selectable.

All functions except expand and constrict go through the uniform transaction process.

5.3.1 Undo Last

The ***undo last*** is one of the most important editing functions. Generally most people do not make the correct modification to the cell the first time. Therefore they may like to remove the last change or more. The ***undo last*** removes the last change by undoing the last entry in the whoops stack. The size of the stack is user definable.

The whoops stack also contains the selected primitive when the stack entry is constructed. The ***undo last*** also restores the selected primitive if there was one.

5.3.2 Place

The place function is used to create an instance of a bubble, port or model. The orientation of the object to be placed is defined by the vector from the ***do it*** point to the ***mark*** point. The pressing of the ***do it*** button causes the execution of the place command. If a rule is selected a bubble or port is created, or if a model is selected a model instance is created. The Edplace Boolean in the rule must be set for a bubble to be placed while the Edport Boolean must be set for a rule to be used as a port.

First we will describe how a bubble or port is placed. If the ***do it*** point is on the perimeter of the cell a port is created; otherwise providing the ***do it*** point is on the interior of the cell, a bubble is created. The description for the actual bubble place is the same for a bubble or port. A bubble B of the appropriate rule is created at the location of the ***do it*** point and added to the add queue. If bubble B is on top of a compatible bubble we would like B to replace the bubble. Also if the bubble B is on top of a compatible wire we would the wire to connect to the bubble B.

We will now describe how bubbles are tested for replacement. The set of bubbles are searched for a bubble Q that is closer to the ***do it*** point than the nominal width of Q and the rule colors of the bubble Q have any intersecting colors with the rule colors of bubble B. If there is a bubble Q, the attached wires to Q will be checked for compatibility with bubble B. Wire to bubble compatibility mean the wire can connect to the bubble as defined in the rule by the connection[*] Boolean. If the wires are compatible, the bubble Q is added to the delete queue along with the attached wires. All wires connecting to bubble Q are then copied and added to the delete queue. This process is then repeated since a bubble could be added that would be compatible with many bubbles. An example of this would be placing a red blue contact on top of a red and blue wire.

Wires are then checked for possible connection to bubble B. The set of wires is searched for any wire w that is closer to the *do it* point than the nominal width of the wire. The wire w is then checked for compatibility with B and if the wire is allowed to connect to B as defined by connection[*] Boolean in the rule. If so, the wire w is added to the delete queue and copied into two wires w_l and w_r, where the end of w_l points to bubble B and the start w_r points to bubble B. This process is then repeated until the list of wires is exhausted. We do not actually have wires, but segments and arounds. The process is exactly the same for segments and arounds.

To place a model, first an instance of the model is created at the *do it* point with the orientation defined by the vector from the *do it* point to the *mark* point. Each instance bubble is processed exactly the same as placing a bubble. The instances segments are then added to the add queue.

After the primitives have been processed by place, the uniform transaction process is called to check for errors and to update the cell, U. The new selected primitive is then set to the last placed bubble.

5.3.3 Wire

The purpose of the wire function is to connect two primitives by a wire with the selected rule. Of course the Edwire Boolean must be set for the selected primitives rule. The start of the wire is defined by the selected primitive. If there is no selected primitive a bubble is placed at the *do it* point with the selected rule as defined by the place function. Only primitives that are connectable by the selected rule are selectable when the wire function is in effect.

If the selected primitive is a wire, the wire is split into two wires with an intervening bubble that become the selected primitive. The wire split location is the selected point. The old wire is added to the delete queue while the new bubble and the two news wires are added to the add queue. Of course the appropriate flags must be set is the rule for the wire. The rule for the intervening bubble is the rule of the wire. Model instance wires may not be split.

We will now connect a wire from the selected bubble to the closest connectable primitive to the selected bubble which intersects the line segment defined from the selected bubble to the *do it* point if one exists. This process is done by searching the bubbles and wire for the closest primitive p that intersects the line segment which has the connection[*] Boolean true for any color in the selected rule. Model instance wires are skipped since they can not be split. If the primitive p is a wire it is split as defined previously and the primitive p is set to the intervening bubble. A wire is created with the selected rule that connects from the selected bubble to the bubble p which is added to the add queue.

If a primitive p was not found, a bubble is created with the selected rule and add the connecting wire. The end bubble of the newly created wire becomes the selected point. This allows a path to be created by a continued press of the *do it* button. The uniform transaction process is then performed.

5.3.4 M-Wire

The M-Wire function is exactly the same as the wire function except that the start point of the newly created wire is absorbed if possible as described in Section 4.2.2. This command allows creation of a tightly stretched wire from a start bubble to an end bubble without any intervening bubbles.

5.3.5 Delete

The delete is used to delete the selected primitive. If the selected primitive is a member of an instance model the whole instance is deleted. If the primitive is a bubble, the bubble and all connecting wires are deleted. For wires the whole wire is deleted and any wire that wraps the deleted wire is allowed to collapse or rubber band in.

We will first describe how a wire is deleted. First, the wire composed of segments and arounds are added to the delete queue. Next, if the wire wraps any bubble each bubble B is processed as follows. If the bubble is on the delete queue, it is skipped. The arounds that wrap the bubble that are not deleted are processed by allowing the around to collapse as described in Section 4.1.2. This collapsation is done to remove the space between wires that wrap bubble B and the wire deleted wire. Also this is necessary to preserve the invariant condition.

Next we will describe how a bubble B is deleted. First the bubble B is added to the delete queue. Next any connecting wires are deleted as previously described. The arounds that wrap bubble B are now collapsed as described in Section 4.1.2.

If the selected primitive is a member of an instance model all primitives of the instance are delete as previously described. This process is easily accomplished by tracing through the links of the instance.

The uniform transaction process is then performed to process the delete.

5.3.6 Move

The move allows the moving of the selected bubble to a new position described by the *do it* point. This command is one of the more powerful functions. Any wire in the path of the move will be pushed out. If the selected primitive is a wire, the wire is split at the selected point as described in the wire function. The intervening bubble becomes the selected bubble.

The move function is processed as follows. First the bubble move algorithm described in Section 4.1 is used to create the add and delete queue. If the move can not be performed the interfering bubbles are highlighted with triangles. Next a line is drawn from the selected bubble to the desired new position. The pressing of the *mark* button causes the bubble to be moved.

The selection process only allows bubbles or wires that can be split to be selected.

The move function has a Boolean option called "move-sb". When set a binary search is done over the move vector for a successful move. If a success is found, the function terminates as normal or errors are given for a single grid step move.

5.3.7 Absorb

The absorb will remove the selected bubble and merge the only two connecting wires as described in Section 4.2.2. Only bubble that are absorbable are selectable.

5.3.8 Expand

The expand function will expand the cell side where the *do it* point intersects the side to be expanded and the amount of expansion is the distance from the *mark* point to the *do it* point. The expand function is described in Section 4.3.1. The screen is then refreshed and the selected primitive is unselected.

5.3.9 Constrict

The constrict function will constrict the cell side where the *do it* point intersects the side to be constricted. The amount of constriction is the distance from the *mark* point to the *do it* point. The constrict function is described in Section 4.3.2. The screen is then refreshed and the selected primitive is unselected.

5.3.10 Probe

The probe function display various attributes about the selected primitive when the *do it* button is pressed.

5.3.11 Set Portname

The set portname function allows altering the portname by pressing the *do it* button with the cursor on top of the port. A response is then requested for a new name for the port. The port names are used to interface the outside world.

5.4 Extensibility

One measure of the quality of an editor is extensibility[Stallman 80]. An example of an accepted text editor with extensive user modification capabilities is EMACS[Stallman 80]. The curvilinear cell editor we have just presented is extendable by using the uniform transaction process and by creating new functions which take advantage of the extremely powerful bubble move algorithm.

We extend the editor by creating a super move function which would move all obstacles in the path of a move of a bubble. This function is easily created by using a LIFO order queue or stack $S\{\}$. We initialize the stack $S\{\}$ by placing the desired move bubble into the stack. We now call the move algorithm with the top bubble in the stack. If the move fails we push on the stack all the interfering bubbles order by distance from the bubble just attempted to be moved. If the bubble move is accepted we take the bubble off the top of the stack. This process is repeated until the stack is empty. This is an example of extending the editor using the move algorithm to create a super move function. The editor with the graphic package is modular and very easily to modify.

One of the problems with existing physical editors is the lack of the ability to open a hole or region to add addition logic. With the supper move we could create a function that moves a block or region out of the desired area for editing. Another more powerful method would be to create a cost function where any primitive in a region would be extremely expensive. The region would be the location for the desired new logic. The compactor is then used to open the region up.

An example of using the uniform transaction process is with a function to cut out a rectangle and paste the result at a different location. To cut a rectangle out, the wires that cross the rectangle are split into two wires with an intervening bubble. The primitives on the interior of the rectangle are added to the delete queue along with the wires to the edge. The bubbles along the edge are added to the delete queue and copied and added to the add queue. The delete queue is then copied in a temporary queue $T\{\}$ for saving for the past. The uniform transaction process is then called for the cut. The primitives in the queue $T\{\}$ are then translated to the past position and the uniform transaction process is called for the past operation. This is an example of using the uniform transaction process to extend the editor.

5.5 Conclusion

The graphic editor just described has been used by the author to enter and modify some of the cells presented in this dissertation. The remaining cells we entered and modified by a mathematician, A.S. Frey at Caltech. The graphic editor has been demonstrated to several visitors at Caltech who after a few minutes of instructions were able to create several large cells. The graphic editor is easily used, easily taught, and extendable.

6

Simulated Annealing

There is a class of combinatorial problems that grow exponentially with the problem size that are called NP complete[Garey 79]. The graph based 2-D compaction problem has been shown to be in this class. Experience has shown that simulated annealing has been an effective method to find a near optimum solution for some problems of this class.

6.1 Optimization by Simulated Annealing

The simulated annealing optimization method is similar to heating and slowly cooling a metal. Extremely rapid cooling of a metal will not produce an ordered crystalline structure. For that reason, when we anneal a system it is very important that the system is cooled very slowly to produce a tightly packed structure. The same holds true in VLSI, that is if we want to find a global minimum we will cool the system very slowly and in doing so will allow the system to explore configurations that are away from a local minimum. This provides a form of backtracking and it involves accepting configurations that are less desirable than the present configuration. The equivalent of extremely rapid cooling in VLSI is the iterative improvement optimization method where only moves that improve the configuration are accepted. Iterative improvement methods will lead to a local minimum. It is only when the problem has no local minima that the two methods will give the same result. This is far from being the case in the problem under our consideration, i.e. the 2–D compaction problem.

The simulated annealing method, developed by Kirkpatrick[Kirkpatrick 83], is an adaptation of the Metropolis algorithm[Metropolis 53] used extensively in the simulation of physical systems in statistical mechanics for describing average properties of a physical system as a function of temperature. The algorithm was shown to produce configurations with the Boltzmann's probability distribution $e^{-E/kT}$ where E is the system energy and k is the Boltzmann's constant. The Metropolis algorithm uses N rigid-spheres in a two-dimensional space. The simulation game is played as follows. An initial configuration is established by placing the spheres in any configuration. A sphere is picked at random and moved to a new position by a small random displacement and the ΔE is calculated. If the $\Delta E \leq 0$ the move is accepted. If $\Delta E > 0$ then the moved is accepted with probability $e^{-\Delta E/kT}$. This is accomplished by comparing a random number $r_n[0,1]$ with $e^{-\Delta E/kT}$. If $r_n \leq e^{-\Delta E/kT}$ then the move is accepted. Otherwise the sphere is return to its previous position. This operation is then repeated. When the system reaches thermal equilibrium, the distribution of the population of the different energy states follows the Boltzmann's distribution. The Metropolis algorithm does not state how long it takes to reach thermal equilibrium.

The translation of the Metropolis algorithm to a general optimization function H that is to be minimized is straight forward. We will use X_i to represent the i^{th} system configuration. The cost of a configuration X_i is $H(X_i)$. The control parameter T_i in the units of function H will play the role of the thermal energy kT.

The simulation at a specific temperature is as follows.

1. Pick a new state X_j at random close to the current state X_i. Let $\Delta H = H(X_j) - H(X_i)$.
2. If $\Delta H \leq 0$ then $X_i \leftarrow X_j$.
3. If $\Delta H > 0$ then if $R_n \leq e^{-\Delta H/T_i}$ then $X_i \leftarrow X_j$. R_n is a uniformly distributed pseudo random number between $[0,1]$.
4. Repeat steps 1–3 until a stopping condition that estimates thermal equilibrium is satisfied.

The simulated annealing method operates as follows. First the system is randomized by simulating at high temperature until an equilibrium condition is approximated. The high temperature is large enough so that any further increase would not affect the average energy very much. Next, the system is slowly cooled by stages of decreasing temperature, T_i; at each stage a simulation is performed until the equilibrium condition is met. The sequence of temperatures and the stopping condition is called an annealing schedule. The annealing schedule is critical in the optimization process and is problem specific. Cooling too fast, skipping a critical temperature range, not waiting long enough at each temperature stage, and not heating high enough can cause lockup on a local minimum. On the other side, cooling very slowly, heating too high, and waiting too long at a temperature can require excessive computation time.

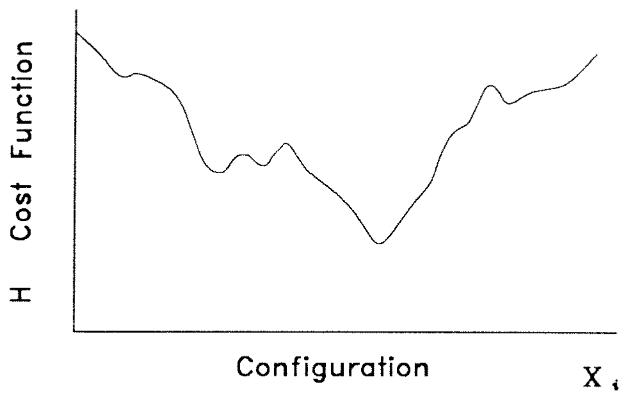


Figure 6.1 Example Cost Function

The simulated annealing optimization method performs well on a cost function with many local minima as shown in Figure 6.1 where no other heuristics are available. The reason is that the method allows jumping out of a local minima. Other algorithms are better suited for convex functions.

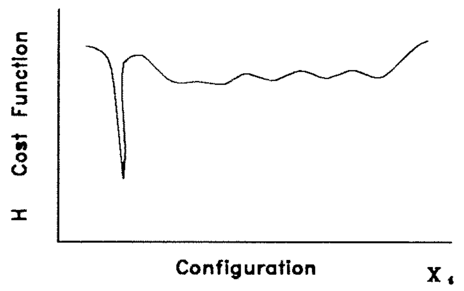


Figure 6.2 Poor Hypothetical Cost Function

There are cases when the likelihood of achieving a global minimum is not too high. A specific realization of this case is shown in Figure 6.2. What is important to notice is that the critical parameter that governs the approach to the minimum is given by the width of the region monotonically approaching the minimum. Extremely narrow minimums are very easy to miss during a finite search. The advantage of simulated annealing is that the whole configuration space does not need to be explored. Fortunately the 2-D compaction problem is not of this type.

Kirkpatrick points out that there are four ingredients for an implementation of the simulated annealing method which are: A concise description of the system, a method to make random perturbations to the configurations, an objective function, and an annealing schedule.

A requirement for an effective implementation of the simulated annealing method is that the perturbation from state X_i to state X_j should be computationally inexpensive. The change to the system should be very small and the computational cost of the change should be proportional to the amount of change. The reason for this is that the perturbation will be done over and over again. The calculation for the ΔH should be computed only using on the changed data and not by computing all of $H(X_j) - H(X_i)$.

It is important that all the allowable states be accessible from any arbitrary initial configuration in some sequence of moves. Also, the probability of choosing a candidate move from state i to state j should be equal to the probability of choosing a candidate move from state j to state i. If these two conditions are met, the Metropolis algorithm is guaranteed to the converge in equilibrium to the Boltzmann's distribution.

The framework described in Chapter 3 maps very closely to the simulated annealing model. The bubbles are similar to the molecules used by Metropolis. This is no accident. The framework was designed for the simulated annealing model. The requirement that the perturbations be small, computationally inexpensive and the cost proportional to the change is adhered to by the bubble move algorithm as described in Chapter 4. The calculation of ΔH is a simple matter of computing the cost of the add queue minus the cost of the delete queue. Changes to the cell are easily accomplished by the nature of the add and delete queues.

6.2 Background

The traveling salesman is one of the first problems to which Kirkpatrick[Kirkpatrick 83] applied the simulated annealing method. This problem is know to be NP complete[Garey 79]. The problem is to connect a set of cities by the shortest path where each city is visited once. The largest exact solution has been found for 318 cities. Kirkpatrick had good solutions for cities up to 6000. Kirkpatrick points out that most cities are clustered with sparse regions in between. He showed how the method first optimized the interconnection of the clusters followed by the optimization of the clusters as the system was cooled. This is an important property of the simulated annealing method. First the global constraints are optimized followed by the local constraints. The annealing schedule was determined empirically.

Other applications of simulated annealing include: global wiring, graph partitioning, number partitioning, graph coloring, PLA folding, etc. Many of these applications were discussed at the Workshop on Statistical Physics in Engineering and Biology held at IBM Yorktown in April 1984[Kirkpatrick 84b, Johnson 84, Mosteller 84, Suaya 84]. Johnson, in particular, compared the best know algorithm for each problem to the method of simulated annealing. The initial temperature for the problem was picked empirically while the cooling schedule was exponential. Best results using simulated annealing were found using the Boltzmann distribution. Johnson concluded that simulated annealing fared well in graph partitioning and possibly graph coloring, but not it did not fair well against heuristics on number partition and traveling salesman problems. Simulated annealing is computer time intensive[Johnson 84, Kirkpatrick 84a].

Simulated annealing was used for global wiring by Vecchi[Vecchi 83]. Experimentation was used to compare simulated annealing to heuristic methods for the global wiring problem. They have shown that the simulated annealing method is an effective algorithm for global wiring.

Simulated annealing was applied to a graph embedding problem[Steele 85] that was used to assign processes to processors on a Cosmic Cube[Seitz 85a]. Besides using the Boltzmann distribution, results were presented with a Fermi Dirac type distribution given by $P(\Delta E) = \frac{1}{1+e^{\Delta E/T}}$, this distribution discriminates against both positive and negative ΔE. The convergence to equilibrium is claimed[Steele 85] to be faster than that obtained with the Boltzmann's distribution.

6.3 Objectives

The objectives for the compaction process will need to be incorporated into the cost function. Tradeoffs between goals should be easily controlled by adjusting external parameters in the cost function. A user of the system should be able to easily provide guiding information on the priorities of the objective function. For example he may wish to have short wires at the expense of a larger cell area.

The first objective is to minimize the cell area. This is the normal goal in most compactors. The priority of the cell area minimization with respect to other objectives should be easily controlled.

In VLSI circuits the wiring plays a key role in the speed or performances of the circuit. We would like to be able to minimize the wire length of the cell being compacted. Various wire types in the design transmit signals at different speeds due to the capacitance of the wire. We would like to be able to control the priorities of wire minimization based on wire type. For example, the polysilicon wires are slower then the metal wires in nMos. We would like the polysilicon wires to be as short as possible at the expense of longer metal wires.

In some circuits to be compacted there are critical paths that effect the performance of the circuit. Examples of these circuits are clock lines and timing paths. We would like to be able to minimize the length of this type of path at the expense of other objectives such as overall wire length and cell area.

A unique requirement of the cost function is to preserve the integrity of the models. Our framework allows circuit devices to distort as described in Chapter 3. Model instances are allowed to bend and change shape during the compaction process. After compaction, we would like our device to be in a valid configuration. The objective function is used to control the model integrity.

Generally when cells are compacted the desired final shape and connector position is directed from a composition as described in Chapter 8. For example if we have two cells that are to be composed horizontally we would like the ports to be at similar positions and the height of the cells to be the same. Our objective function will be able to direct the final shape of the cell and the final port position.

6.4 How We Compact by Annealing

The objective function H which will be described in the next section is a function of the wire lengths, the model wire lengths and the bubble positions within the cell. The wire length part of the cost function satisfies the objective off minimizing the total wire length. The model cost satisfies the objective of preserving the model integrity. The bubble position cost is used for the minimum area and cell shape control. A conceptual way of viewing the bubble position cost is by considering the bubbles as spheres placed in a bowl as shown in Figure 6.3 for the horizontal axis. The positional cost of the bubble is more expensive near the perimeter of the cell and less expensive near the center. This creates a central potential pulling the bubble to the middle of the membrane. The desired final region for the cell is defined by the membrane position as shown in Figure 6.3. The bubble cost at the transition of the membrane will be discussed in the next section. The wall as shown in Figure 6.3 is used to limit the excursion of the bubbles by not allowing the bubbles to penetrate the wall.

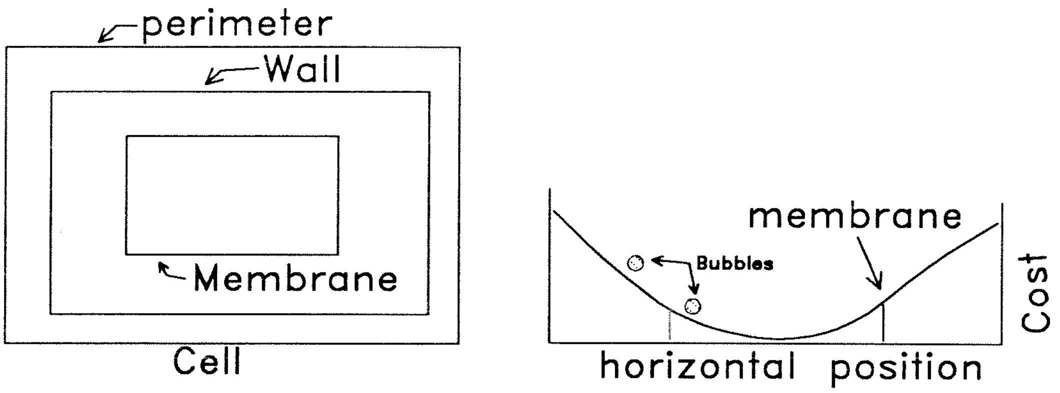


Figure 6.3 Cell Region Costs

The compaction process is divided into three annealing stages. The function of the first stage is to randomize the cell into an initial configuration and to provide a coarse annealing. This is achieved with a large step size (we use 8 grid units) and an exponential cooling schedule: $T_{n+1} = f * T_n$ with $f = .8$. The cell is first expanded if necessary as described in Chapter 4 to a size where the bubbles can freely move about. The wall is set to the perimeter of the cell. The initial temperature, T_i, is a value where no change in the "dominant" parameters would occur at higher temperatures. The dominant parameter is one that would cause the probability for a move with positive delta energy to be approximately 50%. The actual value for T_i will be defined from analysis of the cost function parameters and experimentation. This will be described in Chapter 7.

The central potential is used to guide the bubbles into the desired shape. This central potential is controlled by the placement of the membrane. The membrane is generally placed at the ideal area(see Section 3.9.2). During the first and second stage the central potential is present.

The annealing at a specific temperature T with step size s is accomplished in passes, where each pass tries to move each bubble once. At the completion of each pass, a check is made to see if the stopping condition has been met that would terminate the simulation at the temperature. A pass tries N moves where N is the number of bubbles in the candidate set, $C\{\}$, of bubbles. The candidate set is generally all the bubbles in the cell. Exceptions could be ports that are frozen at a specific position for interface reasons. The direction d of a move is chosen in 45°

increments, where a move length is approximately s grid units with the move end position constrained to be on a grid position.

> The follow steps are repeated N times where $N = |C\{\}|$ for each pass.
> 1. Pick a bubble b from $C\{\}$ at random.
> 2. Pick a move direction d at random. Call bubble move with bubble b and a move vector in direction d with step size s. If there was not a success, repeat step 2 up to 4 times. After 4 tries without a success, return to step 1.
> 3. Calculate ΔH based on the add and delete queues.
> 4. If $\Delta H \leq 0$ then update the cell with the add and delete queue.
> 5. If $\Delta H > 0$ then choose a pseudo random number R_n between $[0, 1]$ and if $R_n \leq e^{-\Delta H/T}$ then update the cell with the add and delete queue.

The purpose of the second stage is to fine tune the compaction. This is accomplished by decreasing the step size to a single grid unit and continuing annealing with a temperature T_i equal to the lowest temperature of the first stages divided by the ratio of step sizes. What this accomplishes is, in first approximation, to match smoothly $\Delta c/T$ of the previous stages to the start of this stage. In addition the constant f was chosen to be .85. The membrane position and central potential function used for the second stage is the same as for the first.

The final annealing stage is to remove the distortions caused by the central potential. The wall is set to the minimum bounding region after stage 2 and the central potential is removed. The initial temperature for stage 3 is based on the wire cost. An annealing is then performed to produce the final compacted cell.

6.5 Choice of the Cost Function

The objective function H is a sum of the costs for the wire lengths, the model wire lengths, the bubble position and the user port control. The objective to minimize the cell area is encapsulated in the membrane cost. The goal to minimize the total wire length is the wire cost. The goal to preserve the soundness of the circuits is the model wire cost. The port cost allows the final position of the ports to be controlled. The cost function is

$$H = \sum_{n=1}^{N_w} C_w(W_n) + \sum_{j=1}^{N_{mw}} C_i(W_j) + \sum_{k=1}^{N_b} C_{mb}(B_k) + \sum_{l=1}^{N_a} C_{ma}(A_l) + \sum_{m=1}^{N_p} C_p(P_m). \quad (6.1)$$

The function C_w is the cost for a wire where the index n ranges over the normal wires. The function C_i is the cost for a model wire where the index j ranges over the model instance wires. The function C_{mb} is the membrane cost for a bubble while C_{ma} is the membrane cost for an around. The index k ranges over the bubbles while the index l ranges over the arounds. The function C_p is the port cost. The index m ranges over the ports.

The rule for a bubble or wire as described in Section 3.3.2 has a value called *COST*, C_l that is the cost for a single grid unit step. All lengths will be measured in the grid unit, D_g. This cost is used by the wires and models. This is the user control that allows priorities between the different types of objects defined by the rules.

6.5.1 Wire Costs

The wire cost is a linear function so that there is no difference in cost caused by intermediate bubbles or structures on the path of a wire. The wire cost is

$$C_w(W_i) = \frac{\text{length}(W_i)}{D_g} * \text{cost}(W_i) \tag{6.2}$$

where the cost(W_i) is the constant, C_l, from the rule for wire W_i.

Experiments concerning the choice of the values for the rule cost are described in Chapter 7. For now let us consider that we would like metal wires to be less costly that polysilicon wires. Suppose we assign a value of 1 to the metal wire cost then the polysilicon wire cost would need a value greater than 1 for the polysilicon wire to have priority over the metal wire. If we assign a very large value to the polysilicon wire cost say 1000 times the metal wire cost, the metal wires would have no influence on the length of the polysilicon wires. Influence is the ability of one wire to give bias to another wire that is felt before the stopping condition is reached. One way to look at it is that the polysilicon wire would lock up long before the metal wire becomes dominant. The effects of wire cost ratios will be shown with experiments and analysis in Chapter 7.

6.5.2 Model Wire Costs

The purpose of the model cost is to insure with a reasonable probability that the integrity of the model is preserved. The cost for the model wires needs to be greater than the attached wires so that the model will pull together in the end. We would like to limit the excursion of model bubbles from the model center so that the model does not pull apart so far that it can not come back together before other wire costs start to dominant. This implies a very large wire cost on the model wires. However if we use a very large value for the model wires, any attached wire to the model will have little influence on the position of the model. To accommodate these conflicting goals, we will use a combination of a quadratic and linear term for model wire cost. When the model wire is long the quadratic term will dominant. However when the model wire is short the linear term will dominant. This means that when the model wire is long or stretched out the delta cost will be very expensive. When the model wire is short the model wire delta cost will be less expensive. This satisfies our goal for model cost.

When the model wire is in the shortest length position, the cost for a single step move from the position is equal to the cost defined in the rule for the wire. Additional steps increase the cost based on a growth factor, G_m. The model wire cost is

$$C_i(W_j) = \frac{G_m}{2} * \left(\frac{\text{length}(W_j)}{D_g}\right)^2 + k * \left(\frac{\text{length}(W_j)}{D_g}\right) \tag{6.3}$$

where G_m is the *model growth rate* of the specific model. The growth rate is a global parameter which is the same for all models.

The constant k controls the initial value for the cost where

$$k = \text{cost}(W_j) - \left(G_m * \frac{L_o}{D_g} + \frac{G_m}{2}\right). \tag{6.4}$$

The length L_o is the initial length of the wire in the model. This is the smallest length for the wire that is possible due to the specific model rules. The cost is defined in the rule for the wire.

The choice for the model wire cost will depend on the expected attached wires to the model. The cost should be a little greater than the cost of the attached wires.

6.5.3 Membrane Cost (Central Potential)

The central potential cost provides the incentive for the bubbles to move into the desired region called the membrane as shown in Figure 6.4. A bubble or an around incurs a cost when its edge is outside the fence. The fence is centered inside of the membrane. The delta cost increases as the distance from the fence increases. The membrane position is the desired shape of the cell. We would like the dominant cost for bubbles outside the membrane to be the central potential cost while bubbles inside of the membrane will have the attached wire cost as dominant. If this were not true we would have wires distorted by the influence of the central potential for bubbles inside of the membrane. There is a separate central potential cost for the horizontal and vertical axes. This allows for different perimeter shapes.

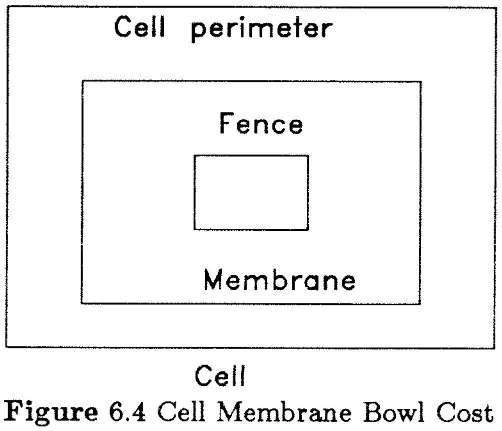


Figure 6.4 Cell Membrane Bowl Cost

We will consider the membrane as a bowl as shown in Figure 6.5 for the horizontal axis. We would like the influence of the membrane to be less than the influence of the wires while primitives are inside of the membrane. When a primitive is outside of the membrane we would like the membrane cost to be predominant. There is a region of the cell where there is not a membrane cost which is inside of the fence.

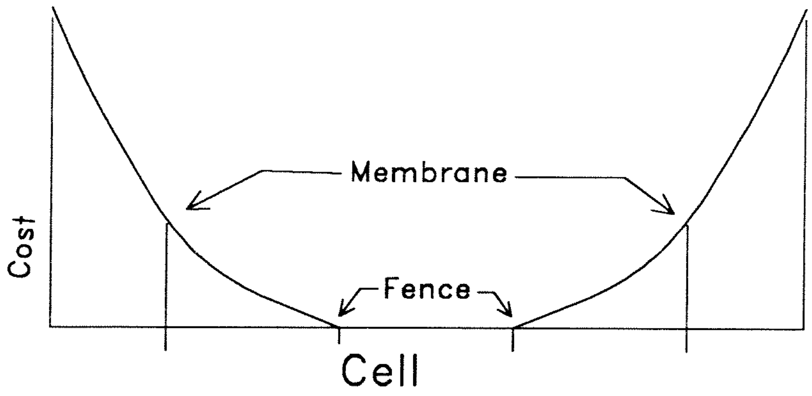


Figure 6.5 Cell Membrane for Horizontal Axis

The central potential cost for a bubble is the sum of the effects of the membrane on each side the fence. Of course two of these will usually be zero. The central potential cost is thus

$$\begin{aligned} C_{mb}(B) = & M(\mathrm{Dst}_l(B), d_h) + M(\mathrm{Dst}_r(B), d_h) \\ & + M(\mathrm{Dst}_t(B), d_v) + M(\mathrm{Dst}_b(B), d_v). \end{aligned} \tag{6.5}$$

The function M is the cost for an object in relation to a side where Dst_l is the directed distance from the outer most edge of the object to the left side of the membrane. The reason we use the outer most edge of a primitive is that bubbles can have different radii where we would like the influence of the membrane to push the object inside of the membrane. The other three Dst functions are distances for the right, top, and bottom edges respectively. The distance d_h is the signed horizontal distance from the membrane to the fence edge while d_v is the signed vertical distance from the membrane edge to the fence.

The proper definition for the function M will be left to the experiments in Chapter 7. For now we will use

$$M(d, d_s) = \begin{cases} 0 & \text{if } d \le -d_s \\ \left(\frac{C_i}{(2*d_s+1)}\right) * (d_s + d)^2 & \text{if } -d_s < d \le 0 \\ \left(\frac{C_g}{2}\right) * d^2 + \left(C_i - \left(\frac{C_g}{2}\right)\right) * d & \text{if } 0 < d. \end{cases} \tag{6.6}$$

The constant C_i is the cost of a one step move outside of the membrane. The value should be a little larger than the maximum port wire cost. This is to insure that a connected bubble will not be pulled away from the membrane by a port wire. Inside the membrane the cost reduces as the object gets closer to the fence. Outside of the membrane the cost increases with a growth factor C_g.

A central potential cost for an around A on one side of the membrane will only occur when a perpendicular vector pointing outward from that membrane side intersects the around and the around is not a member of a wire going to a port on the cell side of the membrane. The function $\mathrm{TST}(A, s)$ returns a 1 if the test is true; otherwise a 0 for this condition.

The membrane cost for an around is the sum of the effects of the membrane side cost where TST is 1 for the around which, i.e.

$$\begin{aligned} C_{ma}(A) =& \mathrm{TST}(A,l) * M(\mathrm{Dst}_l(A), d_h) \\ &+ \mathrm{TST}(A,r) * M(\mathrm{Dst}_r(A), d_h) \\ &+ \mathrm{TST}(A,t) * M(\mathrm{Dst}_t(A), d_v) \\ &+ \mathrm{TST}(A,b) * M(\mathrm{Dst}_b(A), d_v). \end{aligned} \tag{6.7}$$

The cost term, C_i in M may be different than that for the bubbles.

6.5.4 Port Costs

The port cost is the sum of the membrane influence and the user directed costs. The membrane cost is used to pull the ports perpendicular to the desired membrane region. The user port cost is used for composition considerations. The port cost is

$$C_p(P) = U_p(P) + \begin{cases} M(P, \mathrm{Dst}_b, d_v) + M(P, \mathrm{Dst}_t, D_v) & \text{if P Left or Right} \\ M(P, \mathrm{Dst}_l, d_h) + M(P, \mathrm{Dst}_r, D_h) & \text{if P Top or Bottom.} \end{cases} \tag{6.8}$$

The C_i for the ports will be based on the port wire and will be discussed in Chapter 7. The function U_p is the for user control over the ports. This allows adding costs between ports and positions.

7 Experimental Results and Analysis

This chapter describes experiments and their results obtained by applying the compaction method previously described. Experiments show the effect of choosing different cost function parameters, including wire costs and membrane costs. Different annealing schedules are tested. The complexity of the compaction method is estimated using a controlled set of experiments with a range of cell sizes. To estimate the ability of the algorithm to compact, several cells are compacted and compared to hand designed versions. To show that the curvilinear framework can build working chips, several cells were fabricated and tested.

7.1 Choosing Cost Function Parameters

Experiments have been conducted to determine how the various cost parameters interact. There are two types of wire cost functions - a quadratic for model wires and a linear function for normal wires. The coefficients for these functions are specified by the designer in the rule associated with each wire. There also is a central potential cost function which pulls the various primitives toward the center of the cell. We explore how the choice of the parameters for the central potential and the parameters for the wire functions interact. The coming sections presents these experiments.

7.1.1 Wire Cost Experiments and Analysis

Wires form the fundamental structural element between bubbles. To hold the collection of bubbles together the wires exert a contraction force on their respective end bubbles. This contraction force comes from the wire cost function which specifies a pull between connected bubbles. This pull influences the final position of the connected bubbles in a compacted cell.

Within our framework there are wires connecting model instances and wires within a model instance. The wires connecting instances exert a contraction pull on these instances while wires within model instances preserve the integrity of the models. Within an instance we would like to limit the expansion of the model instance and insure there is sufficient force within the instance that a proper shape is present after compaction. On the other hand wires connecting model instances need less cost and should not limit the movement of the model instance. This implies two general classes of cost parameters where the first is for the normal wires and the second is for the model instance internal wires.

Within the model we will limit the excursion of the member bubbles by using a quadratic function. By controlling the parameters we can limit the excursion of the member bubbles.

A linear cost function for the normal wires was chosen so that there would be no bias on intermediate bubbles along the path of a wire and that two wires of the same length, one with an intermediate bubble and one without, would have the same cost. If this was not true, bubbles or bubble structures along the paths of wires would be biased in some direction and would not tend toward their natural position.

In the next section we will describe how to choose the parameters for the normal wires followed by a section for choosing the internal model wire parameters.

7.1.1.1 Linear Cost Experiments and Analysis

What is important about the linear cost parameters is the ratio of the cost of two wires connected to the same bubble. If we want one wire to be shorter than the other then we would make the cost of the first wire less then the cost of the second. The simulated annealing compaction would make the first wire shorter at the cost of the second where the minimum of the two would be found. This is only guaranteed to be true if we simulate until the equilibrium condition is met at each stage of cooling. However in practice we never reach this condition. We try to measure this condition with a stopping criteria however this is not very accurate. Also we would like to minimize cpu time. Therefore the effect of the cost parameters should be as quick as possible.

What we would like to know is how to set the linear cost parameters. Lets consider that we have only two wire types and we would like the first to be shorter than the second.

Consider Wire 1 with cost per unit length of c_1 connected to a bubble b_1 and Wire 2 also connected to bubble b_1 with cost per unit length of c_2 as shown in Figure 7.1. Suppose we would like Wire 2 to be short even at the expense of a longer Wire 1. Therefore c_1 should be less then c_2. But by how much? If we make $c_1 << c_2$ then Wire 2 would dominate in determining the position of b_1 and Wire 1 would have very little effect. If we make c_1 very close to c_2 then there may not be enough bias for Wire 2 to be shorter than Wire 1. To answer these question we will use two experiments.

We ask the question, if two wire costs differ by a given ratio then how much influence will the cheaper wire cost have on the second wires structure? We will use two bubbles, b_1 and b_2, connected by a wire with a cost c_2 and tethered by a wire to a port with cost c_1 as shown in Figure 7.1 as Case 1. We will freeze the port at the center position along the side as shown. Then for a range of ratios between c_1 and c_2, we anneal this configuration(using a fixed annealing schedule described below) and measure the distance of bubble b_1 to the port. This experiment will show the influence of the wire with cost c_1 on the two bubbles structure. If c_1 is very small as compared to c_2 then the structure would be positioned at a random place within the cell after annealing. Of course the two bubbles, b_1 and b_2, will be separated by the minimum distance because they are pulled together by Wire 2. As the system cools the probability of accepting a move by b_2 away from b_1 decreases. Therefore b_2 would lock to b_1 at the end of annealing. For $c_1 << c_2$, during the cooling the probability of b_1 taking a move away from b_2 and toward the port would decrease at about the same rate as b_2 moving away from b_1. Therefore the bubbles would be positioned at a random position in the box but next to each other at the end of annealing. However if cost c_1 is close to c_2 the cost of b_1 taking a move toward the port is small. Thus the probability of moving there is much higher than the probability of b_2 moving away from b_1. Therefore the structure would tend quickly toward the port. Thus for a fixed annealing schedule, the distance from b_1 to the port reflects the influence from the ratio of c_1/c_2.

In experiment using Case 1 of Figure 7.1, the cost c_2 will be fixed at 10 and we will vary cost c_1 from 10 to 1 in increments of 1. The measure we will use is the distance of bubble b_1 to the port. We will assume $c_1 \leq c_2$. When c_1 is very close or equal to c_2 the bubble b_1 will be pulled into the port. The cost c_1 will have a high influence on c_2. When cost $c_1 << c_2$ then there will be little influence and bubble b_1 will assume a random position in the cell. This case will test for how much influence we need.

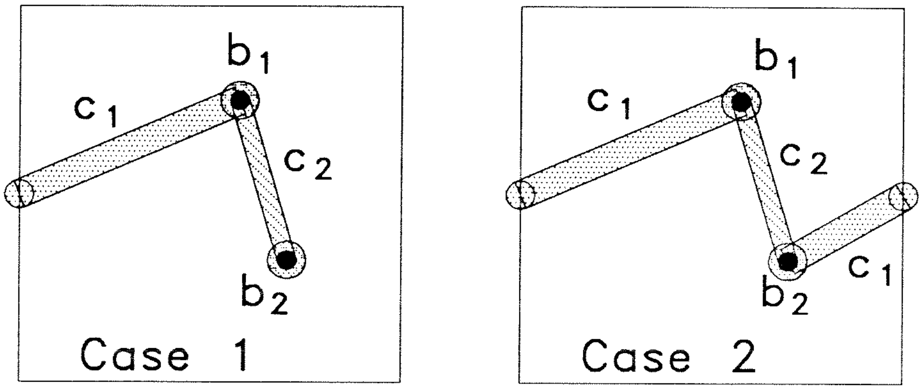


Figure 7.1 Wire Cost Test Set

To go along with the previous case, we need to know how close two wire costs can be and still have an effect from the difference. Case 2 will test for too much influence of c_1 on c_2. We will add a new wire from bubble b_2 to a port on the opposite side of the previous port which is also frozen. When cost $c_1 << c_2$ then the two bubbles will take a random position after the annealer because c_1 has no effect. When cost c_1 is very close to c_2, the two bubbles will take a random position along the line between the ports. When the cost relation is proper the two bubbles will be next to each other at a random position along the line between the ports.

If c_1 is too close to c_2 then the distance from bubble b_1 to b_2 will be large. If c_1 is too far from c_2 then the distance from b_1 to b_2 will be small but the distance of bubbles b_1 and b_2 from the line joining the two ports could be large. The measure we will use for Case 2 is d, the sum of the distance between bubble b_1 and bubble b_2, and the distances of bubble b_1 and bubble b_2 from the line joining the two ports.

To be able to compare various values we use a fixed cooling schedule with a fixed number of passes. A pass is a move of each bubble once. We will use an initial temperature where the probability is 0.5 for bubble b_2 taking a move away from b_1 in Case 1. The temperature is then decreased exponentially by .8. We will use a fixed number of passes per temperature. The move distance is one grid unit(0.25λ). Runs were made where 20, 40, 60, and 80 were the number of passes. The number of passes per temperature was determined experimentally by using the stopping criteria with the value of c_1 equal to 10. The reason we do not use a variable stopping condition is that we would not want the result to be biased by running more passes. What we are trying to see is how strongly the influence is felt in a fixed time.

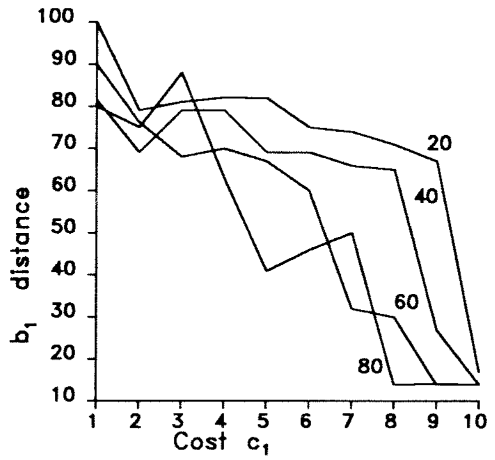


Figure 7.2 Wire Cost Influence

The results of experiments for Case 1 are shown in Figure 7.2. It can be seen that a cost greater than 8 for c_1 will result in the maximum influence for a pass count of 80 where the maximum influence is the smallest b_1 distance. The minimum distance for b_1 is 14. As the pass count decreases, of course, the influence decreases as can be seen from the plots using 60, 40 and 20 passes. A run using the stopping criterion based on reaching near equilibrium should have the effect of the pass count of about 80.

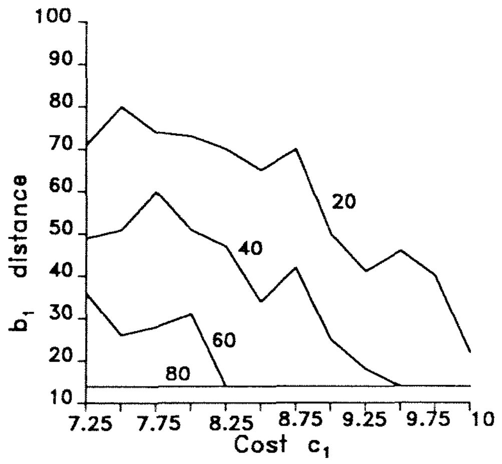


Figure 7.3 Wire Cost Influence Refined

A second independent set of runs was a refinement of the first for the values between 7.25 and 10 in increments of .25 as shown in Figure 7.3. Notice that, for the 60, run a value for c_1 of above 8.25 is necessary for maximum influence. We conclude that for a normal annealing a value greater than 8 will give sufficient influence.

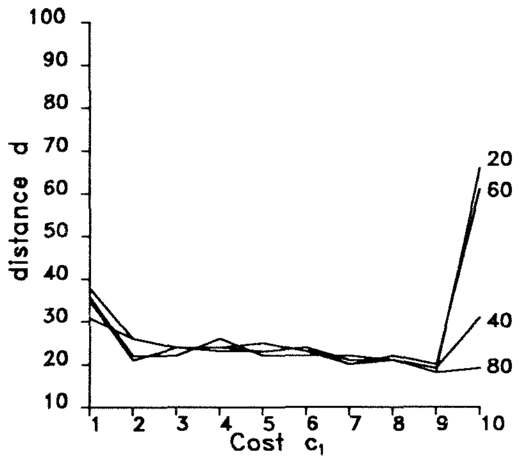


Figure 7.4 Wire Cost Influence Refined

The results of the experiments for the Case 2 are shown in Figure 7.4. As should be expected, for costs $c_1 = c_2$ gives a random distance for d. The maximum value for d is 131 when both bubbles are on the line joining the ports. For the first value of 9 all runs give about the same minimum value where the absolute minimum is where $d = 16$. Notice that for the range of values $2 \leq c_1 \leq 9$, the value d only increases very slowly if at all. The value d starts to increase after c_1 becomes less than 2. This is where cost c_1 has relatively little effect on the bubbles b_1 and b_2.

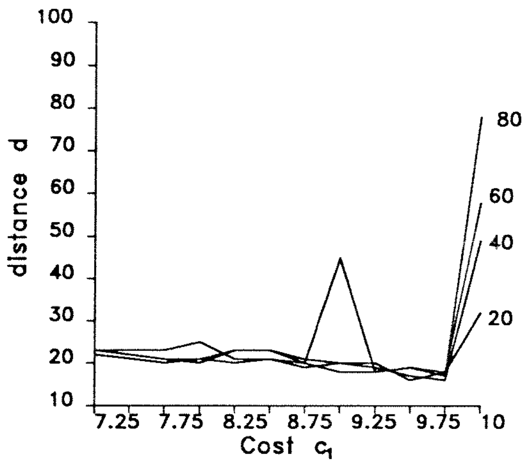


Figure 7.5 Wire Cost Influence Refined

A refinement experiment was done for the second case, where the refinement was for the range from 7.25 to 10 in increments of .25. What can be seen from the refinement is that all values are very good. There is one anomaly in the case for 20 passes which probably is due to the fact that this is a random process.

From Case 1 we can concluded that the ratio of wire cost should be greater than about .8 for sufficient influence, although this is not necessary for minimizing wire length as shown from Case 2. The minimum value for a wire cost ratio should be above .2 which is the lower limit where one would cost relatively mask out the other cost. This would effect the final cell wire length minimization.

The wire costs are flexible and can be tuned by the designer of the rules file for any need. However the minimum cost, c_{min}, should be greater than 20% of the maximum cost c_{max}. If a large influence is desired then the wire costs should only differ by .8. The wire costs we are using adhere to the minimum and are close to the sufficient influence ratio.

7.1.1.2 Model Cost Experiments

A wire in a model has a quadratic function of the form $(ax^2 + bx)$ for its costs. The coefficient a is a global parameter for all models and b is unique to the wire in the model. The coefficient b determines the cost for a single step move at the initial position. This parameter b includes a factor based on the cost for the wire in the technology file. The coefficient a determines how much the cost will grow from the initial position.

The model costs experiments are similar to the previous wire cost experiments. We use the same two cases but with the intermediate wire replaced with a transistor as shown in Figure 7.6. If it was not for the growth factor, we would know from the previous wire experiments how to set the ratio of costs for sufficient influence for a single step move. But the question is how do we set the growth factor. A value too high may not let the model move, or a growth factor too low may not force the model to return to a valid shape. Therefore the experiments are designed to test the effects of different growth factors.

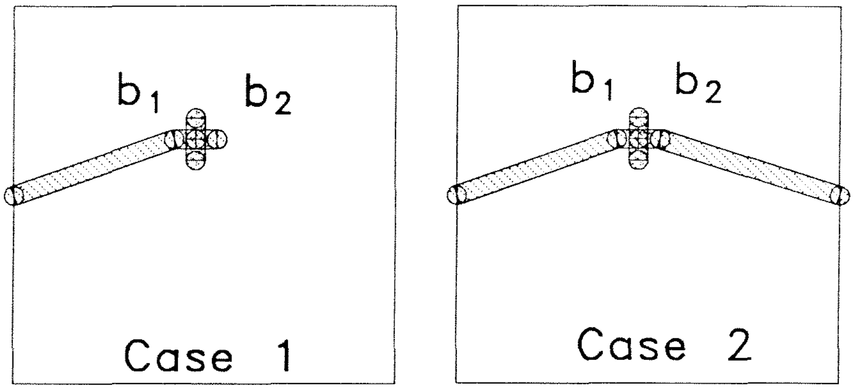


Figure 7.6 Wire Model Test Set

For Case 1 and Case 2 we set the attached wire cost at about 80% of the model wire cost at its initial position. For each case we vary the growth factor. For the model we need a larger number of passes to get an effect. The range for the number of passes needed to reach equilibrium was determined experimentally using are usual stopping criterion. The first experiment used a move size of one grid unit(0.25λ).

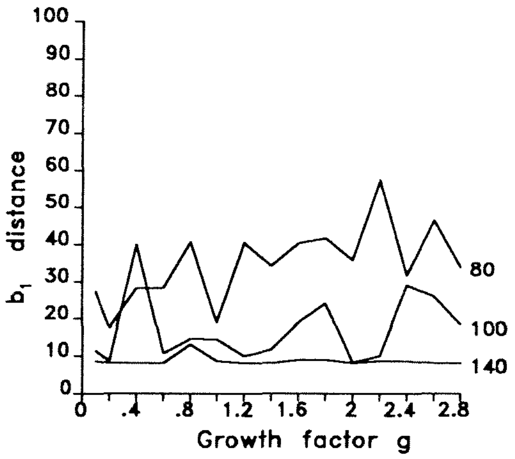


Figure 7.7 Model Growth Cost Influence Case 1

The results of the first case are shown in Figure 7.7. The growth factor is varied from 0.1 to 2.8. We are measuring the distance of bubble b_1 to the port. Observe that the growth factor does not influence the movement of the model. What is extremely important about this is that the model can have a very high growth factor for single step moves and still be pulled to the frozen port. This means that under most conditions, the growth factor can be set high enough so that the models will return to a valid shape. Having it too high does not hurt the compaction. This is a very unexpected result.

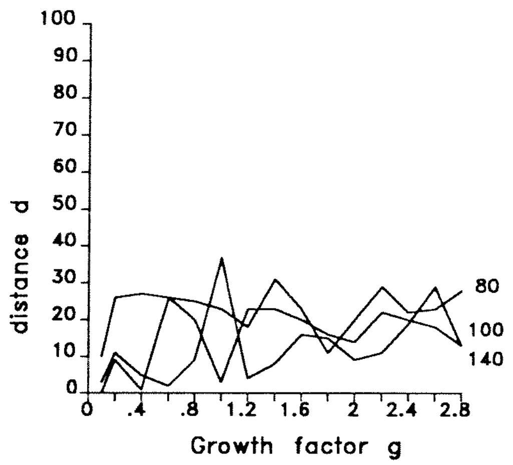


Figure 7.8 Model Growth Cost Influence Case 2

The second case measures the effect of the model being pulled by two wires. The results are shown in Figure 7.8. The distance d is the sum of the distances from the bubbles b_1 and b_2 to a line drawn between the two edge ports. For this case it again appears that the growth factor has little or no effect on the movement of the model.

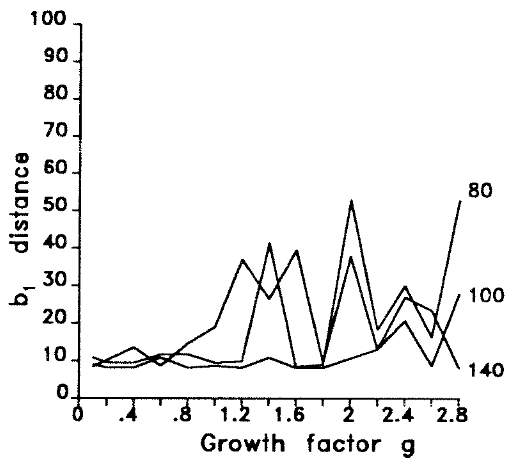


Figure 7.9 Model Growth Cost Influence Case 1, 4 move size

The first case was repeated using a move size of four steps. Notice that in the previous case the cost of minimum move always had the same ratio of cost to the attached wire cost. However in the case of a four step move the cost grows with the growth factor. The growth factor will now have an effect. Notice in Figure 7.9 that the distance b_1 to the port starts to vary randomly at about a growth factor of 1.0. It is just at this point where the ratio of the cost of the wire b_1 to the port to the wire from b_1 to the center bubble of the transistor falls below the .8. As the growth factor grows this ratio gets smaller. As seen in the linear experiments when this ratio gets below .8 the wire to the port starts to lose its influence. What we conclude from this is that the growth factor should be adjusted for the move size.

Two important results came out of these experiments. First, for single step moves the growth factor should be very large. Second, for large moves the growth factor should be adjusted for sufficient influence using the results of the linear experiments.

7.1.2 Central Potential Parameters

The central potential provides an incentive for the bubbles to move toward the center region of the cell. The central potential can be considered gravity in a bowl rising steeply outside the membrane and flattening out in the center of the cell. Bubbles are pulled toward the center of the cell. The force of the central potential on the bubbles increases as the distance from the center increases. A rectangular membrane is used to define the growth of the central potential. The membrane is intended to be placed at the estimated boundary of the compacted cell. The derivative of the central potential at the membrane is specified to determine the growth of the central potential inside the membrane. The growth of the central potential outside the membrane is specified independently. It is used to pull objects into the estimated compaction region.

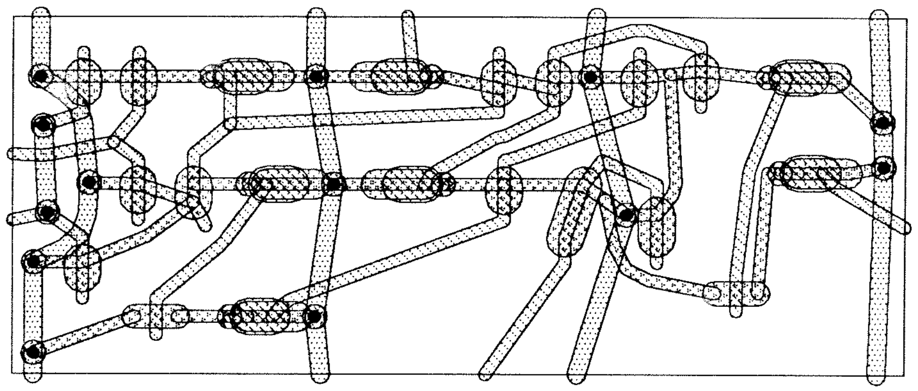


Figure 7.10 Input Cell to be Compacted

To gain an idea of the effects of the central potential, we will show three versions of a compacted cell, one without a central potential, one with an overzealous central potential, and one with a normal central potential. Consider the uncompacted cell shown in Figure 7.10. The cell, compacted without a central potential, is shown in Figure 7.11. It is interesting to observe that this cell is not very compact although the wires have been pulled to their shortest lengths.

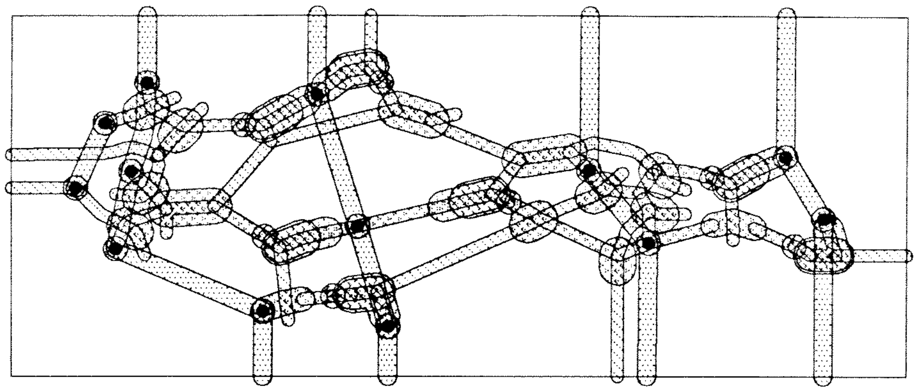


Figure 7.11 Compaction Without Central Potential

Now consider what happens when the membrane is in the wrong position or central potential cost is too high. An example can be seen in Figure 7.12. In this case, the transistors are deformed, and the wires are over stretched by the bubbles pulled to the center. An interesting point about the overzealous membrane is that it causes the running time to go up. This phenomenon is examined further in the next section.

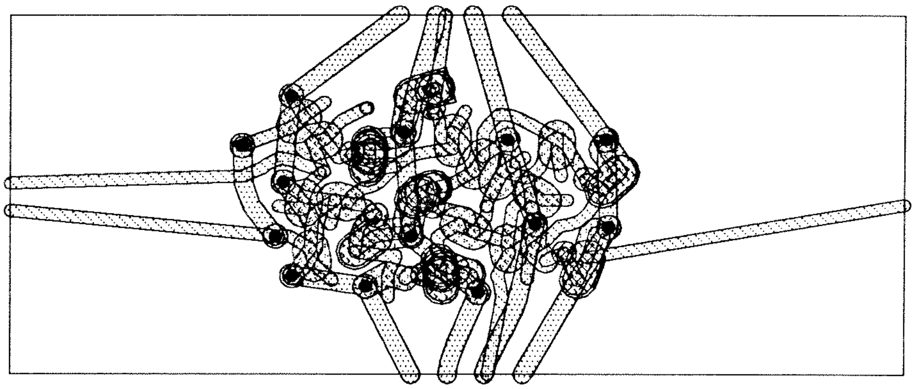


Figure 7.12 Compaction With Overzealous Central Potential

A compacted version with a normal central potential is shown in Figure 7.13. Notice that the transistors are well formed, and the wires are not overly stretched. A properly positioned membrane will pull the bubbles inside the membrane without undue distortion.

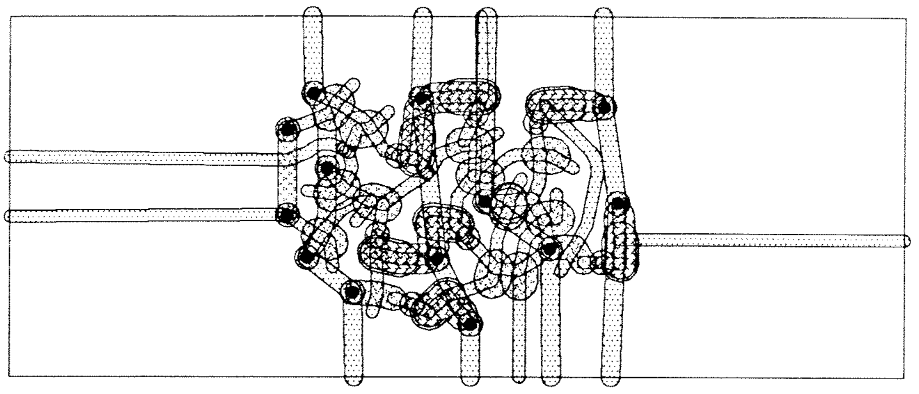


Figure 7.13 Normal Central Potential

In the next section we will describe how to position the membrane. The following section will describe the central potential function and how the parameters are chosen.

7.1.2.1 Choosing Membrane Rectangle

The membrane is chosen by the designer of the cell or by a composition system - see Chapter 8. The membrane perimeter is intended to enclose the region of the final compacted cell and thus describe the final desired shape.

The guide for the size of the membrane is the ideal area, A_{ib}, described in Section 3.9.2. In that section we saw that there is a minimum size for each side of the cell based on ports. For a rectangular cell this can be translated into a minimum horizontal length, H_m, and a minimum vertical length, V_m. The respective sides of the membrane should be greater than or equal to these values. The total area inside the membrane also should be greater than or equal to the ideal minimum area, A_{ib}. For our initial experiments we chose a rectangular membrane to enclose an area equal to the ideal minimum area, A_{ib}, always maintaining horizontal and vertical lengths greater than or equal to the values, H_m and V_m, respectively. A normal cell with the devices taking up most of the room, would compact to about a ratio of 1.5 for actual area divided by ideal area. However, for a cell that has a lot of wiring, the ratio of compacted area to ideal area can be greater than 2. An example of this type of cell is ***stuff*** as described in Section 7.4. In this case it was found that compaction would be improved by using a larger membrane. A better choice for the membrane perimeter was found to be about the square root of two larger.

There are two possible ways to adjust the membrane size. The first would be to augment the calculation of the ideal area with some consideration for wiring area. This could be accomplished by checking nets that go through the whole cell and intermediate long wires. The second is to dynamically adjust the membrane perimeter to be larger when during compaction the minimum bounding area of the cell is larger than the membrane rectangle. Neither of these modification have been implemented. Experiments presented in future section were conducted with the membrane at the ideal area.

However we wanted to know how critical this choice of membrane size would be on the compaction of cells. To evaluate this, we chose a medium-sized cell and compacted it with various membrane sizes. Four experiments were run; first, with the membrane area equal to the ideal area, second, with the membrane perimeter multiplied by two, third, with the membrane perimeter multiplied by four, and last, with the membrane perimeter multiplied by six. Table 7.1 shows results from these experiments.

Membrane scale	Cpu time	Compact size	Actual/Ideal
1	6:0:8	$5451\lambda^2$ $79\lambda \times 69\lambda$	1.76
2	4:49:17	$5440\lambda^2$ $80\lambda \times 68\lambda$	1.73
4	3:42:19	$5762\lambda^2$ $86\lambda \times 67\lambda$	1.86
6	4:40:58	$6390\lambda^2$ $86\lambda \times 67\lambda$	2.07

Table 7.1 Membrane Size Statistics

Observe that the cpu time for compaction decreases dramatically as the membrane position is moved out. Only when the membrane is moved so far out that its effect starts to become negligible does the run time finally increase. Also the compacted size improves when the membrane is moved to twice the ideal area position. These surprising results suggest the need for substantial further research.

The choice of the membrane size can affect the run time of the compaction. Using a membrane that is far too restrictive will cause the compaction run time to increase dramatically. The compaction run time will be minimal for a properly sized membrane. The following figures show the compacted cells for cases 1,2 and 4.

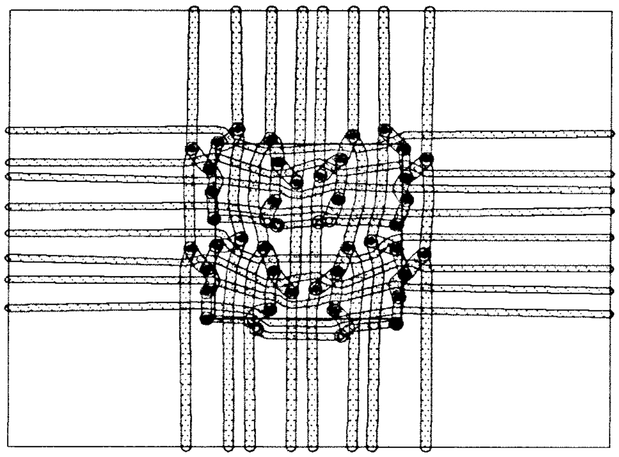


Figure 7.14 Membrane at Ideal Area

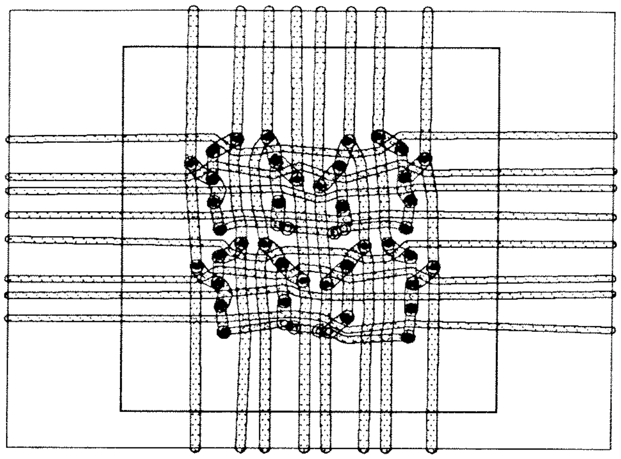


Figure 7.15 Membrane at 2 times Ideal Area

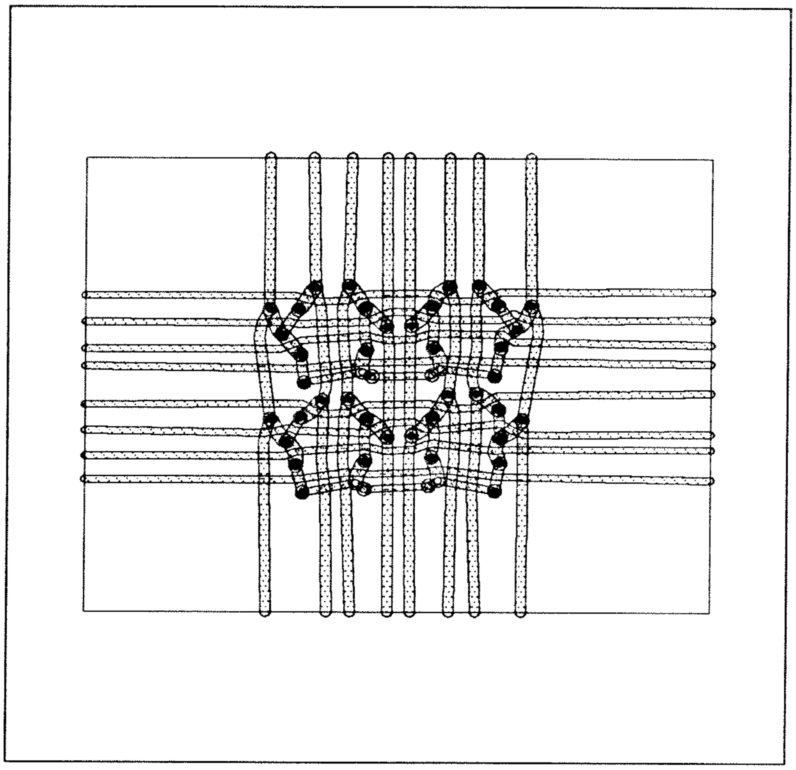


Figure 7.16 Membrane at 4 times Ideal Area

7.1.2.2 Choosing Central Potential Parameters

The purpose of the central potential function is to attract the bubbles into the membrane. We would like the pull on a bubble to diminish once it is inside of the membrane. In the center region of the membrane we would like no central pull at all. The reason for the decrease is that if the potential was large the wires connecting the bubbles would over stretch and distort. It is conjectured that the bubbles outside the membrane will be pulled inside the membrane faster by a stronger central potential. Early on in the experimentation of the membrane a linear pull was used. A large value was necessary to pull the bubbles to a compacted region. However the resulting structures in this region were highly distorted. Thus a long re-annealing time was necessary to remove the distortions. To correct this deficiency a quadratic central potential function was chosen to replace the linear function.

We also consider the possibility that it might be desirable to have a region in the middle of the cell inside of which there was no central potential. To provide for this, we define a square called the *fence*. We use a quadratic function for the central potential going out from the fence to the edge of the membrane as shown in Figure 7.17. This function has the property that the cost decreases as the bubble gets closer to the fence. The fence is the point where the central potential cost flattens out. For all the experiments conducted, we have placed the fence at the center of the membrane so that there is no region without any central potential. At any point on the membrane perimeter, the cost for a bubble to move one step perpendicularly outside the membrane is the same. Notice that the rate of decrease of the cost from the membrane to the fence may be different for each axis for a rectangular membrane.

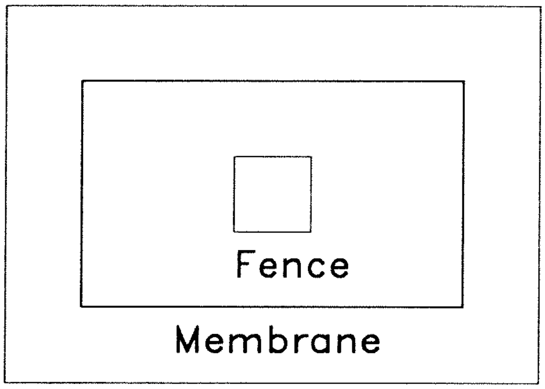


Figure 7.17 Membrane and Fence

There are two controls that govern the costs for the central potential. The first control is the growth factor C_g which controls how fast the cost rises per step outside of the membrane. Outside of the membrane we would like the cost to increase uniformly. This is to provide a uniform pull into the membrane. The proper value for this growth factor is still to be determined.

The second control is the cost,C_i, of a one grid unit move from the membrane edge going outside of the membrane. This cost is chosen to be a little greater than the port wire cost, so as to pull in a device such as a contact that is connected to an edge. For example, lets say we have a contact that connects to the top side with a wire and to the left side with a wire. If the value of C_i was less than the wire cost that the contact would be pulled to the edge.

This choice for C_i has been found experimentally to cause two problems. First, the central potential cost near the membrane perimeter over shadows the wire cost and cause distortions. Second, the over zealous central potential cost causes large running times. The appropriate solution to this problem has not been determined and left for future research. We would like, as we near the final stages of compaction, the effects of the central potential on interior bubbles to be reduced. We conjecture that, by using higher powers for the central potential function inside of the membrane, the goal might be satisfied.

A second conjectured solution is the following. The proper value for C_i should be equal to the minimum difference between the cost of any two wire types. Special provision can be made to pull bubbles that are only connected to ports into the compacted region.

7.1.3 Choice of Move Length

Before we discuss the move length we will review some properties of the bubble move algorithm as described in Chapter 4. The move algorithm has two parts. The first is clear path. The time for clear path increases with the number of primitives examined. For an increase in move length the number of primitives examined grows. The second part of the move algorithm, queue construction, is about the same complexity for large or small moves for the following reason. For each wire to be moved, the number of bubbles need to be examined depend of the segment length more than on the move length. This is so because the length of the segment is a lot longer than the amount being moved. Therefore a series of small moves is more expensive than a large move.

The move length is the distance in grid units that the bubble will try to be moved. When the density of the cell is sparse, a large move size is used since it is more cost effective. We have chosen the average bubble size as the large move size. As the cell is compacted the density goes up and a lot of potential moves are blocked. The ratio of unblocked move calls to total calls decreases rapidly as the density increases - see Section 7.3. At this time it is more cost effective to use a smaller move size which is a grid step(approximately 0.25λ).

We conjecture that a dynamic move size based on the distance outside the membrane would give an optimum move length. For bubbles far outside of the membrane where the cell is less dense we would use a large move length. For bubbles inside the membrane we would use a small move length where the cell is denser. This conjecture is subject to further research.

7.1.4 Choice of Move Directions

We have chosen to consider eight possible move directions as shown Figure 7.18. For a single step move all possible adjacent grid locations can be probe. For larger move lengths not all possible neighbors would be explored.

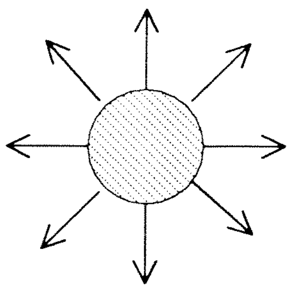


Figure 7.18 Move Directions

When a bubble and a move direction are picked by the annealer there are two possibilities: the move is blocked or the move is unblocked. If the move is blocked the annealer may choose either a new bubble and direction or the same bubble in a new direction. When the cell is in the initial configuration and not very dense, the two methods produce about the same results. However consider what happens when the cell becomes dense as it becomes more compact. Lets say 25% of the potential moves are blocked and blocked moves are spread uniformly among the bubbles. For the first method, we would find that the probability of the second move being blocked would be the same as before, i.e., 25%. However for the second method the probability of the second choice being blocked would be substantially smaller. Of course if blocked direction were highly clustered on a few bubbles the first strategy would be superior. Several experiments were conducted and we concluded that four move directions should be tried for each bubble chosen until an unblocked move direction is found. If after four directions have been tried at random, no unblocked direction has been found, another bubble is then chosen at random.

7.2 Choosing the Annealing Schedule

Choosing the annealing schedule involves choosing the initial temperature, the rate at which the temperature is decreased, and the criteria for stopping the annealing process at any temperature. The compaction of a cell is accomplished in three annealing stages where the temperature ranges for each stage overlaps. The first two stages use the central potential while the final stage does not. The first stage, akin to a wood carver roughing out the initial shape for a piece, does the global type optimization. The wood carver uses large gouges at this time where we use a large move size. The second stage uses the small move size since after the first stage the cell is fairly compact. This stage is used to form the structures inside of the membrane and move most bubbles inside of the membrane. The final stage without the central potential is to reduce the fringe effects at or outside of the membrane and form the final compacted cell. The final stage uses the wall placed at the minimum bounding rectangle to prevent any bubbles from increasing the size of the cell.

7.2.1 Stopping Criteria

An ideal stopping condition would determine when the system is in equilibrium. However this condition is extremely difficult to determine and requires a large number moves which imply a long running time. We have found by experimentation that a heuristic may be used to get reasonable results. We average the delta energy over the last n passes where a pass is a move try of each bubble once. By using the average over passes we account for cells with a different number of bubbles. When this average makes a zero crossing we stop. This produces reasonable results.

The value 8 was found by experimentation to be a reasonable choice for n. We have tried using value of 16 and greater for the average. What happens is that the cell becomes just as compact as with n equal to eight, but with a slightly longer running time for the initialization period. Therefore we have settled on a value of 8.

We have tried other methods with less success that the mentioned test. One method we tried was to average the counts of the good moves versus bad moves. We were able to use this criteria however it took a longer time to converge. Another attractive method was to guess. We assigned a fixed running time to each temperature. This seemed to work quite well. We also used a fixed number of passes at each temperature. This also worked well.

It is clear that further research remains to be done. However, it seems equally clear that the time to get reasonably good compaction is not very sensitive to choice of stopping criteria.

7.2.2 Temperatures and Cooling Schedule

The temperatures are the values which determine the probability distribution of the configuration space. The first stage of compaction starts with the highest temperature which allows the configuration space to be explored. This value is a function of the maximum wire cost and the central potential cost. During the first stage, we allow the system to cool to point where the wire costs start to take effect. The starting temperature for the second stage should be chosen so that with the reduced move size the same portion of the configuration space is explored. The second stage is cooled until a point where the wire costs are not longer dominate. The last stage is to remove the distortion of the membrane. The initial temperature for this stage is only a function of the most expensive wire cost.

The next section will define how the initial temperature is chosen, followed by how the temperatures are varied during cooling.

7.2.2.1 Choosing Initial Temperature

The initial temperature, T_i, should be high enough to allow the configuration space to be easily explored. But how big an area is needed for this explorations? We saw in Section 7.1.2 that the area for a compacted cell was seldom much more than twice the computed ideal area. We have found that choosing the initial cell area to be 4 to 8 times the ideal area provide ample exploration space. This allows the global constraints to be explored. The initial temperature is

$$T_i = \frac{W_{max}}{-\ln(P)} + \frac{C_{mem}}{-\ln(P)} \tag{7.1}$$

where W_{max} is the maximum wire cost, C_{mem} is the cost of a one step move for a bubble when the bubble is at the cell boundary, i.e., 2 to 3 times as far from the center of the cell as the ideal area boundary. The probability P should be approximately .5. A higher value for p does not hurt compaction but only requires more time for cooling.

7.2.2.2 Choosing Cooling Function

The cooling schedule we use is the function that has been used by Kirkpatrick[Kirkpatrick 83] on other problems with very good success. He used the exponential cooling function,

$$T_n = f^n * T_i \tag{7.2}$$

where T_n is the temperature for pass n and T_i is the initial temperature. The value f is less than one and controls the rate of cooling.

We have have followed the lead of other researchers in choosing a value for f in the range $.8 \leq f \leq .9$. We have found that if we cool too fast we get a poor compaction.

7.3 Complexity

We would like to know how the run time for this method of compaction grows with cell size. The importance of run time complexity is that it can limit the size of cells we can compact in a reasonable time. The effective cell size is measured by the number of bubbles in the cell. The question is how does the run time of the algorithm grow with the number of bubbles in a cell?

Before trying to answer this question we must point out some inherent limitations of this examination. First, the complexity of 2–dimensional compaction of VLSI layouts has been shown to be an NP–complete problem[Sastry 82]. For this reason, a worst case analysis would not be fruitful. The algorithm we are using is probabilistic. It is therefore natural to examine the expected complexity for the most probable cells. This probabilistic method has been shown to find the global minimum when equilibrium is reached at each temperature. To reach equilibrium can require an infinite amount of time. Since we only approximate equilibrium we can only hope to find an approximation to the global minimum configuration.

We examine the makeup of an annealing compaction. The process is divided into three stages. The first stage is where the global optimization is accomplished with large moves. The second stage is the refinement with small moves to form the final shape inside the membrane. The final stage is to reduce the distortion due to the central potential. The number of stages is independent of the number of bubbles and is the same for all cells.

At each stage we have a fixed set of temperatures where we simulate at each temperature. Within a temperature we proceed in passes until an equilibrium condition is reached. A pass is a try of a move of each bubble once. The sequence of temperatures is independent of the number of bubbles and the same for all cells. The number of passes depends on reaching equilibrium and depends on the configuration of the cell. We do not know how the number of passes grows with the number of bubbles. We will leave the number of passes for the moment. A pass is a bubble move try of each bubble once so this grows with the number of bubbles N. Therefore we know we have at least a factor of N in the run time per pass.

When a bubble is chosen the algorithm attempts to construct a bubble move in a random direction. If the move is blocked, up to three more attempts are tried before the next bubble is chosen. The more blocks there are in a cell the more time is wasted. The question is, does the blocking of bubbles grow with the number of bubbles in a cell? This really depends on the density of the cell in the area around the bubble. We will investigate how this varies with cell size.

Now examine the bubble move as described in Chapter 4. The bubble move is divided into two processes. The first is *clear path* and the second is *queue construction*. The dominant time for clear path is the search time and the time to examine the objects found. Every time a wire is found in the path another search is needed. Thus the clear path time is order $O(\log(A) * n)$ where A is the area of the initial cell and n is the average number of wires in the move path plus one. The number of primitives in the move path depends on density. Since the move size is small, n should be close to one until the cell is very compact. The dominant time of queue construction is the search time and the time for the triangle processes for each wire in the move path. Each triangle process requires a search. Again this time is $O(\log(A) * m)$ where m is the average number of wires to be moved given that the clear path has been satisfied. The number m should be close to two (since most bubbles have two attached wires) unless the cell is wire dominated or the cell is very compact. If density grows with the number of bubbles in a cell then n and m may also grow. However density is a local function that is bounded.

If the number of passes required to reach equilibrium is independent of N then the complexity of the annealing algorithm appears to be $N(\log(A))f(d)$ where $f(d)$ is a bounded function of average density. The cost of a bubble move call depends on the density of the cell. The denser the cell the more time per bubble move call. If the number of passes also grows with N then there may be an additional factor. In the next section we will run a set of experiments to determine if these estimate are correct.

7.3.1 Experiments

To evaluate our assumptions about how the complexity grows with the number of bubbles, we will run a set of timing tests with the composition of identical cells.

The base cell is the multiplexier cell from previous examples that is shown again in Figure 7.19. To form a series of cells, we will compose this cell with itself. This is easily accomplished with the binary composition described in the next chapter. We chose this method to form the series of cells with different bubble counts so that hopefully the topology of the cells would be uniform across the test set. This is to eliminate any bias from the topology.

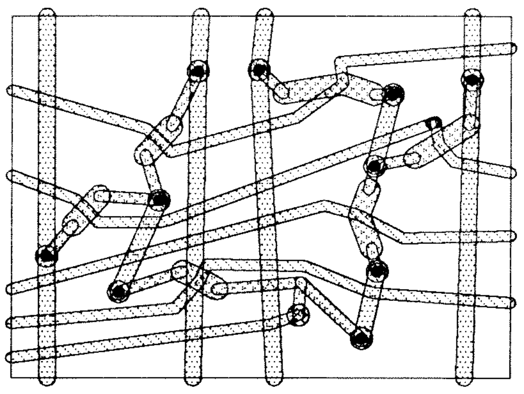


Figure 7.19 Base Cell for Timing Tests

The schematic picture of the five test cells is shown in Figure 7.20. The base cell has 60 bubbles. When two of the base cells are composed to form cell 2 the number of bubbles is not twice the bubbles in test cell 1. The reason for this is that the bubbles that are along the seam of composition are absorbs which reduces the total bubble count.

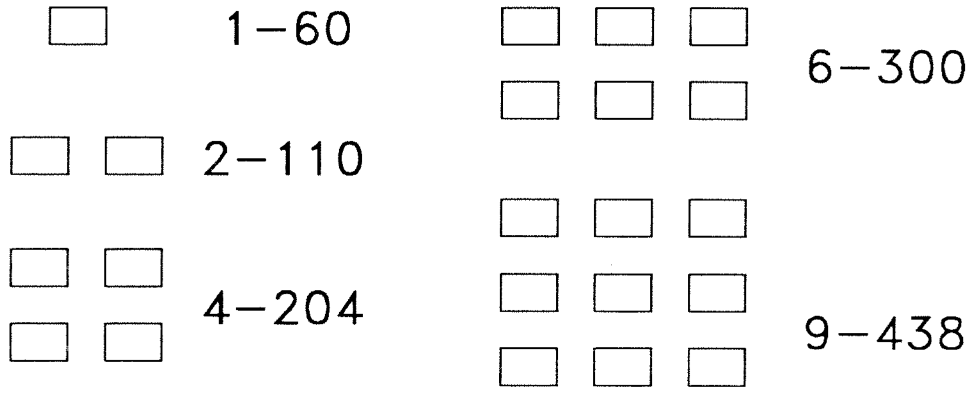


Figure 7.20 Cells For Experiment

The compaction that is used for the test set is the three stage annealing algorithm. Each cell uses the same annealing schedule. The membrane for each cell is set at the computed ideal minimum bounding rectangle.

Table 7.2 shows the compacted size and ratio of actual compacted size to computed ideal area. Appendix C contains figures showing each cell at the end of each stage. Notice that the ratio of actual to ideal area first starts out low and then increases for the second cell and then slowly decreases. We conjecture that this is because the single cell was optimized for the topology while the composition of the cells were arbitrarily composed. However, as the cell gets bigger, the pieces can use the seam area and the ratio decreases.

Cell	Bubbles	Compact size	Membrane Size	Actual/Ideal
T1	60	$1131\lambda^2$	$928\lambda^2$	1.34
		$39\lambda \times 29\lambda$	$32\lambda \times 29\lambda$	
T2	110	$3003\lambda^2$	$1595\lambda^2$	1.88
		$77\lambda \times 39\lambda$	$55\lambda \times 29\lambda$	
T4	204	$5451\lambda^2$	$3024\lambda^2$	1.76
		$79\lambda \times 69\lambda$	$56\lambda \times 54\lambda$	
T6	300	$8540\lambda^2$	$5152\lambda^2$	1.66
		$122\lambda \times 70\lambda$	$92\lambda \times 56\lambda$	
T9	438	$12272\lambda^2$	$7728\lambda^2$	1.60
		$118\lambda \times 104\lambda$	$92\lambda \times 84\lambda$	

Table 7.2 Timing Test Cells Statistics

The running time for the test cells is shown in Figure 7.21 with processor time in hours plotted versus bubble count. The run time for each cell is shown by a $\times$. The continuous line is an $N \log(A)$ plot normalized to the run time of the 300 bubble cell. The run time appears to grow a little faster than $N \log(A)$. Several conjectures we posed in the previous section might explain this growth. We will examine this issue further in a moment.

But first, we would like to consider a different question. Does the compaction time depend on topology? Some very different cells were compacted using the same annealing schedule and with their membrane coinciding with there idea area. These not only have different topology, but also include many different model structures. The run time for each is shown with a triangle above the corresponding bubble count. Notice that they follow the timing test cells. This means that running time is probably not affected by topology.

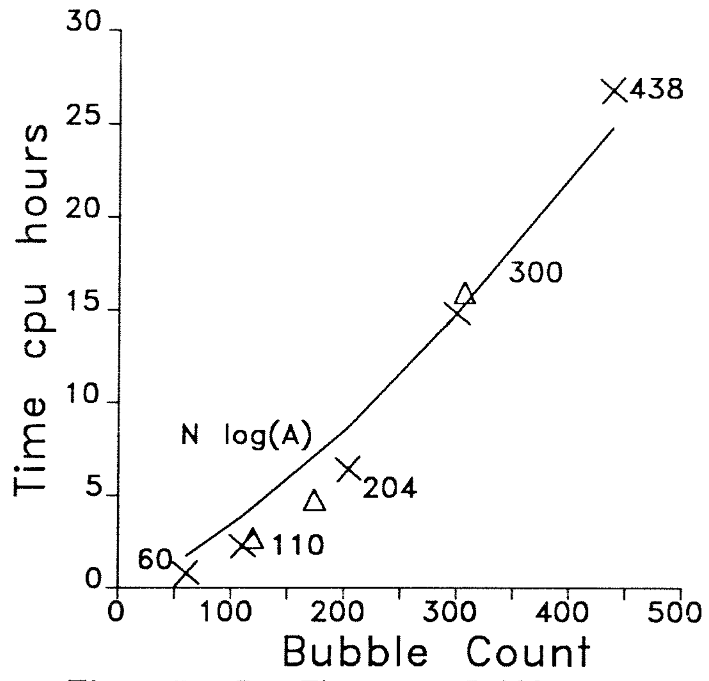


Figure 7.21 Run Time versus Bubble Count

We now return to the test set experiments. The first question that was asked in the previous section was, do the number of passes require to reach equilibrium depend of cell size? The number of passes for the first stage versus temperature is shown in Figure 7.22. The temperature scale has been normalized so that the highest temperature is one.

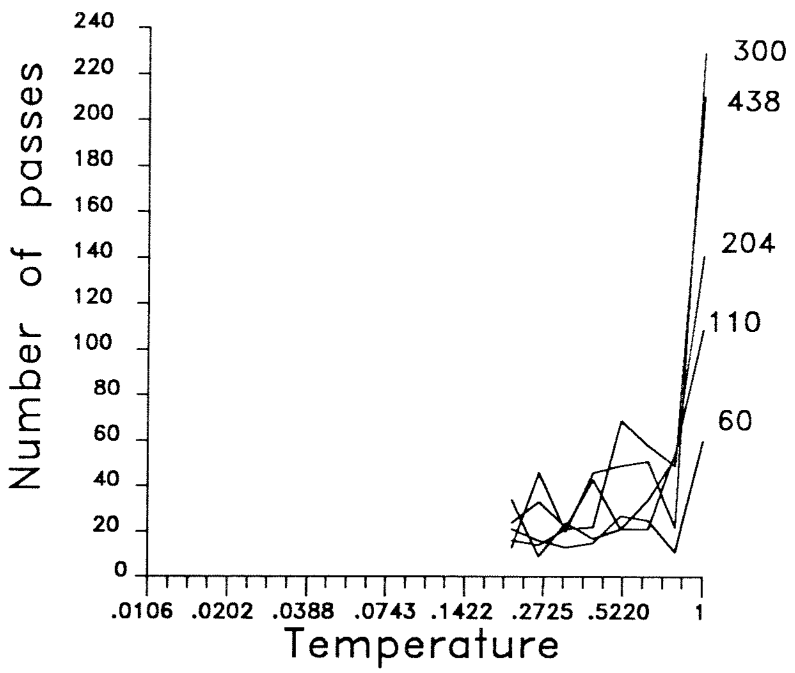


Figure 7.22 Sequence Count versus Temperature - stage 1

The number of passes grows with cell size for the first temperature as shown. A probable reason for this increase is that even at the initial temperature all the bubbles are pulled substantially toward the middle of the cell by the central potential. The time it takes to reach equilibrium will be proportional to the distance that the bubbles have to move from the initial configuration. This number distance should be expected to be proportional to the square root of the number of bubbles for cells with the same shape. Hence a larger number of passes is required. The number of passes versus bubble count is shown in Figure 7.23. The cells with 60 bubbles, 204 bubbles, and 438 bubbles have the same shape. Their growth approximately follows this pattern. The cells with 110 bubbles, and 300 bubbles are elongated and would be expected therefore to require more passes. The fact that the 300 bubble cell requires more passes than the 438 is an anomaly for which we have no explanation.

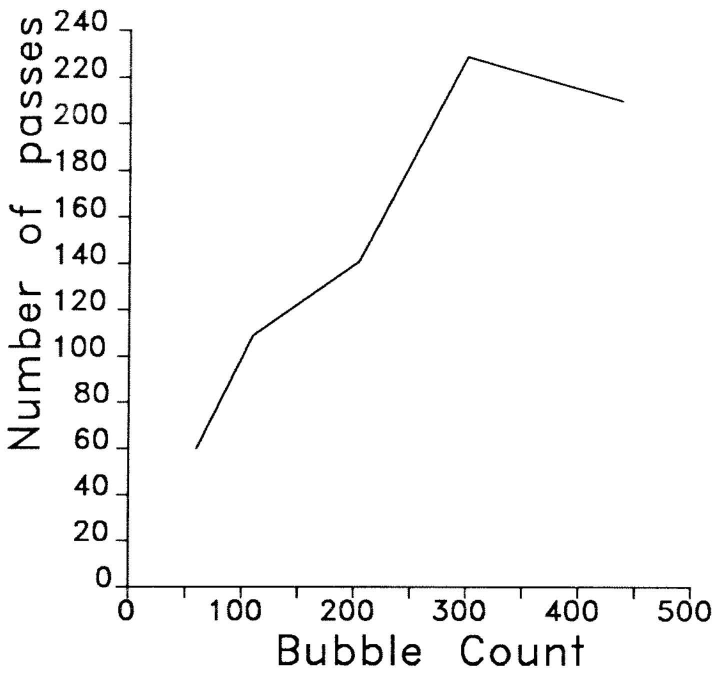


Figure 7.23 First stage, First Sequence Count versus Bubbles

The number of passes for the second stage versus temperature is shown in Figure 7.24. Again the first temperature has a high pass count. This stage has the move size set to 1 grid square. Again since the distance for bubbles to travel is longer, the first pass has a higher count for larger cells.

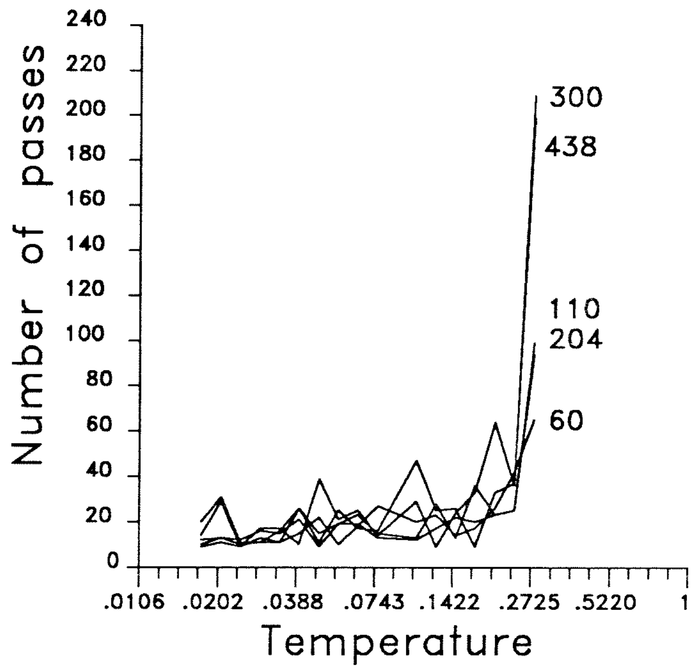


Figure 7.24 Sequence Count versus Temperature - stage 2

The number of passes for the third stage versus temperature is shown in Figure 7.25. This stage has the membrane removed and the wall set to the minimum bounding rectangle. Again the first temperature has a high pass count. Notice that there is a rise in the pass count for the cells at around a temperature of 0.0743. This is probably where there is a transition from one wire cost dominating to a different wire cost dominating.

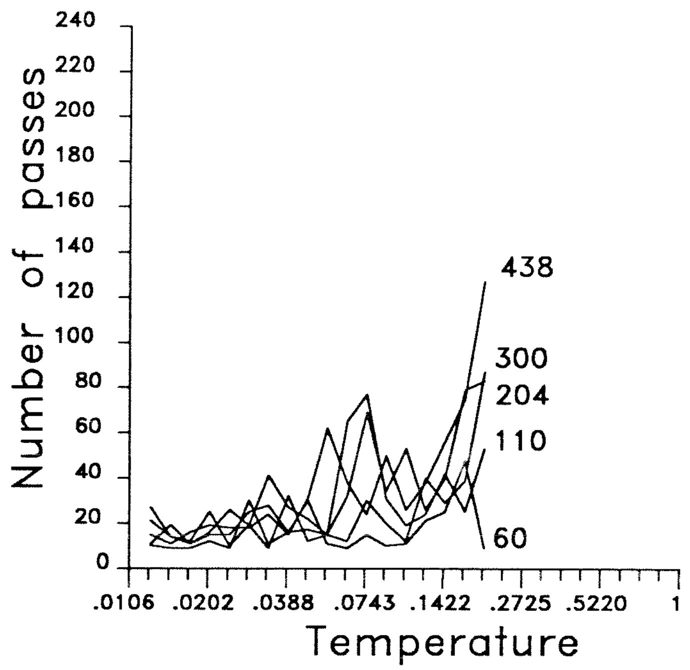


Figure 7.25 Sequence Count versus Temperature - stage 3

Since we now know the first pass is affected by cell size we will plot cpu time with the first pass of each stage removed as shown in Figure 7.26. This is closer to $N\log(A)$. However we still need to examine the difference.

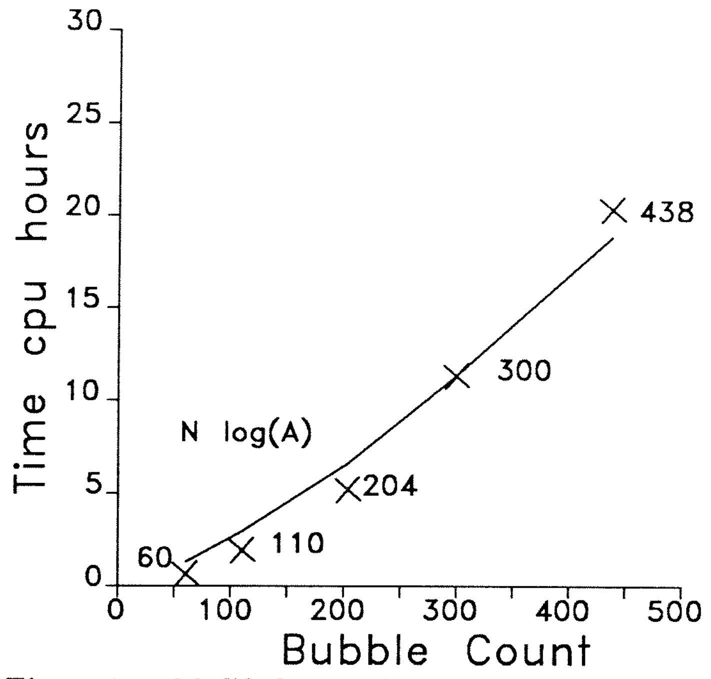


Figure 7.26 Modified Run Time versus Bubble Count

We observed in our earlier complexity discussion that run time could be affected by time wasted on attempted moves in a blocked direction. This was observed to depend on the density of the cell. We now turn to an examination of this phenomenon. Figure 7.27 shows the ratio of unblocked move calls to total move calls as a function of temperature for stage 1.

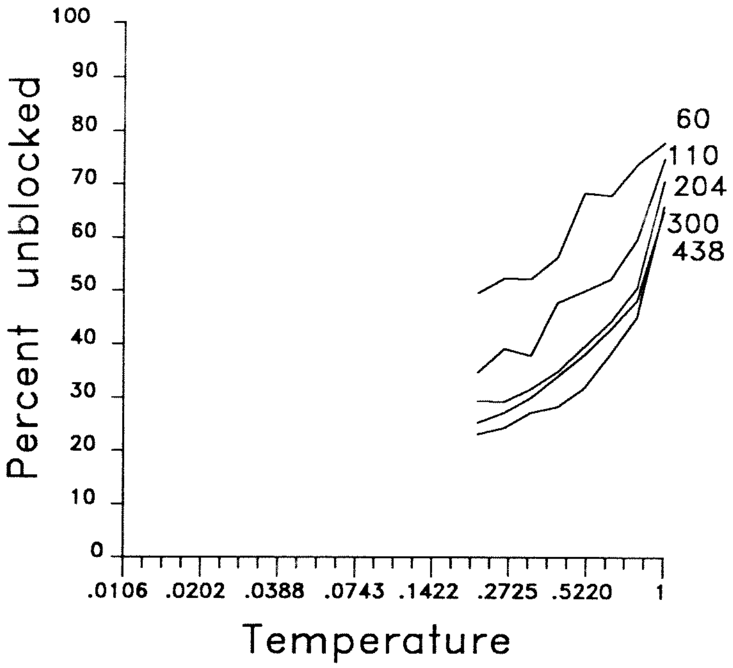


Figure 7.27 Unblocked Calls/Total Calls versus Temperature - stage 1

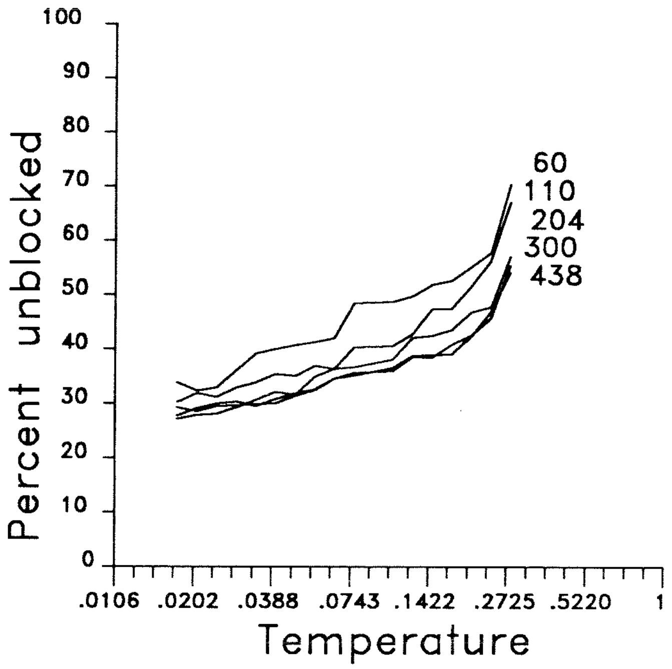


Figure 7.28 Unblocked Calls/Total Calls versus Temperature - stage 2

Figure 7.28 and Figure 7.29 show this ratio for stages 2 and 3 respectively.

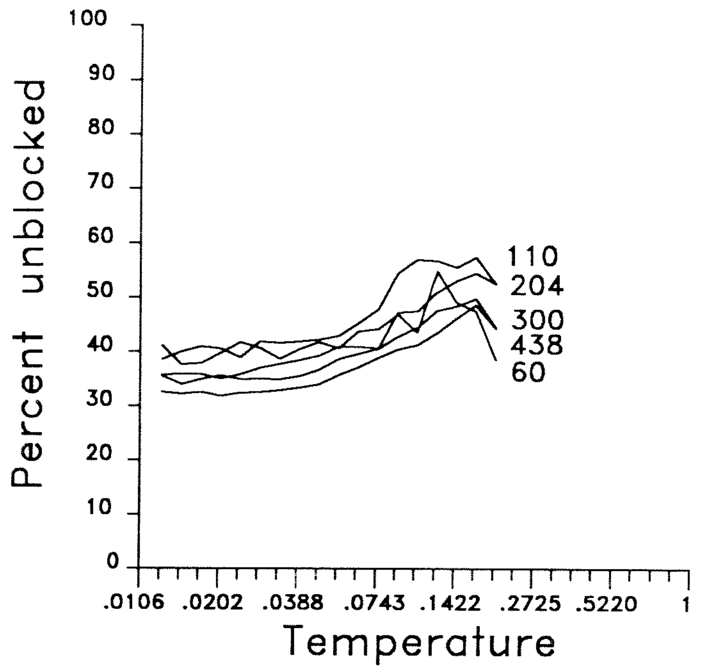


Figure 7.29 Unblocked Calls/Total Calls versus Temperature - stage 3

Observe that the percentage of unblocked calls monotonically decreases as the number of bubbles increases throughout the temperature ranges for stages 1 and 2. For stage 3 the same observation holds except that the base cell takes longer to find its place.

This is a strong indication that for these tests the average density increased as the number of bubbles increased, i.e., with cell size. At first we were surprised by this. But then we understood that this was caused by the increased distance of the membrane inside the final compacted cell area. The central potential outside the membrane was designed to grow much faster than inside in order to pull in the remote bubbles. It was clear that this added pressure was causing the cell to become more dense for larger cell sizes.

The same phenomenon is reflected in the plot of average move call time shown in Figure 7.30 for stage 1. The average move call time increased monotonically with cell size. The spike found on the plot of run time for the 438 bubble cell is an anomal,y probably caused by garbage collection.

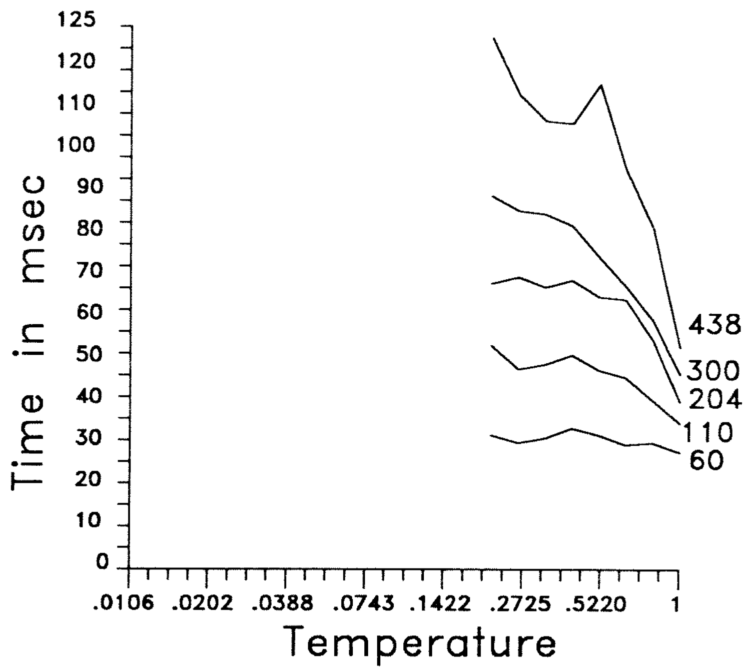


Figure 7.30 Average Move Call versus Temperature - stage 1

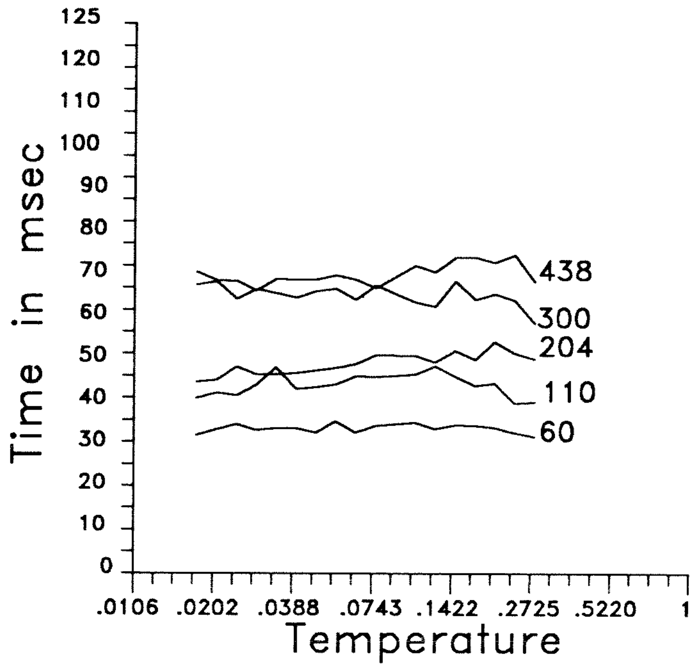


Figure 7.31 Average Move Call versus Temperature - stage 2

Figure 7.32 and Figure 7.31 show average move call time for stage 2 and 3 respectively.

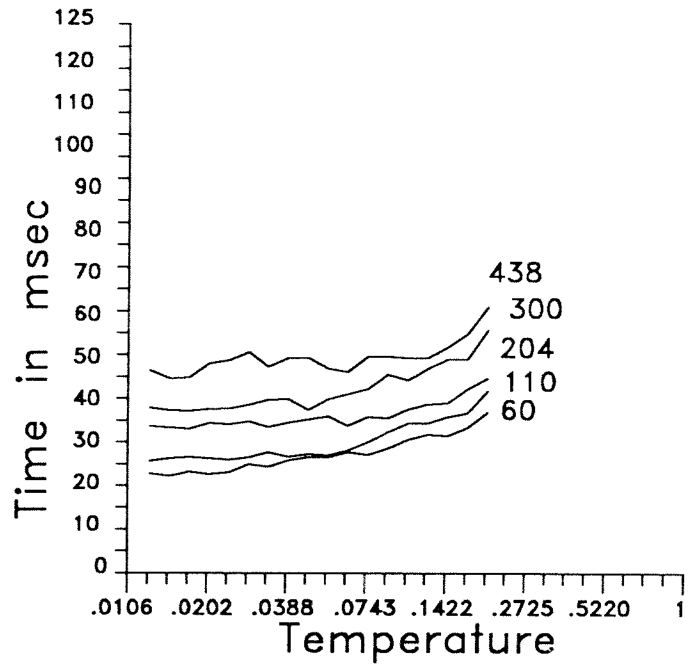


Figure 7.32 Average Move Call versus Temperature - stage 3

Again the effects of increase density are clearly seen.

7.3.2 Summary

It is clear from both the experiments on membrane position and complexity that the central potential is the most critical parameter affecting the run time of this algorithm. It is expected that further research will produce a method for controlling the central potential that will provide outstanding compaction while limiting the growth of run time complexity to order $O(N \log(A))$.

7.3.3 Run Time Enhancements

We conjecture that the run time can be improved by using two techniques. The first method is to vary the move length based on distance from the membrane. From previous sections we know that as the density increases the bubble move time increases. During the compaction process the density decreases from the inside of the membrane to the outer parts of the cells. We can take advantage of this property by using large moves outside of the membrane and small moves inside. Using this method we could combine stages 1 and 2 into a single stage. This would remove one heating cycle. It is expected that this will reduce the overall cpu time.

The second method is to make the choice of which bubble to pick next in the simulation process based on the activity of the bubbles. The activity of a bubble is the average delta cost this bubble has during simulation. We then could keep the bubbles in an order list with the highest activity bubbles first. The random choice function could then be biased to pick bubbles in the list with larger activity. Thus active bubbles would be picked more often while inactive bubbles would be picked less often. It is expected that this will reduce the cpu time.

7.4 Cell Experiments

The cells for the experiment were chosen for their diversity. The first two cells are hand designed curvilinear cells. These two cells were designed by Don Speck at Caltech. A lot of time was spent on optimizing the topology and the size of the cells. The second cell is a composition of the first cell with other cells. The next four cells were taken from the quanternary multiplier chip[Frey 83] that were compacted using the Rest[Mosteller 81] system with human directed compaction. An average of a weeks worth of time was spent compacting each cell. These cells use orthogonal geometry. The first couple of cells use models with a small number of bubbles while the later cells use models with a large number of bubbles. The results of the experiment are shown in Table 7.3.

Cell	Bubbles	Hand Size	Compacted Size	Actual/Ideal
Mux1	60	$1376\lambda^2$	$1216\lambda^2$	1.34
		$43\lambda \times 32\lambda$	$38\lambda \times 32\lambda$	
Mux4	200	$5893\lambda^2$	$5396\lambda^2$	1.63
		$83\lambda \times 71\lambda$	$76\lambda \times 71\lambda$	
C2	119	5891λ	$4760\lambda^2$	1.68
		$137\lambda \times 43\lambda$	$40\lambda \times 119\lambda$	
Cg	174	$5447.75\lambda^2$	$4340\lambda^2$	2.09
		$141.5\lambda \times 38.5\lambda$	$35\lambda \times 124\lambda$	
Bil	307	$14490\lambda^2$	$12093.75\lambda^2$	1.88
		$140\lambda \times 103.5\lambda$	$93.75\lambda \times 129\lambda$	
Stuff	302	$16880\lambda^2$	$14000\lambda^2$	2.26
		$105.5\lambda \times 160\lambda$	$100\lambda \times 140\lambda$	

Table 7.3 Compacted Cells Statistics

Notice that in all cases the automatically compacted cells are smaller. Observe that the cells mux1, mux4, and c2 have a low actual area to ideal area ratio. This implies that these cells are mostly devices. The cells cg, bil, and stuff have a larger ratio which implies more cell wiring. The initial version, and hand versus automatic compaction figure are shown next.

An important point to notice about the cell figures is that the cells could be compacted smaller if some topological changes were achieved.

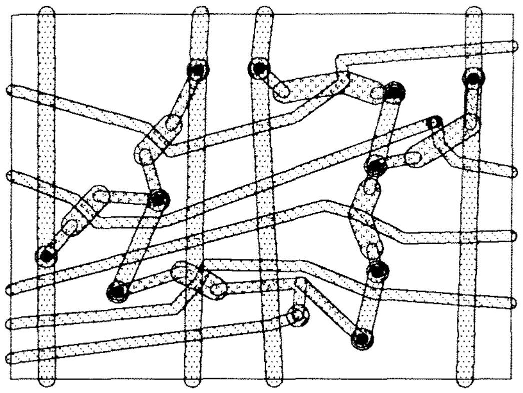


Figure 7.33 Initial Mux1

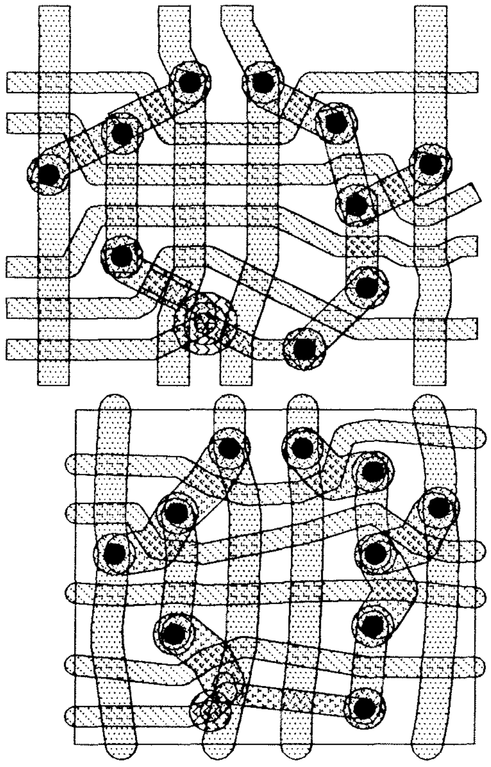


Figure 7.34 Mux1 - Annealed Compacted Compared to Hand Compacted

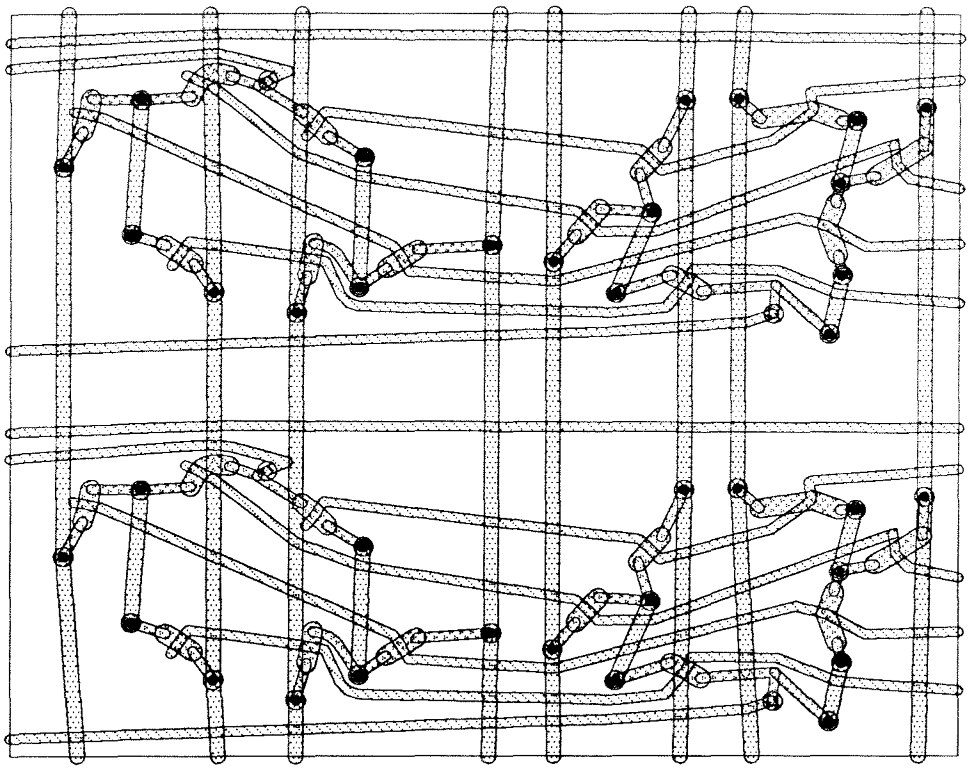


Figure 7.35 Initial Mux4

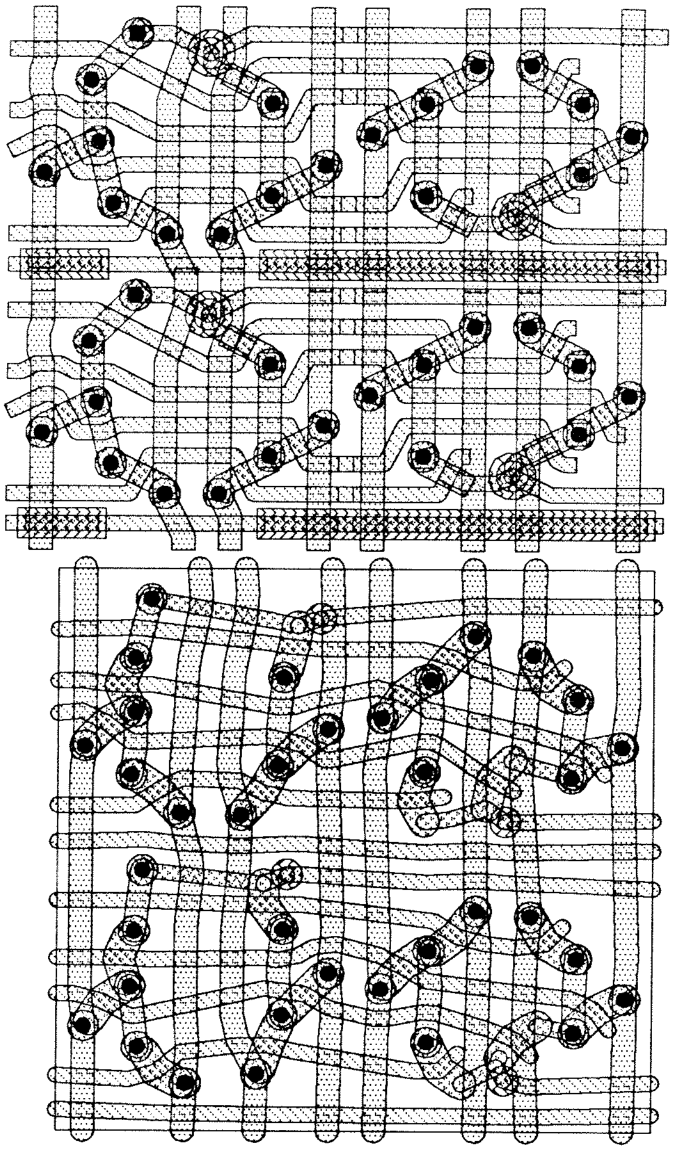


Figure 7.36 Mux4 Annealed Compacted Compared to Hand Compacted

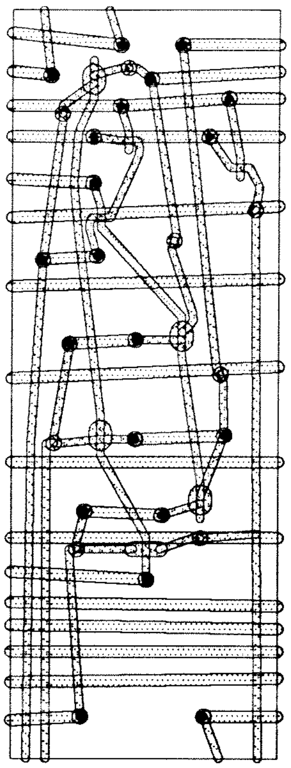


Figure 7.37 Initial C2

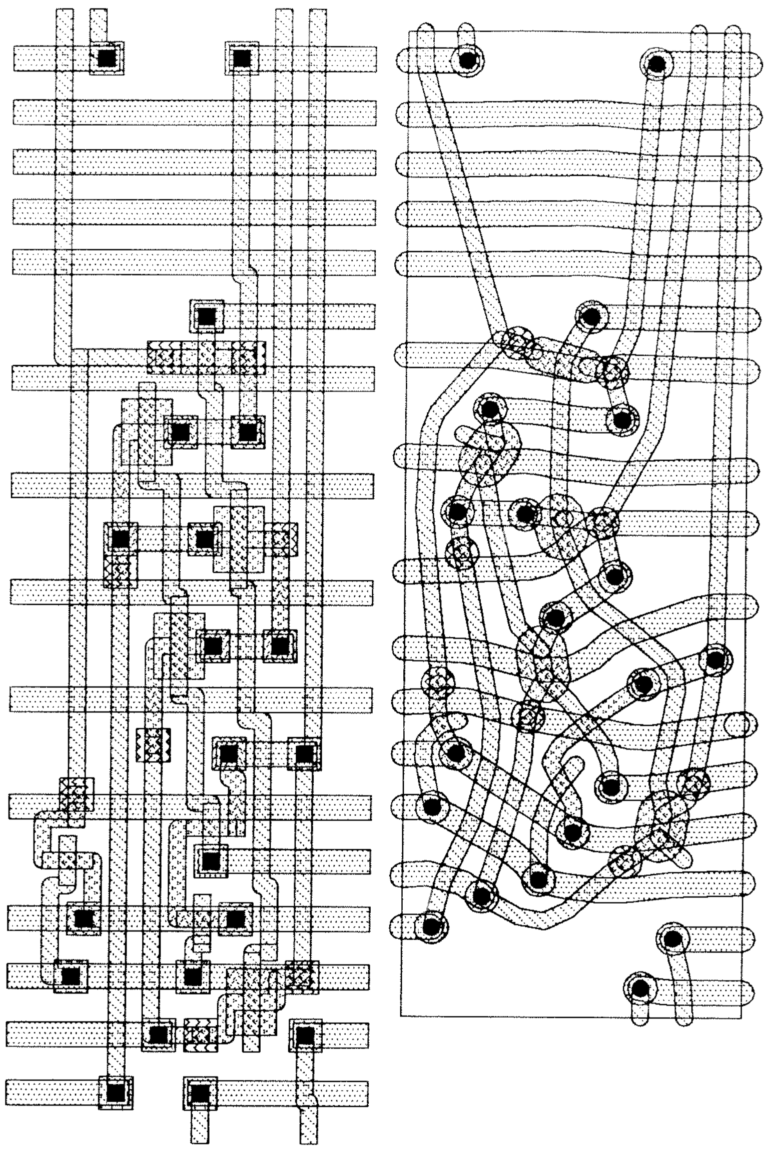


Figure 7.38 C2 Annealed Compacted Compared to Hand Compacted

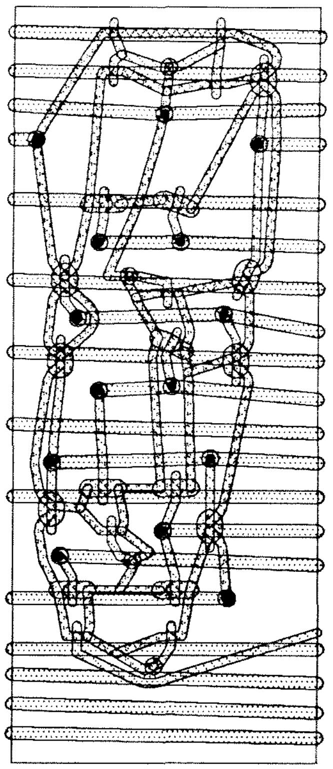


Figure 7.39 Initial Cg

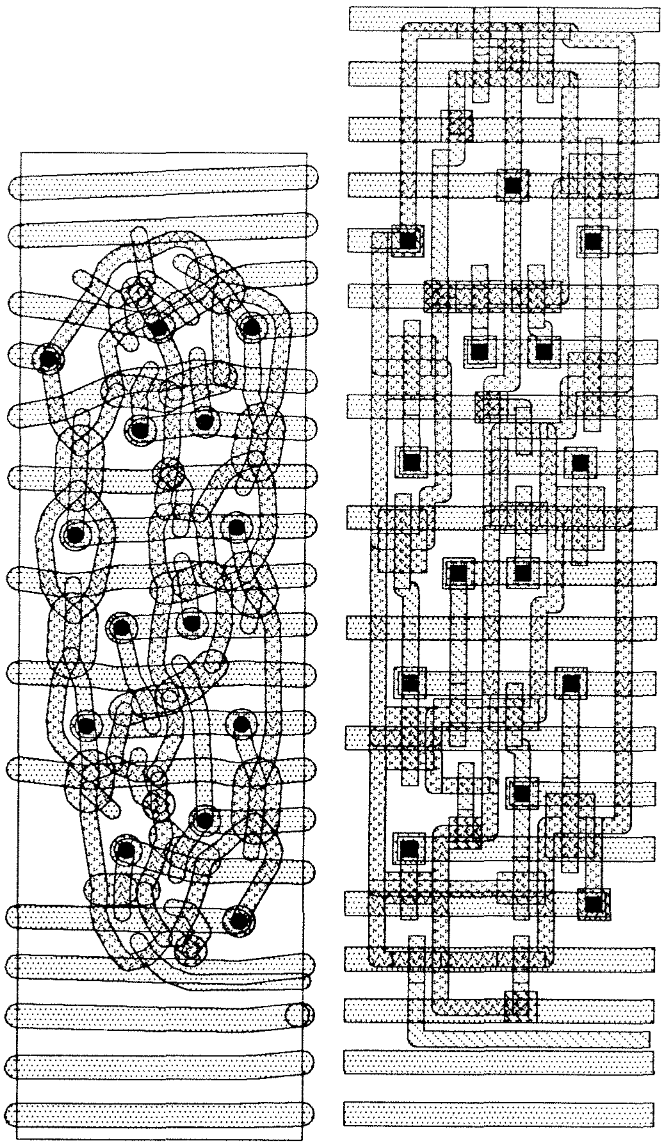


Figure 7.40 Cg Annealed Compacted Compared to Hand Compacted

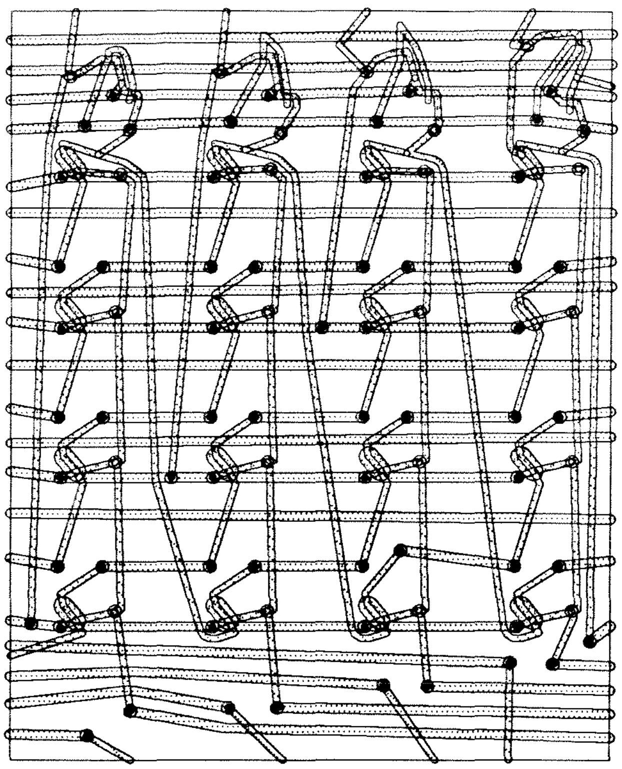


Figure 7.41 Initial Bil

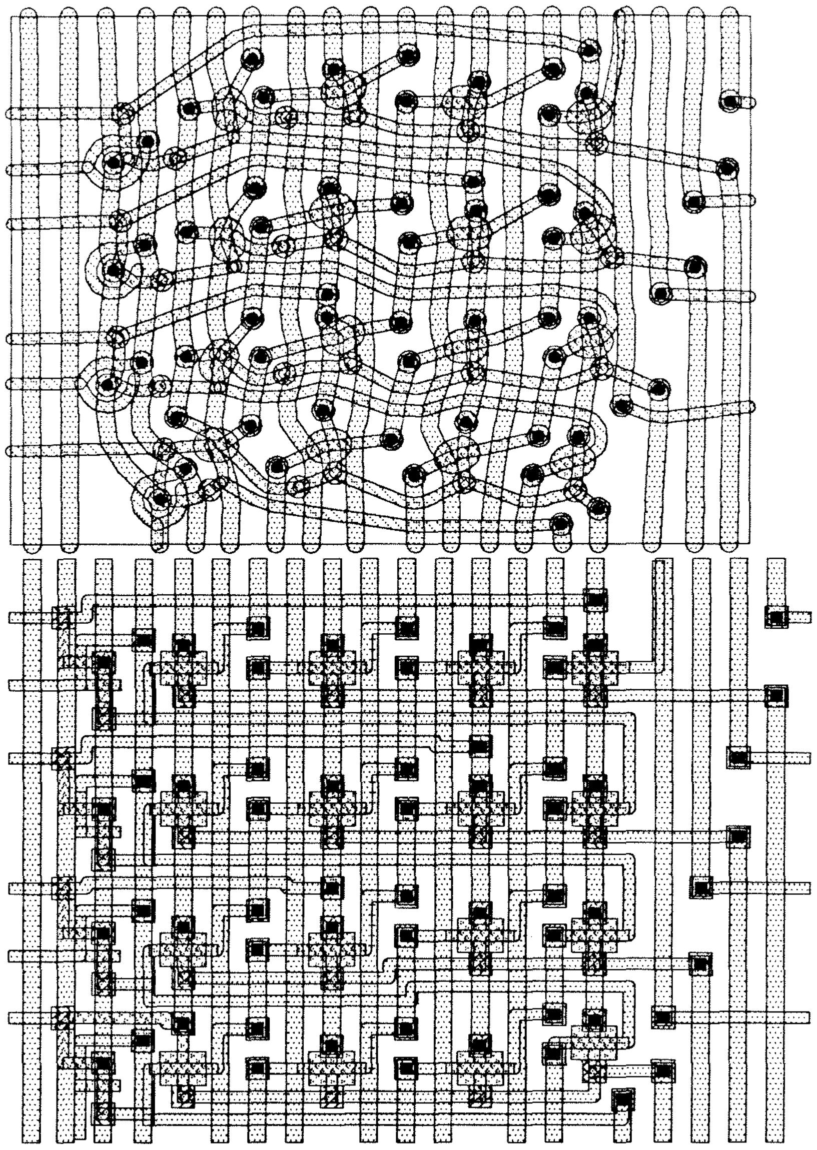


Figure 7.42 Bi1 Annealed Compacted Compared to Hand Compacted

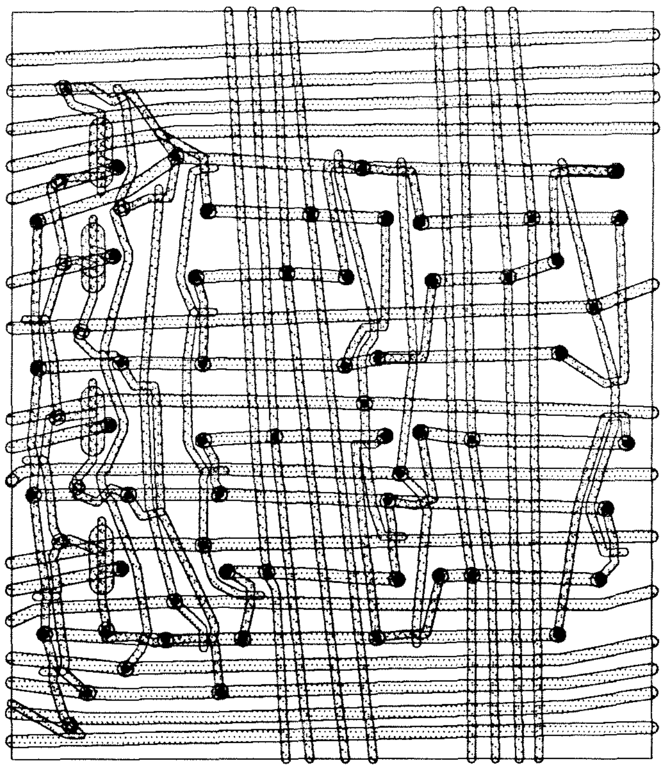


Figure 7.43 Initial Stuff

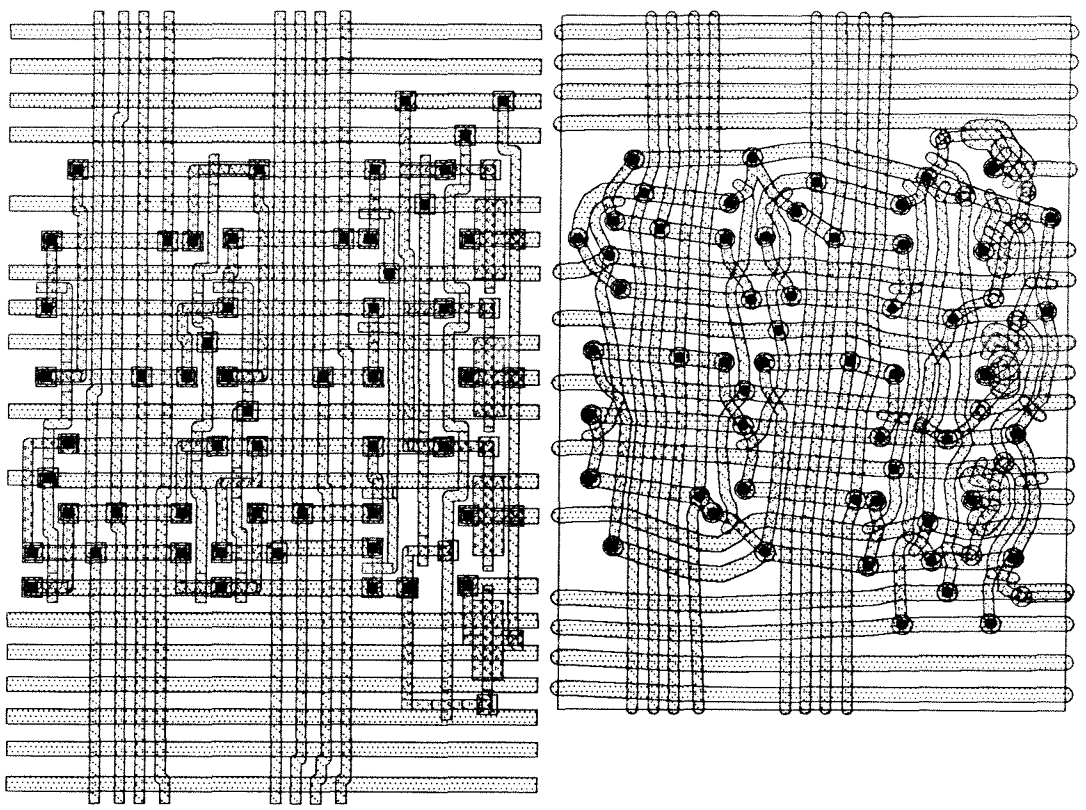


Figure 7.44 Stuff - Annealed Compacted Compared to Hand Compacted

7.5 Fabrication Experiment

A chip was fabricated to test techniques described in this thesis. The goal of the experiment was to show that a design using the models of this framework in curvilinear compacted cells could be built and would perform as designed. A one-bit adder was designed using combinatorial logic with no pass gates. The reason for using such a simple cell was to minimize the chance of error. The logic diagram is shown in Figure 7.45.

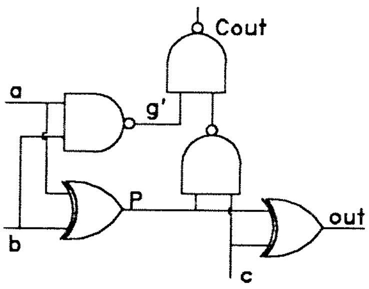


Figure 7.45 One-Bit Adder Logic diagram

Three cells were fabricated on one chip. The initial version of the one-bit adder is shown in Figure 7.46. The first two were two versions of the one-bit adder, compacted. The third was four one-bit adders stacked and compacted. The four-bit compacted cell was smaller than four times a single bit cell.

The cells were assembled using the "Fusion" [Ayres 85] process from Mosis [Mosis 84]. The chip used a 4 micron nMOS process which is very conservative. The design of the cells and assembly took about two days.

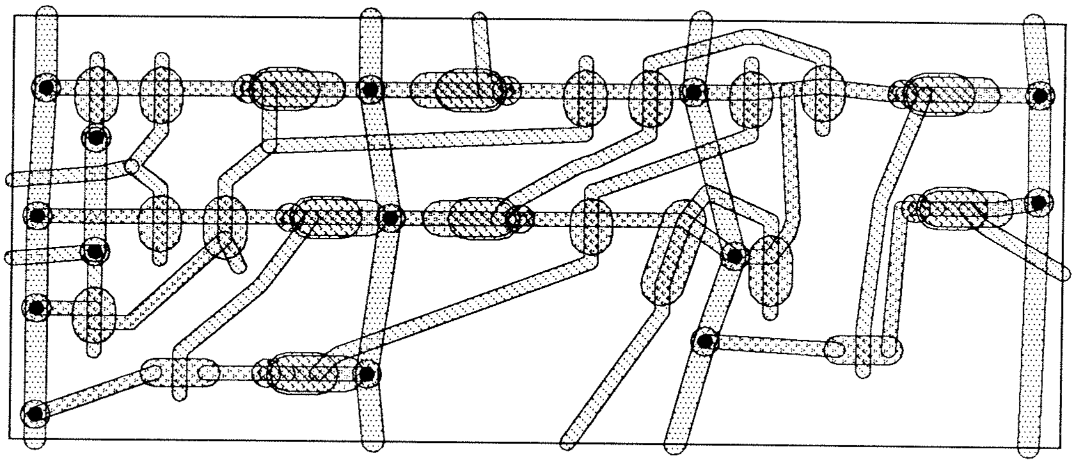


Figure 7.46 Uncompacted One-Bit Adder

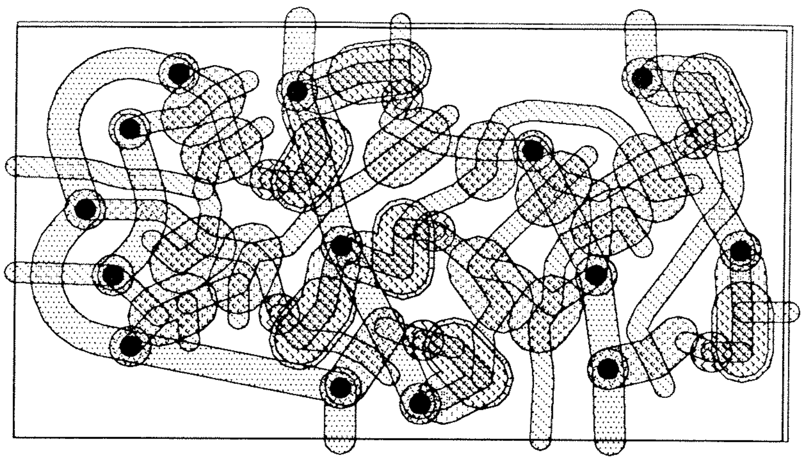


Figure 7.47 Compacted First One-Bit Adder

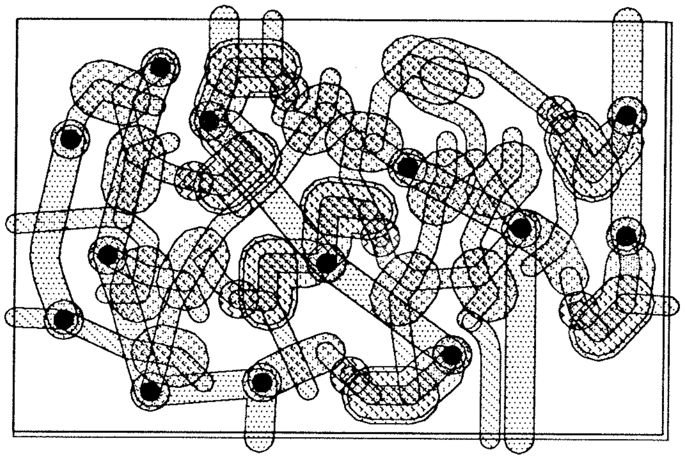


Figure 7.48 Compacted Second One-Bit Adder

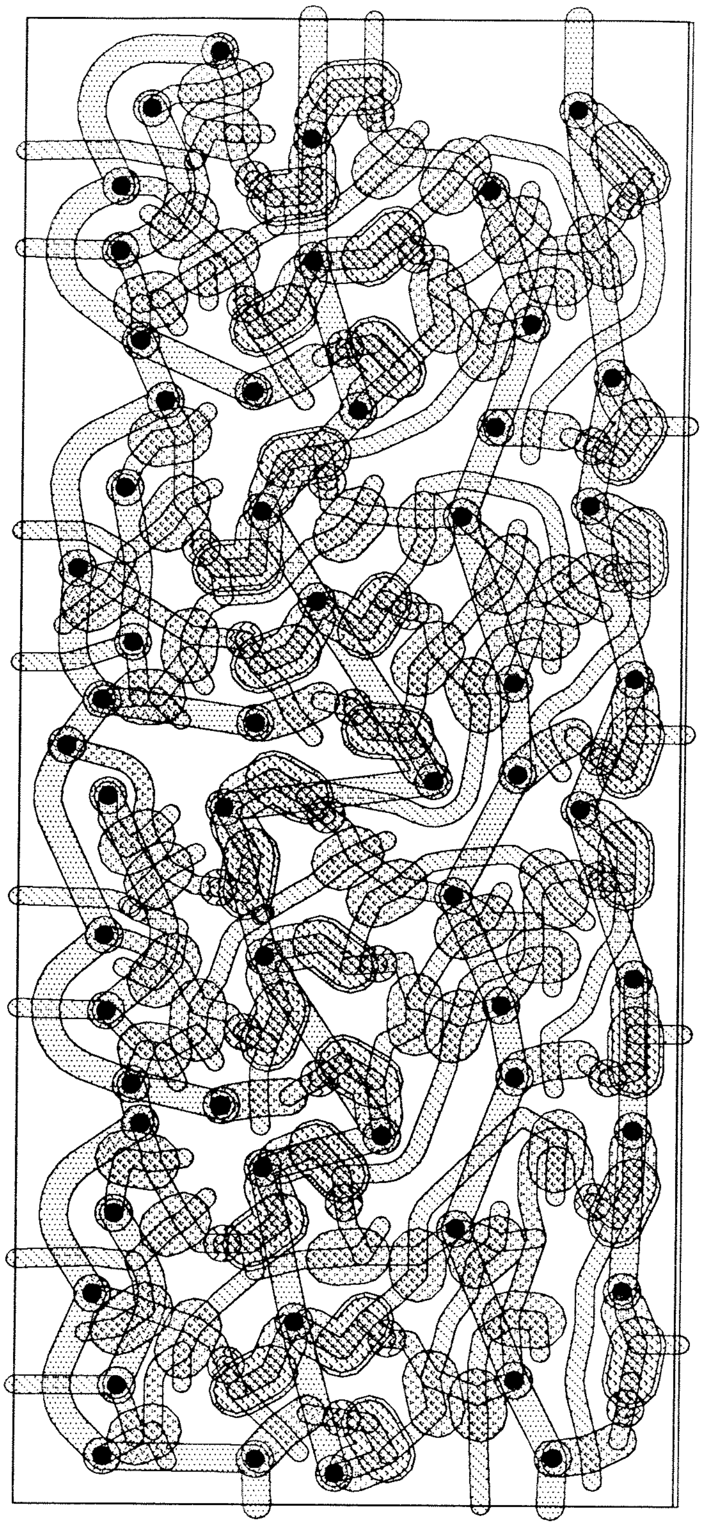


Figure 7.49 Compacted Four Bit Adder

8
Cell Composition

Composition is the assembly of cells as directed from the floor plan to form an integrated circuit. Composition has generally been a tedious manual task. There are automatic checking aids to insure a composition correct design. There have been some past composition tools that do automatic composition at the expense of chip area. In this chapter we will introduce a binary composition operator that assembles two cells with minimal area increase and correctness with respect to geometric design rules and connectivity requirements. This operator can then be used to assemble the chip. The lowest level cells, called leaf cells, are combined by the binary operator to form composition cells which are further combined to form a complete chip.

In past compaction systems [1] leaf cells are compacted generally without regard to composition. We will show how to direct the compaction of leaf cells from the floor plan that allows an optimal composition of the VLSI chips. Further reduction of area can be achieved with the method of ultra compaction to remove the seam area of composition.

8.1 Specifying a Floor Plan

A floor plan is a structured hierarchical description of the cell tiling for a VLSI chip. The structured design methodology, as described in Section 2.1, with the restricted hierarchy, supports good floor planning techniques. With this restricted hierarchy, there are two types of cells: leaf cells contain the geometric representation, and composition cells define the interface between cells. With this method, cells are connected by abutment where there is a one-to-one mapping of the connectors along the interface. The composition strategy insures that connectors align and no geometric design rules are violated.

The first step in the design of a VLSI chip is to establish the major functional blocks which in turn are translated to an initial floor plan of major blocks. At this

[1] See Chapter 2.

time the wiring strategy is defined. Each block is then refined into its sub-blocks until the leaf cell level is reached. Refinement continues until a desired floor plan is achieved. This floor plan defines the composition for the VLSI chip. After the floor plan is completed, the leaf cells are designed.

We will define the floor plan with a composition language as in Rcomp[Mosteller 82] which has a binary composition operator. A floor plan described with this language has a binary tree representation for the composition as shown in Figure 8.1 for a sample composition of cells A, B, and D. A composition specification for the example would be

$$(B/(A:A)):D$$

where : represents horizontal composition and / represents vertical composition. The language has the ability to rotate cells or composition of cells. The language also has a special operator for omitting ports to allow a generic cell to be used at various places.

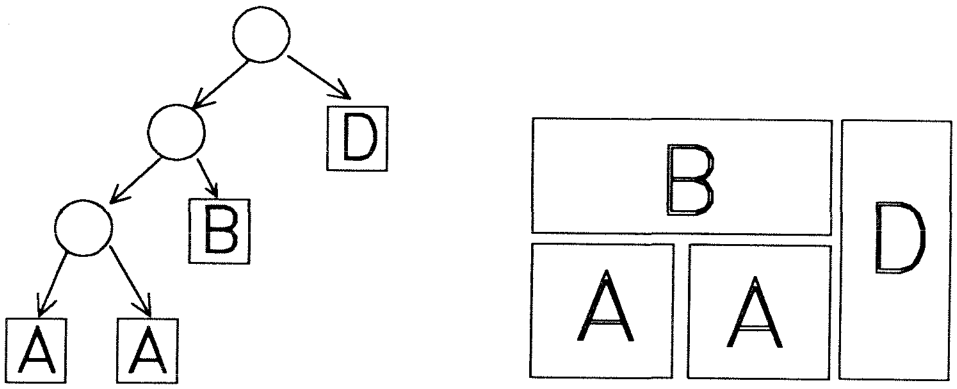


Figure 8.1 Sample Floor Plan and Composition Tree

The advantage of specifying a floor plan with a composition language is that it is easy to specify, easily updated and representation independent. A large VLSI integrated circuit chip can easily be specified with only a few pages of code or possible one. This code can easily be updated with a text editor. No change is required to the composition specification for cell wiring changes provided the cell tiling has not changed. Since the composition specification is representation independent, the composition may be used by other tools such as functional simulators provided the leaf cell representation needed for the tool has been defined.

As we have shown the cells are limited to rectangles. This restriction may be eliminated by modifying the language to compose sides which could be used for the polygonal cell described in Section 3.9.

8.2 A Binary Composition Operator

A binary composition joins two cells along cell boundaries to produce a cell where electrical connectivity between boundary ports is insured and no geometric design rule errors are introduced. This method of binary composition joins two cells with the minimum possible separation distance without generating any design rule violations. A requirement for the sides to be joined is that there is a one-to-one mapping of ports with proper type. The port type is the type of signal on the port where an example would be "VDD" for a power type. The ports do not need to physically line up. The composition operator pitch matches the two cells to be joined, aligns the ports, and adjusts cell sizes so no geometric design rule would be violated upon composition. The two cells can then be placed in juxtaposition on the cell boundaries where the ports on the joining side will be on top of each other to insure electrical connectivity. Recall from the definition of a cell in Section 3.9 that the port centers are coincident with the cell side.

Consider the two cells in Figure 8.2 that are to be composed horizontally. The first step is to pitch match the cells if they are not already pitched matched. The pitch matching is achieved, first by centering as close as possible the centroid of the ports of the smaller cell side with the centroid of the ports of the larger cell side without going over the size of the the larger cell. The smaller cell sides are than expanded as described in Section 4.3.1 to the size of the larger cell. The result being the two cells are pitched matched. Notice in the example that the pitch matching is not necessary.

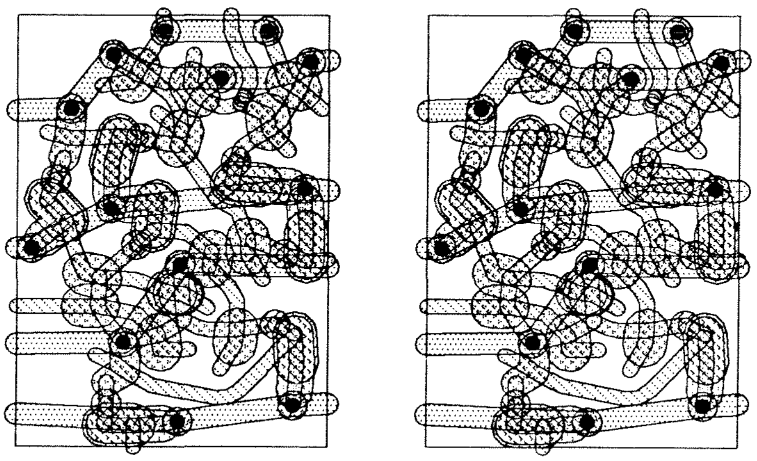


Figure 8.2 Cells to be Composed

The next step is to align the ports along the cell edges to be composed. Port alignment is accomplished by first expanding the cells on the corresponding sides by an amount that allows the ports to be moved. This amount can be calculated from the number of ports. The ports are then aligned by the move algorithm described in Chapter 4 and annealed for minimum cost. This allows the wiring for the cells to be internal to the cells, if possible. Next the cell composition sides are constricted to the minimum bounding rectangle as described in Section 4.3.2. This constriction can be seen in the example of Figure 8.3.

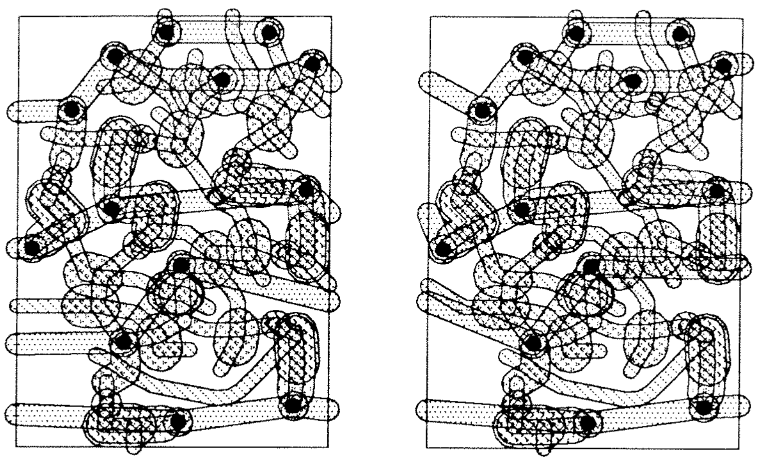


Figure 8.3 Port Aligned Cells

The next step is to determine the minimum distance, D_b, needed between the cell boundaries to avoid all geometric design rule errors upon composition. The minimum distance, D_b, is calculated by a design rule check among the boundary primitives in each cell. A boundary primitive is one that is closer to the cell side to be composed than the maximum of the minimum separation distances for the primitives.

First an equivalence mapping is constructed for the related numbers[2] between corresponding ports. This mapping can be considered as a set of ordered pairs of equivalent related numbers. We will use a modified related function which uses this mapping to determine if two primitives are related. Next a geometric design rule check is done between the boundary primitives of the cell where the minimum distance, D_b of the two cells is calculated. D_b is the maximum of all the checked primitives orthogonal minimum separation distance, M_o, minus the orthogonal distance to the cell edges. M_o is the minimum orthogonal separation distance for the two primitives along the axis being composed. The primitive positions are transformed for the purposes of the check to a point where the cell edges are coincident. Due to the nature of the primitives all combinations are not necessary.

Only the primitives that are within the maximum separation distance from the joining cells edges need to be checked. By constructing a search tree as described in Section 4.4 with only these primitives the search time is order $O(\log(a))$ where a is the area defined by the cell edge and the maximum separation distance. The construction of the search tree is order $O(\log(A))$ where A is the initialized cell area. This check is easily performed and quite fast due to the limited search.

The final step is to expand each cell if necessary by half of the calculated minimum separation distance. The result of the binary composition is shown in Figure 8.4.

[2] Related numbers are described in Section 3.6.

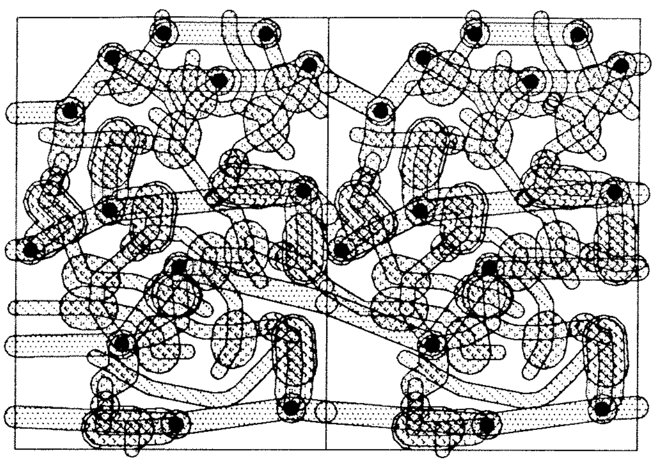


Figure 8.4 Composed cells

The cells may now be joined at their corresponding edges to form a new cell without any geometric design rule errors and connectivity between joined cell ports insured. Alternately an abstraction cell of the two cells can be constructed with the proper related equivalence number functions. This decision is a matter of the implementation method.

It is important to realize the advantage of this method. First, the cells are joined with the minimum possible separation distance without violating any geometric design rule error. Past systems, as described in Chapter 2, did not do this automatically. This composition could only be done manually with careful scrutiny. Second, the ports are aligned with minimum wire cost without stretching the cells. Previous methods stretched the cells to align the ports, which increased the overall resulting area and possibly destroyed the integrity of the circuit. This stretching could drastically increase the final chip area. The final result is a cell that is geometrically design rule correct, properly connected, and minimum composed area. This area could further be reduced by the method in Section 8.5.

8.3 Composition Compaction Controls

A compaction algorithm is quite limited unless the final shape of a cell can be directed and also the final position of the ports within the cell. This control is used by the composition process. Our compaction algorithm allows the setting of the membrane position which governs the final shape of the cell and allows various controls over the port positions.

There are three methods to control the port position where they may be combined on a single port basis. The first is that of "locking" the ports at the desired position. This locking is accomplished by using the bubble move algorithm to position the ports and marking them as "frozen". Frozen ports are not allowed to be moved by the annealer. During the compaction process the body of the cell would conform to the port position. The second method is to use a linear cost from the port to the desired position or to another port. The annealer will try to minimize this cost. The final control is the ability to lock two ports together so they move as a unit. The compactor would treat the two ports as one. This port coupling is used for cells that are going to be replicated together.

The shape of the cell is controlled by the shape of the membrane where the membrane is the desired final region for the cell. The annealer will try to compact the cell to the membrane region. The minimum area the which membrane may be set to is the ideal area, A_{ib}, as described in Section 3.9.2. The minimum length for the vertical side of the membrane is V_m which is the minimum vertical side distance based on the vertical ports. The minimum length for the horizontal side of the membrane is H_m which is the minimum horizontal side distance based on the horizontal ports. Within these limits, the membrane may be set by the user. An automatic choice can be done by the compactor for a cell which will be based on the ideal area. In the next section you will see how this choice is accomplished based on the floor plan.

The degree of flexibility of the membrane can be seen in Figure 8.5. The cell was compacted for a long flat cell on the left while the same membrane was rotated 90° for the cell on the right. Due to the constraints of internal circuitry the cell could not be tight for the tall and narrow case. However both of these cases had a minimum bounding area of approximately 1.4 times the ideal area.

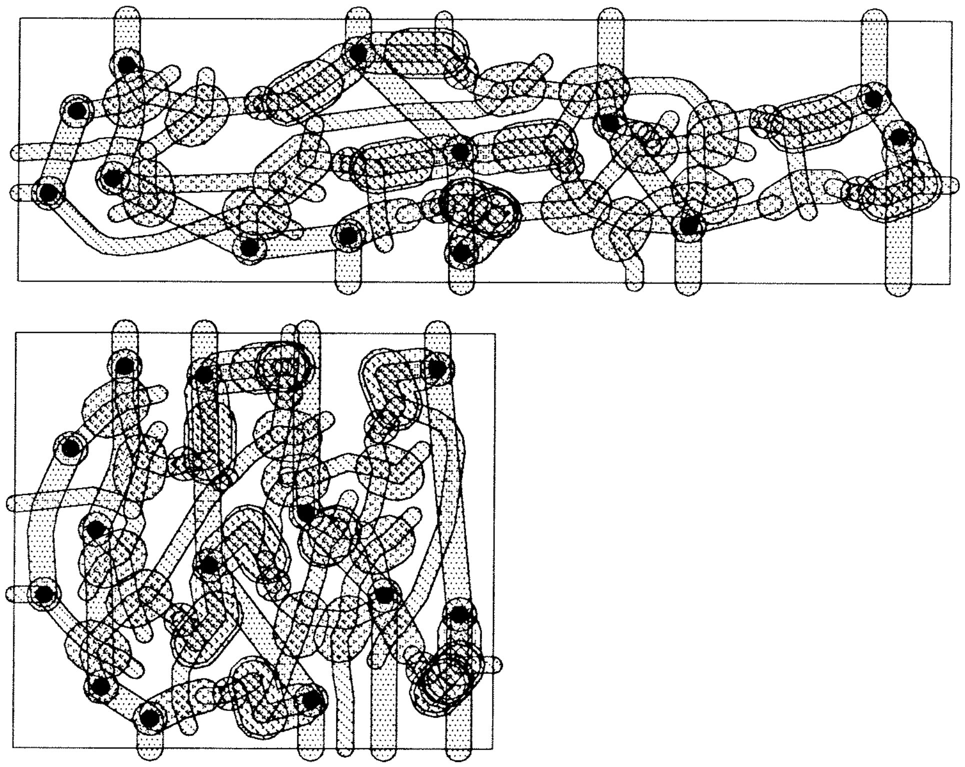


Figure 8.5 Sample Membrane Control

8.4 Composition Directed Cell Compaction

In this section we will show how the cells of VLSI chips are compacted as directed from the floor plan. In past composition and compaction systems, as discussed in Chapter 2, the cell compactions were not directed from the floor plan. To gain an optimum VLSI chip the cells must be compacted to be in harmony with each other. This compaction is accomplished by a top down directed compaction of the cells from the floor plan.

Assuming the floor plan has been specified and the cells have been designed the first step is to determine the ideal area, A_{ib}, the minimum vertical length, V_m and the minimum horizontal length, H_m, for the top level cell of the floor plan. The user now specifies the desired membrane for the top level cell within these limits. The next step is to partition the membrane to the leaf cells for compaction. After compaction, the cells are composed to form the top level cell. At this time the "ultra" compaction method described in the next section may be used to further reduce the area.

It is assumed that the leaf cells at the leaves of the tree are instances of the designed cell. Since a cell may be used at several locations in the design, there can be several unique compactions of the same cell. Later in this section we will show how a replicated structure of a single cell type is treated as a single cell.

8.4.1 Cell Size Estimation

Before the cells can be compacted, the membranes will need to be defined for the individual leaf cells. This definition is accomplished from the floor plan and the user's input on the desired final shape. The first step is to pass the cell's ideal minimum area up the composition binary tree to the top cell. This is shown in Figure 8.6 for sub-tree for cells *a* and *b* and the composition cell *c*.

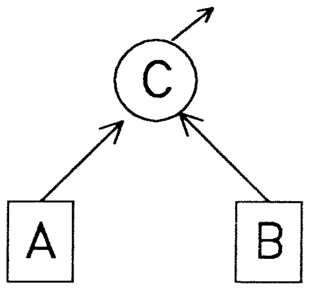


Figure 8.6 Estimate Sub-Tree

The ideal area, A_{imc}, minimum vertical length, V_{mc}, and horizontal length,

H_{mc} for composition cell c is

$$V_{mc} = \begin{cases} V_{ma} + V_{mb}, & \text{if vertical composition;} \\ \max\left(V_{ma}, V_{mb}\right), & \text{if horizontal composition.} \end{cases}$$
$$H_{mc} = \begin{cases} H_{ma} + H_{mb}, & \text{if horizontal composition;} \\ \max\left(H_{ma}, H_{mb}\right), & \text{if vertical composition.} \end{cases} \tag{8.1}$$
$$A_{imc} = \max\left((A_{ima} + A_{imb}), (V_{mc} * H_{mc})\right).$$

Notice that the ideal area is the maximum of the sum of the two cells ideal area and the product of the vertical and horizontal lengths. This calculation is to account for any mismatch between cells. This process of determining the ideal area for a low level cell is repeated until the top cell ideal area is defined.

The next step is for the user to specify the top level cell membrane. The choice may be made within the limits of the ideal area, vertical minimum length and horizontal minimum length. Generally the designer will have a good idea of the desired final shape.

The next step is, starting at the root node of the floor plan, to partition the membrane to the lower cells. This partitioning is done recursively starting at the root node of the floor plan. Generally the membrane at the top cell is specified by a height, H, and a width, W. We will now show how a cell with membrane H and W is partitioned to the two lower cells a and b. The value H_a, H_b, W_a, and W_b is specified by

$$k_a = \begin{cases} \max\left(\frac{A_{iba}}{V}, H_a\right) & \text{if horizontal composition;} \\ \max\left(\frac{A_{iba}}{H}, V_a\right) & \text{if vertical composition} \end{cases}$$
$$k_b = \begin{cases} \max\left(\frac{A_{ibb}}{V}, H_b\right) & \text{if horizontal composition;} \\ \max\left(\frac{A_{ibb}}{H}, V_b\right) & \text{if vertical composition.} \end{cases}$$
$$H_a = \begin{cases} H * \left(\frac{k_a}{(k_a + k_b)}\right) & \text{if horizontal composition;} \\ H & \text{if vertical composition.} \end{cases}$$
$$H_b = \begin{cases} H * \left(\frac{k_b}{(k_a + k_b)}\right) & \text{if horizontal composition;} \\ H & \text{if vertical composition.} \end{cases} \tag{8.2}$$
$$W_a = \begin{cases} W & \text{if horizontal composition;} \\ W * \left(\frac{k_a}{(k_a + k_b)}\right) & \text{if vertical composition.} \end{cases}$$
$$W_b = \begin{cases} W & \text{if horizontal composition;} \\ W * \left(\frac{k_a}{(k_a + k_b)}\right) & \text{if vertical composition.} \end{cases}$$

This process of assigning the membrane to lower cells is repeated until the leaf cell is reached. We have now defined the membrane for all cells in our floor plan.

8.4.2 Guided Cell Compaction

There are two methods to compact the cells in our design. The first is the straight forward method, where each cell is compacted and then constricted in size. This compaction can be accomplished on one or more machines to complete the compaction of the cells. This method has been used and will produce quite good results. However the ports are not aligned during compaction.

The second method requires a large multiprocessor computer such as the Cosmic Cube[Seitz 85a]. This system consists of many processors connected as a Boolean N-cube. Each cell will be one separate process running in a node communicating to its neighbors about its size and port position. This allows abutting cells to agree on final port position and shape. Cells abutting on the left, right, top and bottom will be in adjacent nodes to limit the communication to one processor away. Each port in a cell is connected to its corresponding port by a cost function. The messages between the processes contains port position and when a port is going to be moved. This allows the final port position of a cell to be governed by the floor plan.

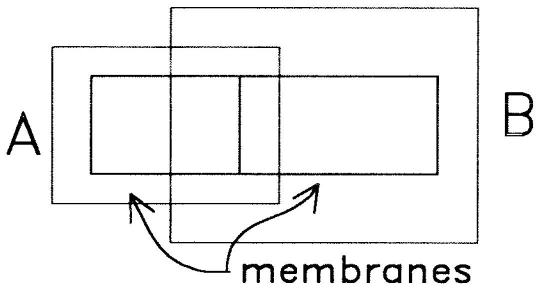


Figure 8.7 Cell Compaction Translation

A translation is used to correlate the relative position of ports between communicating cells. We create a translation rectangle for each cell which is initially placed at the membrane for the cells. The translation for a horizontal composition of two cells is shown in Figure 8.7. The ports are translated in such a fashion to make the translation rectangle juxtaposition as shown. This translation is performed starting at the leaf cells, proceeding up the floor plan tree, to each node within the tree.

During the communicating compaction process the translation rectangle is slowly increased to the minimum rectangle of each cell. This modification is accomplished by cooperative increase with the composition neighbor cell. That is to say, for two cells that are joined horizontally the height H of each cell is equal.

When the compaction process is completed, the cells are restricted to their translation rectangle for a perfect fit with the composition neighbor. It may be possible that some ports do not exactly line up. This problem will be taken care of in the next section.

This method can be extended by communicating edge information between joined cells to further enhance the method.

8.4.3 Cell Composition

After the cells have been compacted they are joined as specified by the floor plan using the binary composition operator. This composition will produce the VLSI chip.

8.4.4 Replicated Structures

A problem with the compaction method is that if we have a cell that is repeated as in an array each component will be compacted separately. We will now shown how they are combined to appear as a single cell. Consider the cell A that is in an array of three cells as shown in Figure 8.8 for horizontal composition. Recall from Section 8.3 that we can lock two ports together so they are processed as a single entity. For a horizontal array of the same cell we will lock the ports on the left side with the ports on the right side after aligning them. This port locking will allow the cell to be composed horizontally with itself without any port adjustment. Further, after the cell is compacted the adjustment of the edges as described in Section 8.2 is done with itself. Therefore we will have perfect mating for an array of n of these cells.

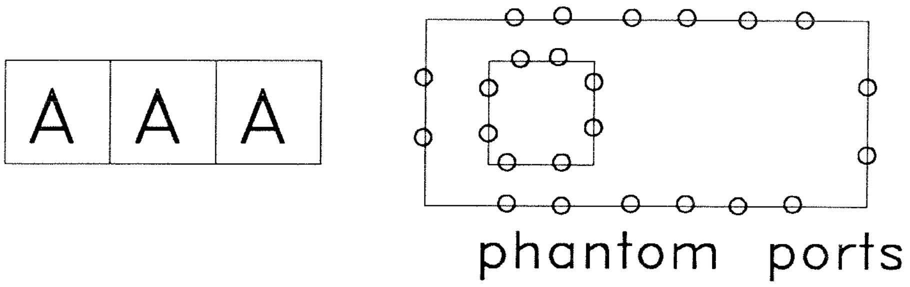


Figure 8.8 Cell Array

When this array cell is composed with other cells the ports are still locked together so the cells making up the array still compose. We do not allow the edge of the array to be expanded but add an interface wiring cell for this purpose. This does not destroy the compactness of the array.

This method can also be used for a matrix. In this case we would lock the left ports to the right, and the top to the bottom ports. Everything else is the same.

In the case of the cell processes for compaction as previously described, we would still only compact one cell for the array. We would use the locking concept for the left and right port and use phantom ports for the others as shown in Figure 8.8.

8.5 Ultra Compaction

When two cells are composed using the binary operator there is wasted area along the joining seam. The reason for this is that the extent of the cell sides is limited by the further out primitive. This means there is a small amount of wasted area along the sides. When two cells are joined this wasted area is the sum of the two cells' wasted area. Ultra compaction is the removal of this wasted area.

After the cells are joined, a new cell is constructed from the two. This cell is then annealed without increasing the size of the cell. This compaction further reduces the composed cells. The process of composing and compacting starts at the bottom of the floor plan tree and works up to the top node. This process creates an extremely compact design at increased computational expense.

9
Conclusion and Continuing Research

It has been shown by several examples that cells compacted by simulated annealing in 2 dimensions simultaneously can produce cells compacted at least as well as an experienced designer can do by hand. A unique representation to minimize the run time of simulated annealing has been developed. This representation has the added benefit of minimizing wire length. Using this representation a simulated annealing program has been written with complexity of $O(N * \log(A))f(d)$ where $f(d)$ is a slowly growing function of density, and A is the initialized cell area. Experiments indicate that $f(d)$ has very little or no dependence on N.

Development of integrated circuit design and layout tools has a software parallel. In the early days of software, executable machine code was generated by hand using an assembler. As software developed with high level languages, compilers were invented to translate the languages to machine code. In integrated circuit compaction, abstract representations are used to represent the circuit topology[Williams 77] where compactors are used to translate this form to near–minimal geometry. The number of compiler generated assembly language instructions was large as compared to hand assembled code. In automated integrated circuit compaction, the area was large as compared to a hand design. Early compilers were very crude and unable to generate code comparable to hand assembled code. This was the state of integrated circuit compactors.

Modern compilers[Robertson 81] generate code far superior to hand designed code. Code generated by these compilers is extremely compact and efficient. Hand generated code is easily read while such compiler generated is are not. Akin to the modern compilers, the integrated circuit compactor we have presented in this thesis produces extremely compact layouts, although they are not as easily read as hand compacted designs.

As anticipated by Carver Mead[Mead 83], the compaction technique developed has the advantageous characteristic of providing design portability with a technology when the ground rules for that technology are modified. Previous design techniques have either had to use inefficient constraints to allow for scaling or require complete redesign when technology changes.

9.1 Extensions

9.1.1 Restriction to Orthogonal Geometry

Some of the current software design tools and fabrication houses are limited to orthogonal geometry. Although the curvilinear framework was designed for all angle compaction, it could be restricted to orthogonal geometry. This is accomplished by using square boxes for the bubbles and right angles segments for the arounds. Wires would still be composed of segments and arounds, however the segments would be limited to being parallel to the x or y axis. The bubble move algorithm would be about the same except the processing of a primitive would be much simpler. We are planning to implement this version some time in the future.

9.1.2 Probabilistic Geometric Design Rules

Present day geometric design rules have been extracted from empirical estimation of fabrication requirements. Although these design rules are presented as hard and fast separation requirements, in fact they are based on statistical estimates of the probability of fabrication defects. If these probabilities were to be converted into a function of separation distance between objects, they could then be incorporated into the annealing cost function. Better fabrication yield could also be achieved by this technique.

9.2 Future Research

We have limited the compaction problem to a fixed topology. It can be seen from the cell examples in Chapter 7 that small topological changes could make substantial improvement in compaction. The area of automatic topological changes is open for further research.

In Chapter 7 we have shown by experiments that the compaction run time and compacted size are very susceptible to the membrane position and membrane cost. Although we have a method to position the membrane and assign the cost, it is clear that this method may not be the optimum. The question of what would be the optimum parameters and position for the membrane is for future research.

We have identified two possible methods to improve the computer run time for this algorithm in Chapter 7. Much more work can be done in this area.

Appendix A
Technology File

A.1 Technology File Description

We will define the syntax for the technology file in Backus–Naur form commonly known as **BNF**. The notation was defined in the 1950's by John Backus and Peter Nuar. The technology file is composed of a color definition, a set of rules and models, and the default parameters.

<Technology file>→<colors><rules and models><default>
<rules and models>→<rule> | <rule><rules and models> |
<model> | <model><rules and models> |

The color definition defines the colors for our design. The color attributes are used both in the color definition and in the rules definition. The symbol "*" on a color attribute denotes this attribute may only be used in the color definition while the symbol "†" denotes an attribute that may be used in a rule definition. All others may be used in either place. The attribute "physical" denotes a color that is to be displayed in plotting and graphic editing. The attribute "rule" denotes a color that is a rule color. Rule colors must have the "mindistance" attribute defined. The "width" defines the width of the rule. The "connect" attribute allows connection if the two primitives that are to be connected have this color. The "relsrule" defines this color as related significant. The "relswith" defines the related width. If it is not defined the width will be used. We define the syntax for the color definition as follows.

<Colors>→ colors(<color entries>)
<Color entries>→<color entry> | <color entry>, <color entries>
<colorentry>→<color name> | <color name> [<color attributes>]
<color attributes>→<color attribute> |
                    <color attributes>, <color attributes>
<color attribute>→ physical | rule| connect$_\dagger$ |
                    mindistance$_*$ =<real> | width=<real> |
                    relsrule$_*$ |relsrule[<integer]$_\dagger$|
                    relswidth[<integer>]$_\dagger$ =<real>

The rule definition defines a rule that may be used by a primitive. The rule has a name and a body composed of rule parts. The rule part color definition describes the colors and associated color attributes for the rule. The rule part edit describes how the rule is to be used in the graphic editor. An edit type of port allows a port to be created with this rule. The edit type of bubble allows this rule to be used as a wiring bubble while an edit type of segment allows the rule to be used for a wire. The flag type of showfill will display the rule fleshed out in the graphic editor. The cost is the cost used to control the compaction process. We define the rules syntax as follows.

<rule>→rule <rule name> (<rule parts>)
<rule parts>→<rule part> | <rule part>, <rule parts>
<rule part>→ alias=<alias name> |
              color= (<color entries>)|
              edit= (<edit types>)|
              flag= (<flag types>)|
              cost=<real>
<edit types>→<edit type> | <edit type>, <edit types>
<edit type>→ port|bubble|segment
<flag types>→<flag type> | <flag type>, <flag types>
<flag type>→ showfill

The model definition describes a model that can be used in a design. The model has a model name and a model body. The body is composed of a model related part and the primitives of the model. The model related part defines all the related equivalence numbers and their corresponding colors and side that will be used by the model. The model part defines the bubbles and connecting segments that makeup the model. The bubble entry defines an mbubble which has a name, a rule, a location, and related part. The segment entry defines an msegment which has a rule, a connect part, and a related part. The connect part defines the two mbubbles for the msegment by their corresponding names. The order of presentation of the msegment connect bubbles define the order of the fields in model structure which defines the relationship for the related side. The related part defines the equivalence related numbers and their associated related index.

When the model is instanced by the graphic editor the graphic editor creates the instance structure and copies the model's coordinates. The graphic editor then translates the instance to the proper position. The zero point in the model coordinate system is translated to the graphic editors place point in the design. The location of the bubbles define the position of the bubbles in the model coordinate space.

```
<model>→ model <model name> (<model related parts>
                        <model parts>)
<model related parts>→<model related part> |
                        <model related part><model related parts>
<model related part>→related(<model related entries>)
<model related entries>→<model related entry> |
                        <model related entry><model related entries>
<model related entry>→<integer> |
                        <integer>,color= (<color name list>)|
                        <integer>,color= (<color name list>),
                        side=<side name>
<side name>→ leftside|rightside
<color name list>→<color name> |
                        <color name>,<color name list>
<model parts>→<model part> | <model part>,<model parts>
<model part>→<mbubble entry> | <msegment entry>
<mbubble entry>→bubble <bubble name> (<bubble parts>)
<mbubble parts>→<mbubble part> | <mbubble part>,<mbubble parts>
<mbubble part>→rule=<rule name> |
                        location= (<x real>,<y real>)|
                        <related part>
<related part>→related= (<related entries>)
<related entries>→<related entry> | <related entry>,<related entries>
<related entry>→<integer> |
                        <integer>rels=<integer>
<msegment entry>→segment (<msegment parts>)
<msegment parts>→<msegment part> | <msegment part>,
                        <msegment parts>
<msegment part>→rule=<rule name> |
                        connect= (<nbubble name>,<nbubble name>)|
                        <related part>
```

The default defines the grid for the bubbles, the initial bounding box for a new cell, shown grid spacing, and the snap to grid value.

```
<default>→default (<default parts>)
<default parts>→<default part> | <default part>,<default parts>
<default part>→ grid=<real> |
                bbox= (<real>,<real>,<real>,<real>,<real>)|
                showngrid=<real> |
snapgrid=<real>
```

A.2 nMOS Example

```
colors(
        blue [PHYSICAL,Rule, width=300,mind=300,cif=nm],
        green [PHYSICAL,RULE, width=200,MINDISTANCE=300,cif=nd ],
        red [PHYSICAL,RULE ,width=200,MINDISTANCE=200,cif=np ],
        blah [ RULE ,width=200,MINDISTANCE=100 ],
        gate [ RULE ,width=200,MINDISTANCE=200 ],
        black [PHYSICAL,WIDTH=200 cif = nc],
        yellow [PHYSICAL,cif=ni ],
        brown [PHYSICAL,CIF=nb ],
)
RULE red(alias=polysilicon,
    COLOR=(
            red[PHYSICAL,RULE,CONNECT],
            blah[RULE]
    ),
    edit=(bubble,port,segment),
    COST=2)
)
RULE green(alias=diffusion,
    COLOR=(
            green[PHYSICAL,RULE,CONNECT],
            blah[RULE]
    ),
    edit=(bubble,port,segment),
    COST=4
)
RULE blue(alias=metal,
    COLOR=(
            blue[PHYSICAL,RULE,CONNECT]
    )
    edit=(port,bubble,segment),
    COST=1
)
RULE "blue-4"(alias=metal,
    COLOR=(
            blue[PHYSICAL,RULE,CONNECT, width=400]
    )
    edit=(port,bubble,segment),
    COST=1
)
RULE "blue-6"(alias=metal,
    COLOR=(
            blue[PHYSICAL,RULE,CONNECT,width=600]
    )
    edit=(port,bubble,segment),
    COST=1
)
RULE "blue-8"(alias=metal,
    COLOR=(
            blue[PHYSICAL,RULE,CONNECT,width=800]
    )
    edit=(port,bubble,segment),
    COST=1
)

RULE CONBLUGRE(
    COLOR=(
            green[PHYSICAL,connect,RULE, WIDTH=400 ],
            blue[PHYSICAL,connect, RULE, WIDTH=400 ],
            black[PHYSICAL],
            blah[RULE, WIDTH=400 ]
            gate [RULE,relsrule],
    ),
    COST=1)

RULE CONBLURED(
    COLOR=(
            red[PHYSICAL,connect, RULE, WIDTH=400 ],
            blue[PHYSICAL,connect, RULE, WIDTH=400 ],
            black[PHYSICAL],
            blah[RULE, WIDTH=400 ]
            gate [RULE,relsrule],
    ),
    COST=1
)
model ContactMD(
    related(1,color=(green),2,color=(blue))
    bubble ar1(rule=CONBLUGRE,location=(0,0),
    related=(1,2)),
)
model ContactMP(
    related(1,color=(red),2,color=(blue))
    bubble ar1(rule=CONBLURED,location=(0,0),
    related=(1,2)),
)

rule bur(
    color=(
            red[physical,rule],
            green[physical,rule],
                brown[physical,width=400],
                blah[rule,width=400,relsrule[0],relswidth[0]=100,
                relsrule[1],relswidth[1]=100],
                gate [RULE,relsrule[0],relsrule[1],relsmrule,relsmwidth=0],
        ),
        cost=1
)

rule "bur-gns"(
        color=(
                green[physical,rule],
                blah[rule,relsrule[0],relswidth[0]=100]
        ),
        flag=(showfill),
        cost=.2)
)

rule "bur-rds"(
        color=(
                red[physical,rule],
                blah[rule,relsrule[1],relswidth[1]=100]
        ),
        flag=(showfill),
        cost=.2
)
model buried(
        related(1,color=(red),2,color=(green))
        bubble a(rule=bur,location=(0,0),related=(1 rels=0,2 rels=1))
        bubble al(rule=red,location=(-200,0),related=(1))
        segment(rule="bur-rds",connect=(a,al),related=(1,2 rels=1))
        bubble ar(rule=green,location=(+200,0),related=(2))
        segment(rule="bur-gns",connect=(a,ar),related=(1 rels=0 ,2 rels=1)),
)
model "buried-redY"(
        related(1,color=(red),2,color=(green))
        bubble a(rule=bur,location=(0,0),related=(1 rels=0,2 rels=1))
        bubble at(rule=red,location=(0,200),related=(1))
        segment(rule="bur-rds",connect=(a,at),related=(1,2 rels=1))
        bubble al(rule=red,location=(-200,0),related=(1))
        segment(rule="bur-rds",connect=(a,al),related=(1,2 rels=1))
        bubble ar(rule=green,location=(+200,0),related=(2))
        segment(rule="bur-gns",connect=(a,ar),related=(1 rels=0 ,2 rels=1)),
)
model "buried-grnY"(
        related(1,color=(red),2,color=(green))
        bubble a(rule=bur,location=(0,0),related=(1 rels=0,2 rels=1))
```

```
     bubble al(rule=red,location=(-200,0),related=(1))
     segment(rule="bur-rds",connect=(a,al),related=(1,2 rels=1))
     bubble ar(rule=green,location=(+200,0),related=(2))
     segment(rule="bur-gns",connect=(a,ar),related=(1 rels=0 ,2 rels=1)),
     bubble ab(rule=green,location=(0,200),related=(2))
     segment(rule="bur-gns",connect=(a,ab),related=(1 rels=0 ,2 rels=1)),
)
model "buried-X"(
     related(1,color=(red),2,color=(green))
     bubble a(rule=bur,location=(0,0),related=(1 rels=0,2 rels=1))
     bubble al(rule=red,location=(-200,0),related=(1))
     segment(rule="bur-rds",connect=(a,al),related=(1,2 rels=1))
     bubble ar(rule=red,location=(+200,0),related=(1))
     segment(rule="bur-rds",connect=(a,ar),related=(1,2 rels=1))
     bubble at(rule=green,location=(0,200),related=(2))
     segment(rule="bur-gns",connect=(a,at),related=(1 rels=0 ,2 rels=1)),
     bubble ab(rule=green,location=(0,-200),related=(2))
     segment(rule="bur-gns",connect=(a,ab),related=(1 rels=0 ,2 rels=1)),
)

rule t11c(
     color=(
             red[physical,rule],
             green[physical,rule,relsrule[0]],
             blah[rule,
             relsrule[0],relswidth[0]=50,
             relsrule[1],relswidth[1]=50]
             gate [RULE,relsrule,relsrule[1]],
     ),
     cost=1
)
rule t11gs(
     color=(
             green[physical,rule],
             blah[rule,
             relsrule[0],relswidth[0]=50]
     ),
     flag=(showfill),
     cost=.2
)
rule t11rs(
     color=(
             red[physical,rule],
             blah[rule,
             relsrule[1],relswidth[1]=50]
     ),
    flag=(showfill),
    cost=.2
)
model "EH1-1"(
    related(1,color=(red)),
    related(2,color=(green)),
    related(3,color=(green)),
    bubble ac(rule=t11c,location=(0,0),related=(1,2 relsrule=1,3 relsrule=1))
    bubble al(rule=green,location=(-250,0),related=(2,3))
    segment(rule=t11gs,connect=(ac,al),related=(1 rels=0,2 rels=1,3 rels=1))
    bubble ar(rule=green,location=(+250,0),related=(3,2))
    segment(rule=t11gs,connect=(ac,ar),related=(1 rels=0,2 rels=1,3 rels=1))
    bubble ab(rule=red,location=(0,-250),related=(1))
    segment(rule=t11rs,connect=(ac,ab),related=(1 rels=0,2 rels=1,3
            rels = 1))
    bubble at(rule=red,location=(0,+250),related=(1))
    segment(rule=t11rs,connect=(ac,at),related=(1 rels=1,2 rels=1,3 rels=1))
)
rule t12c(
    color=(
            red[physical,rule]
            green[physical,rule,width=400,relsrule[0]]
            blah[rule,width=400,
            relsrule[0],relswidth[0]=200,
            relsrule[1],relswidth[1]=50]
            gate [rule,width=200,relsrule[0],relsrule[1]],
    ),
    cost=1
)

rule t12g(
    color=(
            green[physical,rule,width=400,connect]
            blah[rule,width=400]
    ),
    cost=.2
)
rule t12gs(
    color=(
            green[physical,rule,width=400]
            blah[rule,width=400,
            relsrule[0],relswidth[0]=200]
    ),
    flag=(showfill),
    cost=.2
)
```

```
model “EH1-2”(
     related(1,color=(red)),
     related(2,color=(green)),
     related(3,color=(green)),
     bubble ac(rule=t12c,location=(0,0),related=(1 rels=0,2 rels=1,3 rels=1))
     bubble al(rule=t12g,location=(-350,0),related=(2,3))
     segment(rule=t12gs,connect=(ac,al),related=(1 rels=0,2 rels=1,3 rels=1))
     bubble ar(rule=t12g,location=(+350,0),related=(3,2))
     segment(rule=t12gs,connect=(ac,ar),related=(1 rels=0,2 rels=1,3 rels=1))
     bubble ab(rule=red,location=(0,-300),related=(1))
     segment(rule=t11rs,connect=(ac,ab),related=(1 rels=0,2 rels=1,3 rels=1))
     bubble at(rule=red,location=(0,+300),related=(1))
     segment(rule=t11rs,connect=(ac,at),related=(1 rels=0,2 rels=1,3 rels=1))
)
rule t14c(
     color=(
             red[physical,rule]
             green[connect,physical,rule,width=600,relsrule[0]]
     blah[rule,width=600,
             relsrule[0],relswidth[0]=300,
             relsrule[1],relswidth[1]=200]
             gate [rule,width=200,relsrule[0],relsrule[1],relsmrule,relsmwidth=0],
     ),
     flag=(showfill),
     cost=.333
)
model “EH1-4”(
     related(1,color=(red)),
     related(2,color=(green),side=leftside),
     related(3,color=(green),side=rightside),
     related(4),
     bubble at(rule=red,location=(0,+400),related=(1))
     bubble ac1(rule=t14c,location=(0,0),related=(4,1 rels=0,2 rels=1,3 rels=1))
     segment(rule=t11rs,connect=(at,ac1),related=(1 rels=0))
     bubble ac2(rule=t14c,location=(0,-200),related=
     (4,1 rels=0,2 rels=1,3 rels=1))
     segment(rule=t14c,connect=(ac1,ac2),related=(4,1 rels=0,2 rels=1,3rels=1))
     bubble ab(rule=red,location=(0,-600),related=(1))
     segment(rule=t11rs,connect=(ac2,ab),related=(1 rels=0))
)
rule t110c(
     color=(
             red[physical,rule]
             green[connect,physical,rule,width=600,relsrule[0]]
             blah[rule,width=600,
             relsrule[0],relswidth[0]=300,
                relsrule[1],relswidth[1]=200]
                gate [rule,width=200,relsrule[0],relsrule[1],relsmrule,relsmwidth=200],
        ),
        flag=(showfill),
        cost=.2
)
model "EH1-5"(
        related(1,color=(red)),
        related(2,color=(green),side=leftside),
        related(3,color=(green),side=rightside),
        related(4),
        bubble at(rule=red,location=(0,+400),related=(1))
        bubble ac1(rule=t110c,location=(0,0),related=(4,1 rels=0,2 rels=1,3 rels=1))
        segment(rule=t11rs,connect=(at,ac1),related=(1 rels=0))
        bubble ac2(rule=t110c,location=(0,-400),related=
        (4,1 rels=0,2 rels=1,3 rels=1))
        segment(rule=t110c,connect=(ac1,ac2),related=(4,1 rels=0,2 rels=1,3rels=1))
        bubble ab(rule=red,location=(0,-800),related=(1))
        segment(rule=t11rs,connect=(ac2,ab),related=(1 rels=0))
)
model "EH1-10"(
        related(1,color=(red)),
        related(2,color=(green),side=leftside),
        related(3,color=(green),side=rightside),
        related(4),
        bubble at(rule=red,location=(0,+400),related=(1))
        bubble ac1(rule=t110c,location=(0,0),related=(4,1 rels=0,2 rels=1,3 rels=1))
        segment(rule=t11rs,connect=(at,ac1),related=(1 rels=0))
        bubble ac2(rule=t110c,location=(0,-400),related=
        (4,1 rels=0,2 rels=1,3 rels=1))
        segment(rule=t110c,connect=(ac1,ac2),related=(4,1 rels=0,2 rels=1,3rels=1))
        bubble ac3(rule=t110c,location=(0,-800),related=
        (4,1 rels=0,2 rels=1,3 rels=1))
        segment(rule=t110c,connect=(ac2,ac3),related=(4,1 rels=0,2 rels=1,3rels=1))
        bubble ac4(rule=t110c,location=(0,-1200),related=
        (4,1 rels=0,2 rels=1,3 rels=1))
        segment(rule=t110c,connect=(ac3,ac4),related=(4,1 rels=0,2 rels=1,3rels=1))
        bubble ab(rule=red,location=(0,-1600),related=(1))
        segment(rule=t11rs,connect=(ac4,ab),related=(1 rels=0))
)

rule d11c(
        color=(
                red[physical,rule],
                green[physical,rule,relsrule[0]],
                blah[rule,
            relsrule[0],relswidth[0]=50,
            relsrule[1],relswidth[1]=50]
            gate [RULE,relsrule,relsrule[1]],
            yellow[physical,width=500]
        ),
        cost=1
)
rule d11gs(
        color=(
            green[physical,rule],
            blah[rule,
            relsrule[0],relswidth[0]=50]
        ),
        flag=(showfill),
        cost=.2
)

model "DP1-1"(
        related(1,color=(red)),
        related(2,color=(green)),
        related(3,color=(green)),
        bubble ac(rule=d11c,location=(0,0),related=(1,2 relsrule=1,3 relsrule=1))
        bubble al(rule=green,location=(-250,0),related=(2,3))
        segment(rule=d11gs,connect=(ac,al),related=(1 rels=0,2 rels=1,3 rels=1))
        bubble ar(rule=green,location=(+250,0),related=(3,2))
        segment(rule=d11gs,connect=(ac,ar),related=(1 rels=0,2 rels=1,3 rels=1))
        bubble ab(rule=red,location=(0,-250),related=(1))
        segment(rule=t11rs,connect=(ac,ab),related=(1 rels=0,2 rels=1,3
            rels = 1))
        bubble at(rule=red,location=(0,+250),related=(1))
        segment(rule=t11rs,connect=(ac,at),related=(1 rels=1,2 rels=1,3 rels=1))
)
rule d21c(
        color=(
            red[physical,rule,width=400]
            green[physical,rule,relsrule[0]]
            blah[rule,width=400,
            relsrule[1],relswidth[1]=200,
            relsrule[0],relswidth[0]=50]
            gate [rule,relsrule[0],relsrule[1]],
            yellow[physical,width=500],
        ),
        cost=1
)
rule d21gs(
        color=(
            green[physical,rule],
            blah[rule,
            relsrule[0],relswidth[0]=50]
            yellow[physical,width=500]
    ),
    flag=(showfill),
    cost=.2
)
rule d21r(
    color=(
            red[physical,rule,width=400,connect]
            blah[rule,width=400]
    ),
    cost=.2)
)
rule d21rs(
    color=(
            red[physical,rule,width=400]
            blah[rule,width=400,
            relsrule[1],relswidth[1]=200]
    ),
    flag=(showfill),
    cost=.2
)
model "DP2-1"(
    related(1,color=(red)),
    related(2,color=(green)),
    related(3,color=(green)),
    bubble ac(rule=d21c,location=(0,0),related=(1 rels=0,2 rels=1,3 rels=1))
    bubble al(rule=green,location=(-300,0),related=(2,3))
    segment(rule=d21gs,connect=(ac,al),related=(1 rels=0,2 rels=1,3 rels=1))
    bubble ar(rule=green,location=(+300,0),related=(3,2))
    segment(rule=d21gs,connect=(ac,ar),related=(1 rels=0,2 rels=1,3 rels=1))
    bubble ab(rule=d21r,location=(0,-350),related=(1))
    segment(rule=d21rs,connect=(ac,ab),related=(1 rels=0,2 rels=1,3 rels=1))
    bubble at(rule=d21r,location=(0,+350),related=(1))
    segment(rule=d21rs,connect=(ac,at),related=(1 rels=0,2 rels=1,3 rels=1))
)
rule D14c(
    color=(
            red[connect,physical,rule,width=600,relsrule[1]]
            green[physical,rule]
            blah[rule,width=600,
            relsrule[1],relswidth[1]=400,
            relsrule[0],relswidth[0]=200]
            gate [rule,width=200,relsrule[0],relsrule[1],relsmrule,relsmwidth=0],
                yellow[physical,width=500]
        ),
        flag=(showfill),
        cost=.2
)
model "PULL4-1"(
        related(1,color=(red)),
        related(2,color=(green)),
        related(3,color=(green)),
        related(4),
        bubble at(rule=green,location=(-400,0),related=(2))
        bubble ac1(rule=d14c,location=(0,0),related=(4,1 rels=0,2 rels=1,3 rels=1))
        segment(rule=d21gs,connect=(at,ac1),related=(2 rels=0))
        bubble ac2(rule=d14c,location=(200,0),related=
        (4,1 rels=0,2 rels=1,3 rels=1))
        segment(rule=d14c,connect=(ac1,ac2),related=(4,1 rels=0,2 rels=1,3rels=1))
        bubble ab(rule=green,location=(600,0),related=(3))
        segment(rule=d21gs,connect=(ac2,ab),related=(3 rels=0))
)
rule D101c(
        color=(
                red[connect,physical,rule,width=600,relsrule[1]]
                green[physical,rule]
                blah[rule,width=600,
                relsrule[1],relswidth[1]=400,
                relsrule[0],relswidth[0]=200]
                gate [rule,width=200,relsrule[0],relsrule[1],relsmrule,relsmwidth=200],
                yellow[physical,width=500]
        ),
        flag=(showfill),
        cost=.2
)
model "DP5-1"(
        related(1,color=(red)),
        related(2,color=(green)),
        related(3,color=(green)),
        related(4),
        bubble at(rule=green,location=(-400,0),related=(2))
        bubble ac1(rule=d101c,location=(0,0),
        related=(4,1 rels=0,2 rels=1,3 rels=1))
        segment(rule=d21gs,connect=(at,ac1),related=(2 rels=0))
        bubble ac2(rule=d101c,location=(400,0),related=
        (4,1 rels=0,2 rels=1,3 rels=1))
        segment(rule=d101c,connect=(ac1,ac2),related=(4,1 rels=0,2 rels=1,3rels=1))
        bubble ab(rule=green,location=(800,0),related=(3))
        segment(rule=d21gs,connect=(ac2,ab),related=(3 rels=0))
```

```
)
model "DP10-1"(
     related(1,color=(red)),
     related(2,color=(green)),
     related(3,color=(green)),
     related(4),
     bubble at(rule=green,location=(-400,0),related=(2))
     bubble ac1(rule=d101c,location=(0,0),
             related=(4,1 rels=0,2 rels=1,3 rels=1))
     segment(rule=d21gs,connect=(at,ac1),related=(2 rels=0))
     bubble ac2(rule=d101c,location=(400,0),related=
     (4,1 rels=0,2 rels=1,3 rels=1))
     segment(rule=d101c,connect=(ac1,ac2),related=(4,1 rels=0,2 rels=1,3rels=1))
     bubble ac3(rule=d101c,location=(800,0),related=
             (4,1 rels=0,2 rels=1,3 rels=1))
     segment(rule=d101c,connect=(ac2,ac3),related=(4,1 rels=0,2 rels=1,3rels=1))
     bubble ab(rule=green,location=(1200,0),related=(3))
     segment(rule=d21gs,connect=(ac3,ab),related=(3 rels=0))
)

rule p41(
     color=(
             red[connect,physical,rule,width=200,relsrule[1]]
             green[physical,rule]
             blah[rule,width=200,
             relsrule[1],relswidth[1]=200,
             relsrule[0],relswidth[0]=200]
             gate [rule,width=200,relsrule[0],relsrule[1],relsmrule,relsmwidth=200],
     ),
     flag=(showfill),
     cost=.2
)

model "pull4-1"(
     related(1,color=(red)),
     related(2,color=(green)),
     related(3,color=(green)),
     related(4),
     bubble al(rule=green,location=(-200,0),related=(2))
     bubble ac1(rule=bur,location=(0,0), related=(4,1 rels=0,2 rels=1))
     segment(rule="bur-gns",connect=(al,ac1),related=(4,1 rels=0,2 rels=1))
     bubble ac2(rule=d101c,location=(300,0),
     related=(4,1 rels=0,2 rels=1,3 rels=1))
     segment(rule=p41,connect=(ac1,ac2),related=(4,1 rels=0,2 rels=1))
     bubble ac3(rule=d101c,location=(700,0),related=
     (4,1 rels=0,2 rels=1,3 rels=1))
```

```
            segment(rule=d101c,connect=(ac2,ac3),related=(4,1 rels=0,2 rels=1,3rels=1))
        bubble ab(rule=green,location=(1100,0),related=(3))
        segment(rule=d21gs,connect=(ac3,ab),related=(3 rels=0))
)
model "pull6-1"(
        related(1,color=(red)),
        related(2,color=(green)),
        related(3,color=(green)),
        related(4),
        bubble al(rule=green,location=(-200,0),related=(2))
        bubble ac1(rule=bur,location=(0,0), related=(4,1 rels=0,2 rels=1))
        segment(rule="bur-gns",connect=(al,ac1),related=(4,1 rels=0,2 rels=1))
        bubble ac2(rule=d101c,location=(300,0),
                related=(4,1 rels=0,2 rels=1,3 rels=1))
        segment(rule=p41,connect=(ac1,ac2),related=(4,1 rels=0,2 rels=1))
        bubble ac3(rule=d101c,location=(700,0),related=
                (4,1 rels=0,2 rels=1,3 rels=1))
        segment(rule=d101c,connect=(ac2,ac3),related=(4,1 rels=0,2 rels=1,3rels=1))
        bubble ac4(rule=d101c,location=(1100,0),related=
                (4,1 rels=0,2 rels=1,3 rels=1))
        segment(rule=d101c,connect=(ac3,ac4),related=(4,1 rels=0,2 rels=1,3rels=1))
        bubble ab(rule=green,location=(1500,0),related=(3))
        segment(rule=d21gs,connect=(ac4,ab),related=(3 rels=0))
)
        default(grid=25,bbox=(4000,4000),showngrid=100,snapgrid=100)
```

Appendix B
Miscellaneous Utility Functions

B.1 Inside Around Algorithm

The function "insidearound"(A, x, y) return a logical true if the point defined by x, y is inside the around; otherwise it returns a logical false. In Figure B.0 the point P_1 is inside the around A while the point P_2 is outside the around A. One of the uses of insidearound is to determine if an around is pushed out by bubble B or the around is a block similar to a bubble.

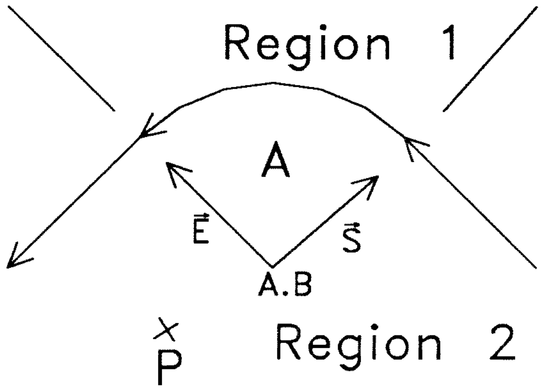


Figure B.1 Inside Around

We will assume the around is a counter clockwise around where the radius is positive. For the case where the around is clockwise we would interchange the definition vectors. The function inside around is divided into three parts. The first case is if the distance from the point $P = (x, y)$ to the center of the around, $(A.B.x, A.B.y)$ is less than or equal to the radius then the point is within the around. The test is

$$\sqrt{(P.x - A.B.x)^2 + (P.y - A.B.y)^2} \leq |A.r|.$$

The second case is to test if the point P is in region 1 as shown in Figure B.1. If the point P is in region 1 then the point P is outside of the around. We let the vector

$$\overrightarrow{Q} = (P.x - A.B.x, P.y - A.B.y) \qquad (B.1)$$

be directed from the center of the around to the point. Point P is in region 1 if vector $\overrightarrow{Q}$ is in the left half plane of vector $\overrightarrow{S}$ and in the right half plane of vector $\overrightarrow{E}$ for the around of less than or equal to 180°. For the vector $\overrightarrow{Q}$ to be in the left half plane of vector $\overrightarrow{S}$ the imaginary part of the product of vector $\overrightarrow{Q}$ and the conjugate of vector $\overrightarrow{S}$ rotated by 90° which is magnitude of vector $\overrightarrow{Q}$ times the sine of the angle between $\overrightarrow{Q}$ and $\overrightarrow{S}$ rotated by 90° would be greater than or equal to zero. In a similar manner vector $\overrightarrow{Q}$ can be tested for presents in the right half plan of vector $\overrightarrow{E}$. For an around greater than 180° we would logically "or" the test. The point P is in region 1 if the following is true.

$$\begin{aligned}
&(\mathrm{IMG}(\overrightarrow{E} * \overline{\overrightarrow{S}}) \geq 0 \wedge \\
&\mathrm{IMG}(\overrightarrow{Q} * \overline{(\overrightarrow{S} * (1,0))}) > 0 \ \wedge \mathrm{IMG}((\overrightarrow{E} * (-1,0)) * \overline{\overrightarrow{Q}}) > 0) \\
&\vee \\
&(\mathrm{IMG}(\overrightarrow{E} * \overline{\overrightarrow{S}}) < 0 \wedge \\
&\mathrm{IMG}(\overrightarrow{Q} * \overline{(\overrightarrow{S} * (1,0))}) > 0 \ \vee \mathrm{IMG}((\overrightarrow{E} * (-1,0)) * \overline{\overrightarrow{Q}}) > 0)
\end{aligned}$$

Either the previous cases determined whether point P was inside the around or the point is in region 2. We now need to know if the point is inside the bounding segments attached to the around. We know the vectors $\overrightarrow{S}$ and $\overrightarrow{E}$ are unit vectors. We take the imaginary part of the product of conjugate vector $\overrightarrow{S}$ rotated −90° which is

$$|\overrightarrow{Q}|\sin(\theta)$$

where θ is the angle between $\overrightarrow{S}$ rotated by −90° and $\overrightarrow{Q}$. If the value $|\overrightarrow{Q}|\sin(\theta)$ is less than or equal to the radius of the around the point is on the inside of the start segment.

$$\begin{aligned}
&(\mathrm{IMG}(\overrightarrow{E} * \overline{\overrightarrow{S}}) \geq 0 \wedge \\
&\mathrm{IMG}((\overrightarrow{S} * (-1,0)) * \overline{\overrightarrow{Q}}) \leq |A.r| \ \wedge \mathrm{IMG}(\overrightarrow{Q} * \overline{(\overrightarrow{E} * (1,0))}) \leq |A.r|) \\
&\vee \\
&(\mathrm{IMG}(\overrightarrow{E} * \overline{\overrightarrow{S}}) < 0 \wedge \\
&\mathrm{IMG}((\overrightarrow{S} * (-1,0)) * \overline{\overrightarrow{Q}}) \leq |A.r| \ \vee \mathrm{IMG}(\overrightarrow{Q} * \overline{(\overrightarrow{E} * (1,0))}) \leq |A.r|)
\end{aligned}$$

```
procedure insidearound(A,x,y)
begin
    real  $as, bs, ae, be$;
    if $A.r > 0$then
    begin
        $as \leftarrow A.as$; $bs \leftarrow A.bs$;
        $ae \leftarrow A.ae$; $be \leftarrow A.be$;
    end else
    begin
        $ae \leftarrow A.as$; $be \leftarrow A.bs$;
        $as \leftarrow A.ae$; $bs \leftarrow A.be$;
    end;
    $x \leftarrow x - A.B.x$; $y \leftarrow y - A.B.y$;
    return(
    return(
    $\sqrt{x^2 + y^2} \leq |sA.r| \vee$
    $(((-ae * bs + be * as \geq 0 \wedge$
    $(-x * bs + y * as \geq 0 \wedge x * be - y * ae \geq 0 \wedge)$
    $\vee/((-ae * bs + be * as < 0 \wedge$
    $(-x * bs + y * as \geq 0 \ \vee \ x * be - y * ae \geq 0 \wedge))$
    $\wedge((ae * bs - be * as \geq 0 \wedge$
    $(x * bs - y * as \geq |A.r| \wedge -x * be + y * ae \geq |A.r| \wedge)$
    $\vee/((ae * bs - be * as < 0 \wedge$
    $(x * bs - y * as \geq |A.r| \ \vee \ -x * be + y * ae \geq |A.r| \wedge)))$
    )
end;
```

B.2 Between a Around Segments Algorithm

The boolean function "betweenaseg"(A, x, y) returns true if the vector defined by the pair (x, y) is between the vector parallel to and pointing outward from the end segment of around A and the vector parallel to and pointing outward from the start segment of A for a counter clockwise around as shown in Figure B.2. For a clockwise around the start and end vectors are interchanged.

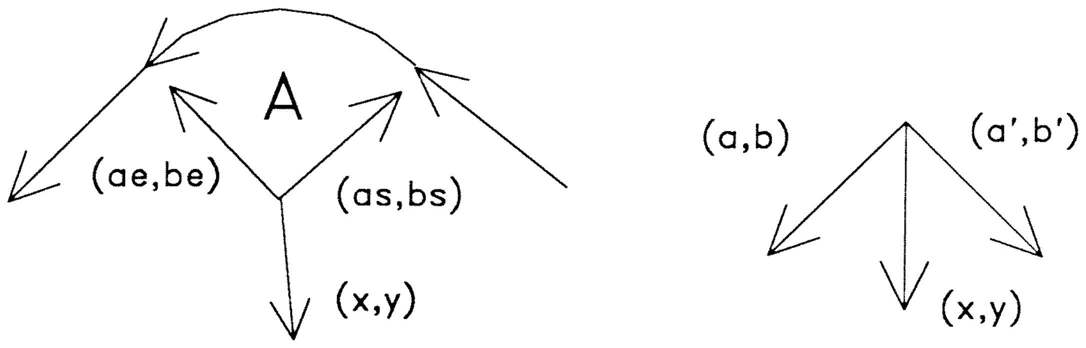


Figure B.2 Attached Around Test

We define the function betweenaseg for around A and vector (x, y) as the following. If the around A is counter clockwise we set

$$\begin{aligned} as &= A.as, & bs &= A.bs, \\ ae &= A.ae, & be &= A.be, \\ a &= -A.be, & b &= A.ae, \\ a' &= A.bs, \text{ and } & b' &= -A.as. \end{aligned} \tag{B.2}$$

Otherwise for a clockwise around A we set

$$\begin{aligned} as &= A.ae, & bs &= A.be, \\ ae &= A.as, & be &= A.bs, \\ a &= A.bs, & b &= -A.as, \\ a' &= -A.be, \text{ and } & b' &= A.ae. \end{aligned} \tag{B.3}$$

If the around A is greater than 180° then the function must be logically false since the vector (x, y) can not be between the segments. We can test if the vector (x, y) is between the vector (a, b) and the vector (a', b') by testing if vector (x, y) is in the left half plane of vector (a, b) and the right half plane of vector (a', b') for an angle between (a, b) and (a', b') of less than or equal to 180°. For an angle greater than 180° the test would be logically or. We define the function betweenaseg as:

$$\begin{aligned} & -ae * bs + be * as < 0.0 \wedge \\ & ((a' * b + b' * a \geq 0 \wedge \\ & x * b + y * a \geq 0 \ \wedge \ a' * y + b' * x \geq 0) \\ & \vee \\ & (a' * b + b' * a < 0 \wedge \\ & x * b + y * a \geq 0 \ \vee \ a' * y + b' * x \geq 0)) \end{aligned} \tag{B.4}$$

Appendix C

Intermediate Structures

This appendix contains the figures from the timing experiments as described in Chapter 7.

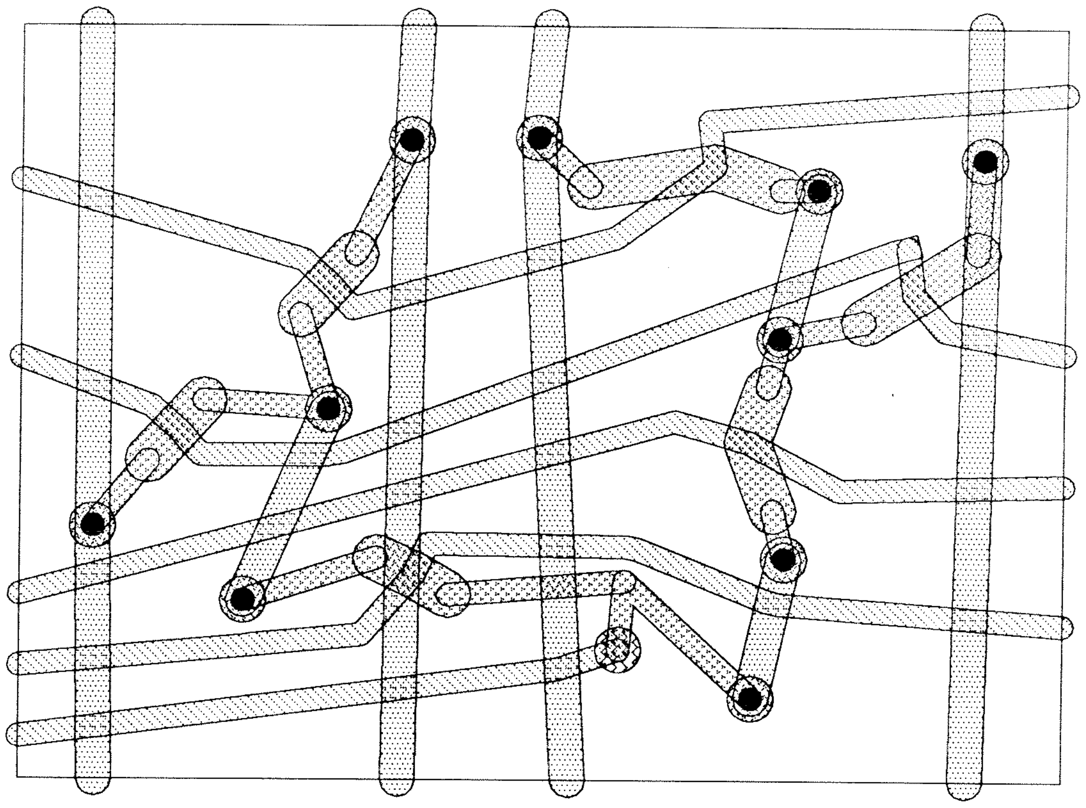


Figure C.1 1 Cell - Initial

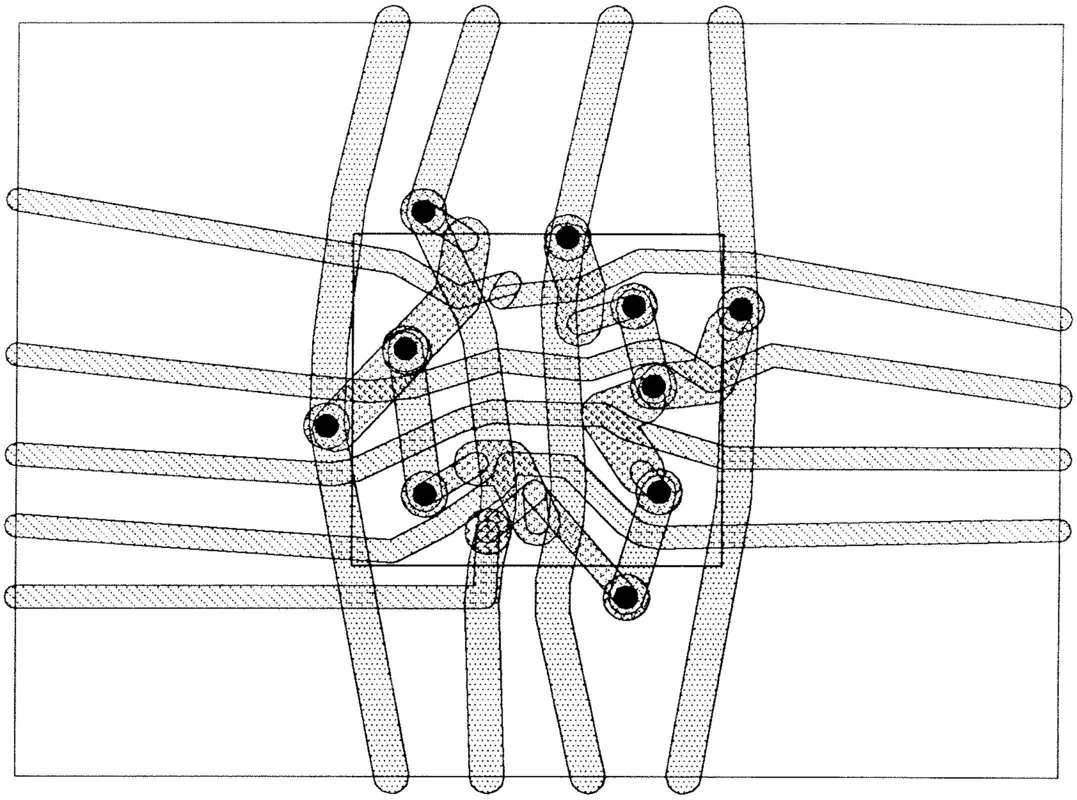


Figure C.2 1 Cell - end of first stage

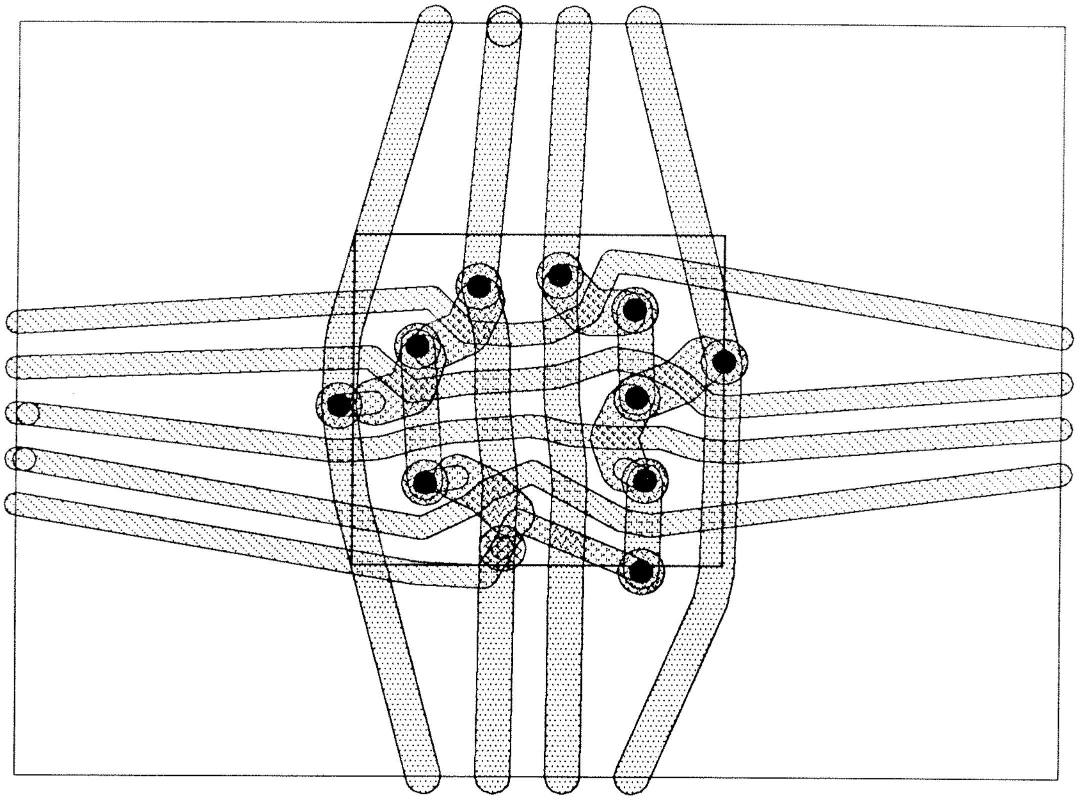


Figure C.3 1 Cell - end of second stage

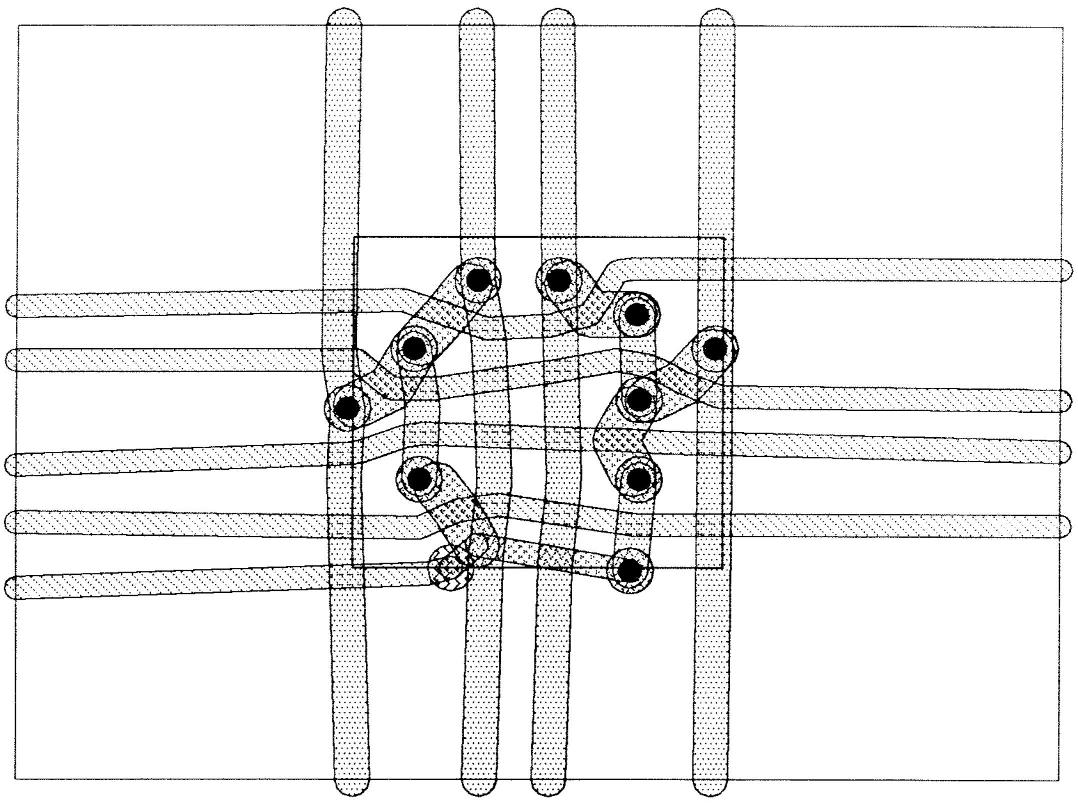


Figure C.4 1 Cell - end of final stage

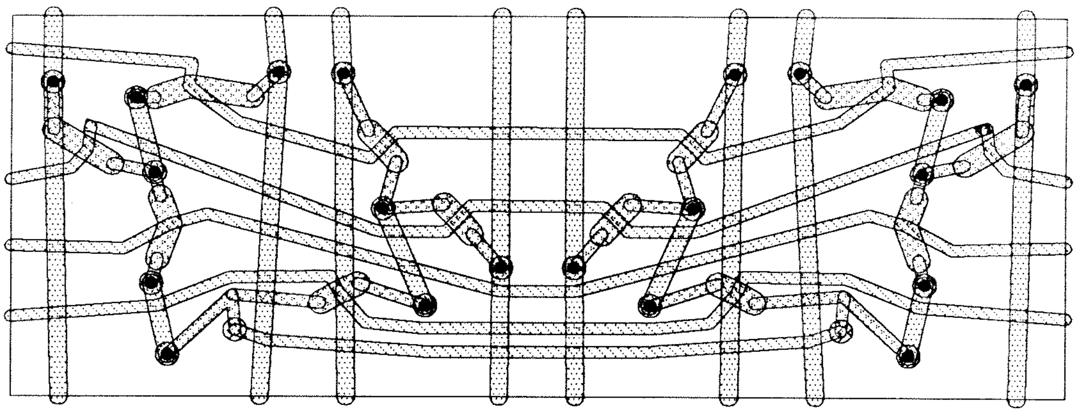


Figure C.5 1 by 1 Cell - Initial

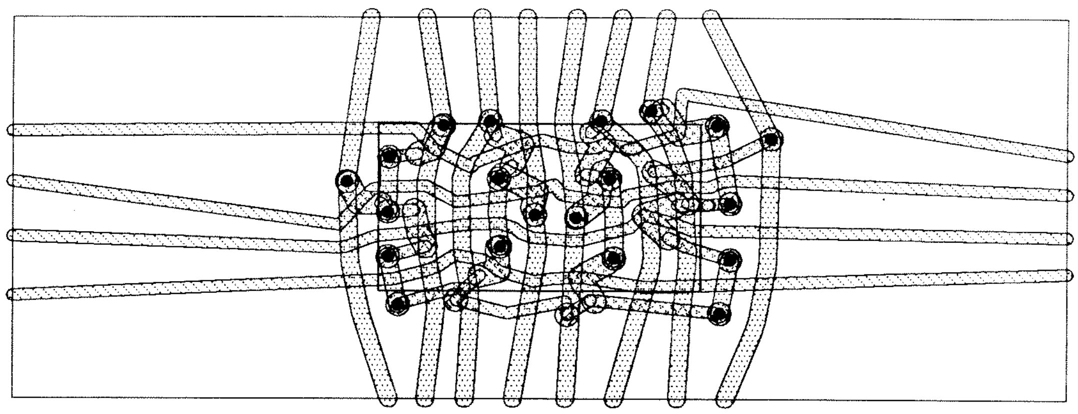


Figure C.6 1 by 1 Cell - end of first stage

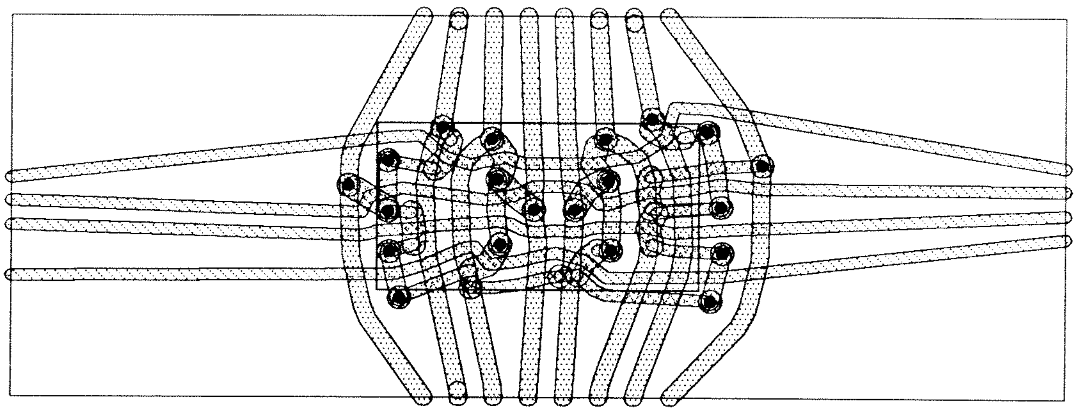


Figure C.7 1 by 1 Cell - end of second stage

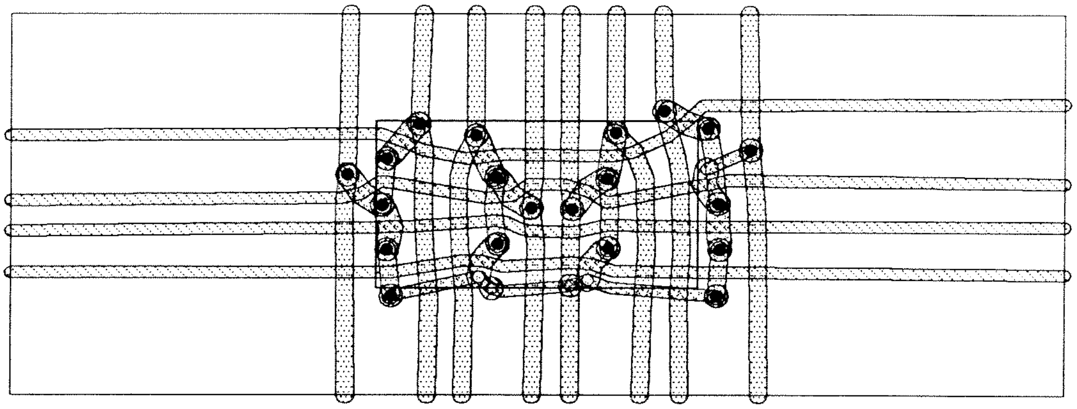


Figure C.8 1 by 1 Cell - end of final stage

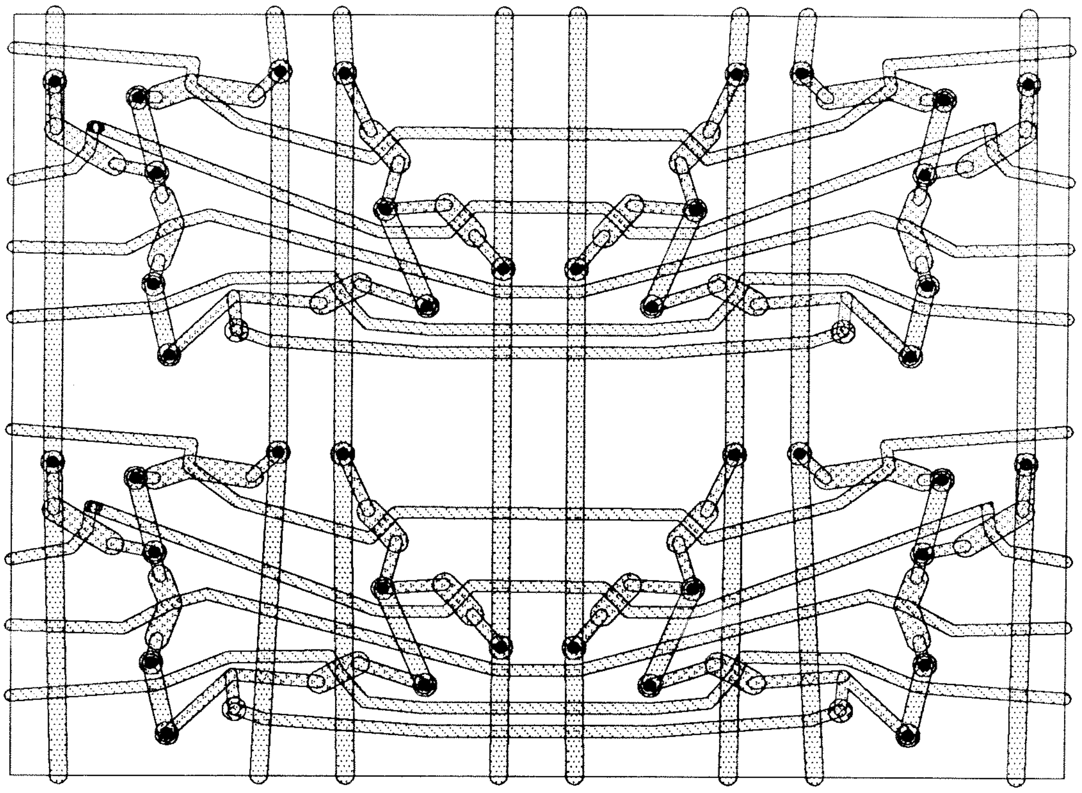


Figure C.9 2 by 2 Cell - Initial

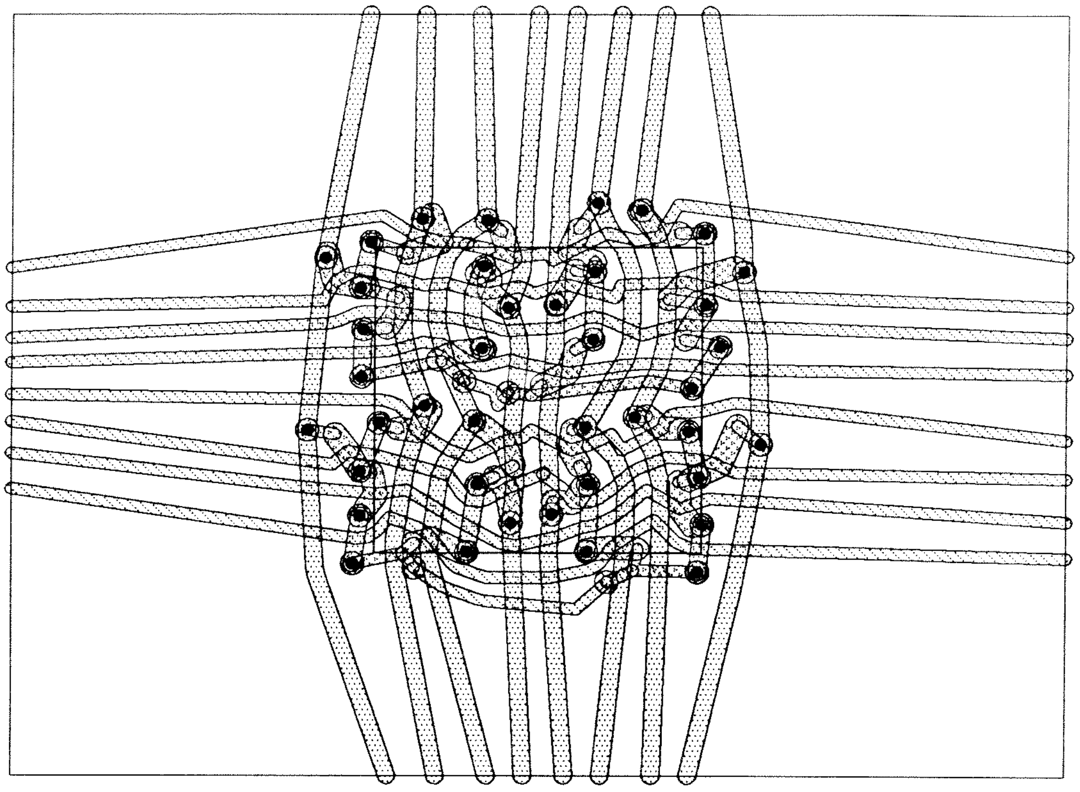


Figure C.10 2 by 2 Cell - end of first stage

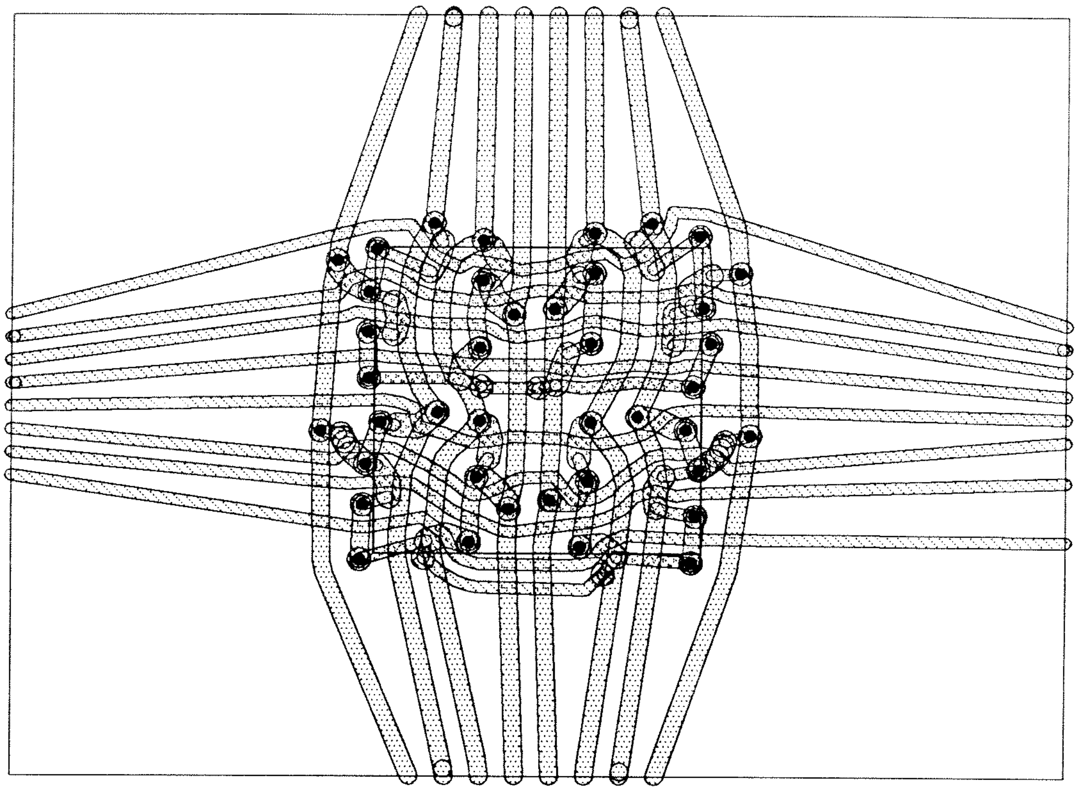


Figure C.11 2 by 2 Cell - end of second stage

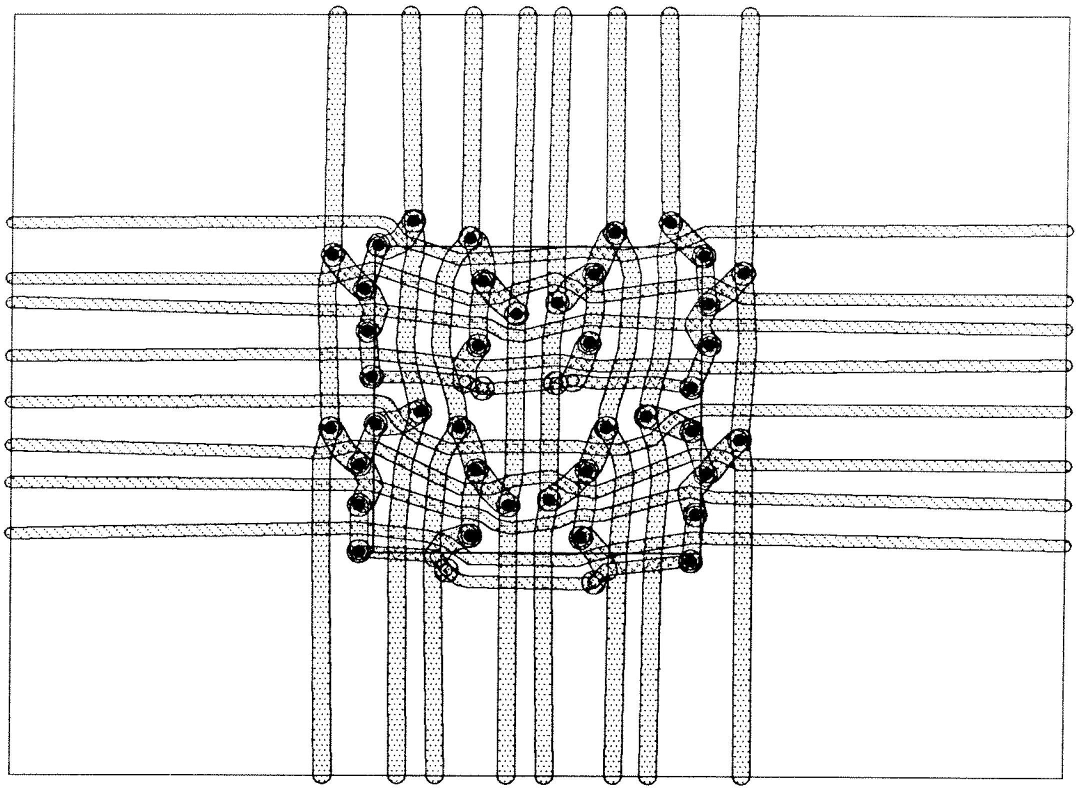


Figure C.12 2 by 2 Cell - end of final stage

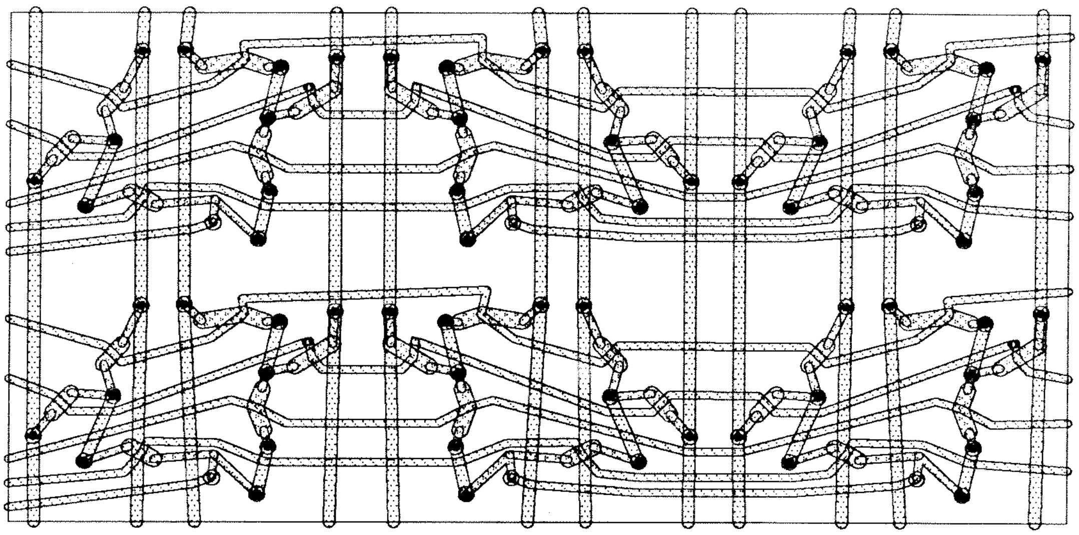


Figure C.13 2 by 3 Cell - Initial

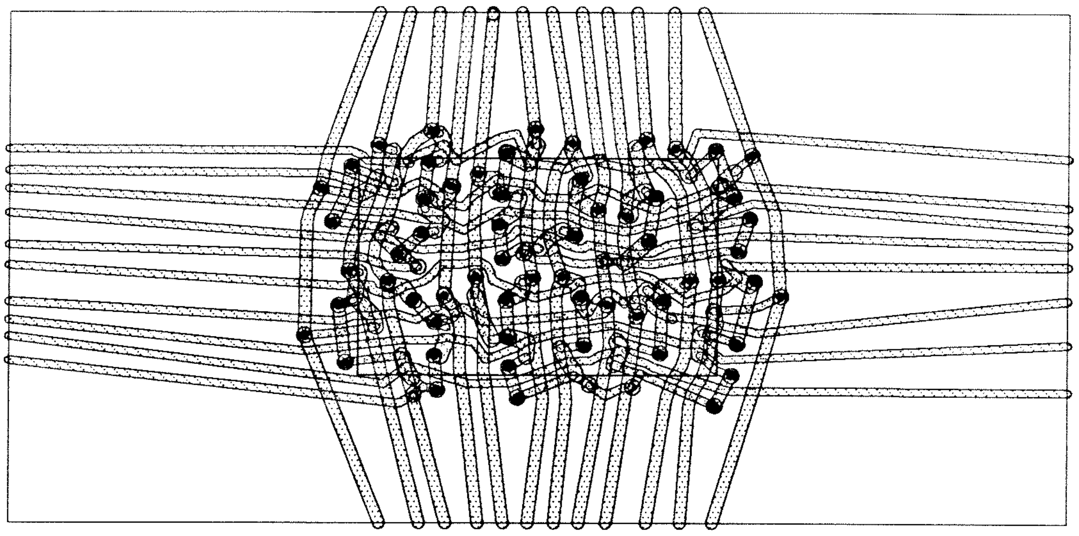


Figure C.14 2 by 3 Cell - end of first stage

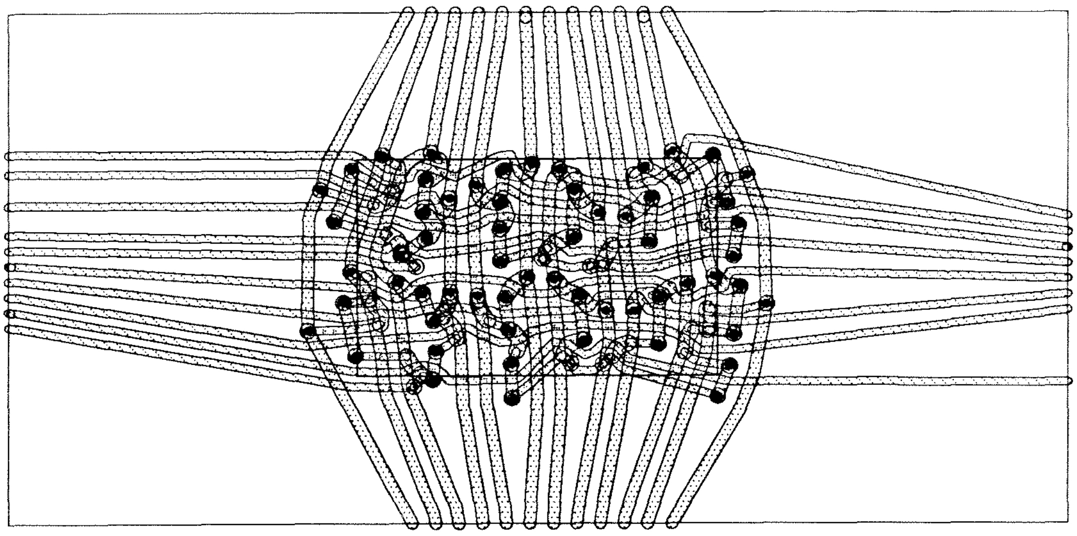


Figure C.15 2 by 3 Cell - end of second stage

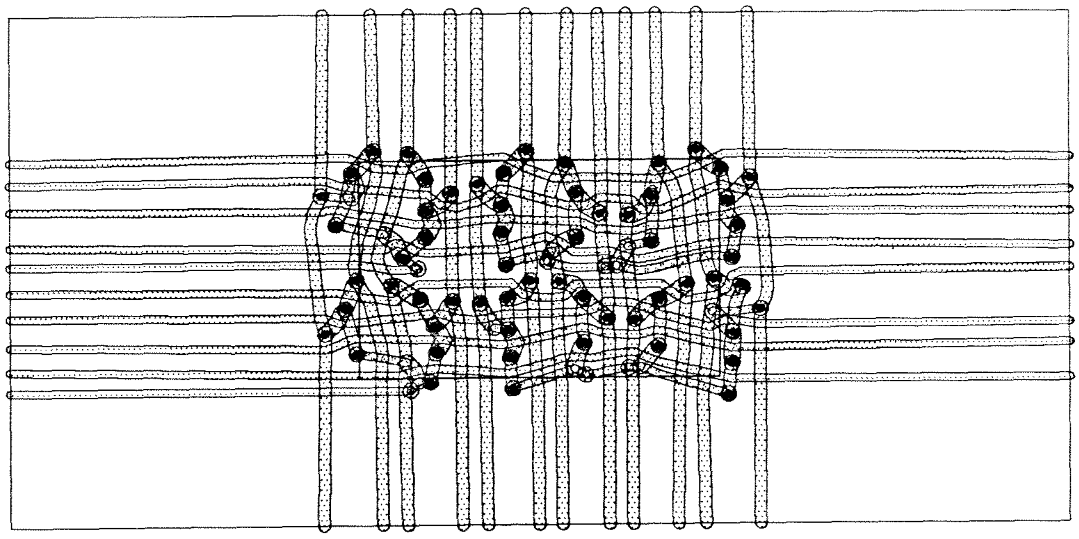


Figure C.16 2 by 3 Cell - end of final stage

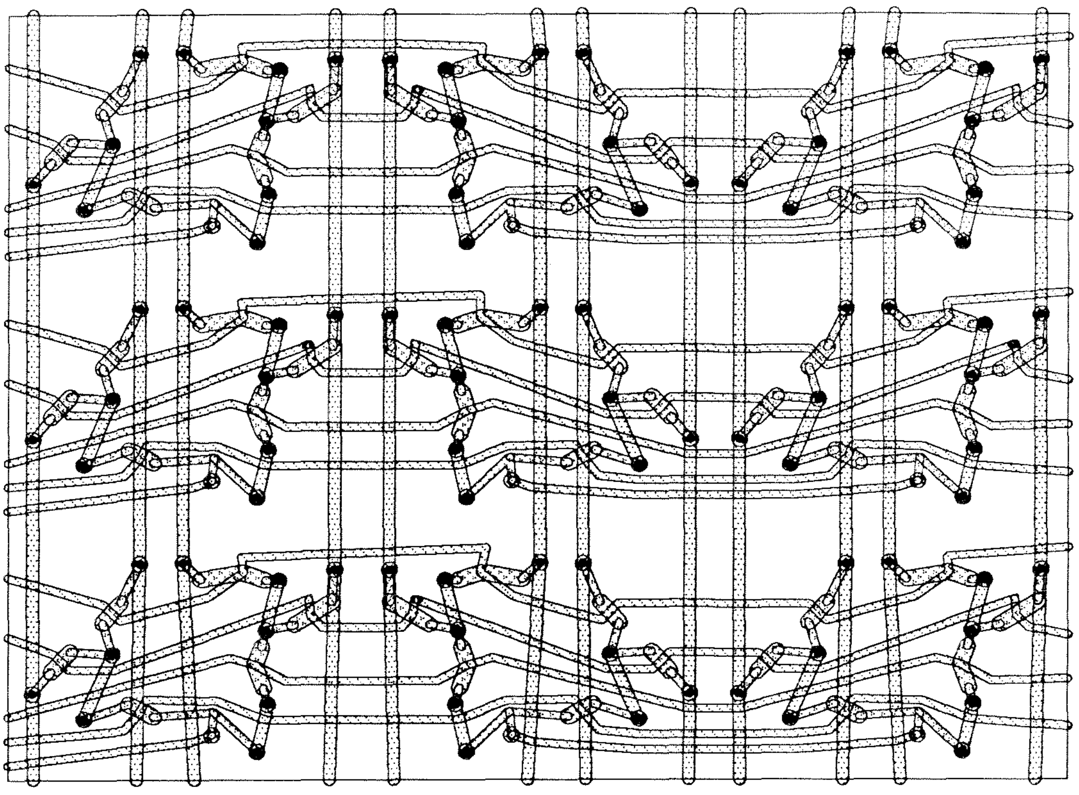


Figure C.17 3 by 3 Cell - Initial

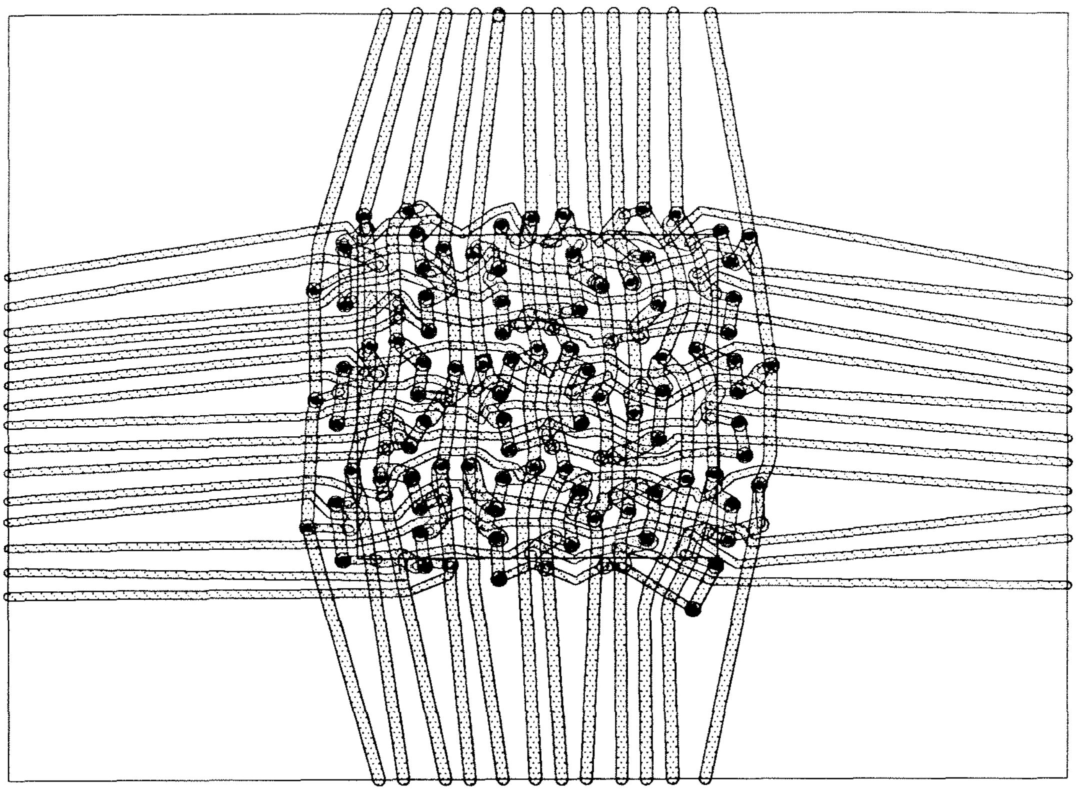


Figure C.18 3 by 3 Cell - end of first stage

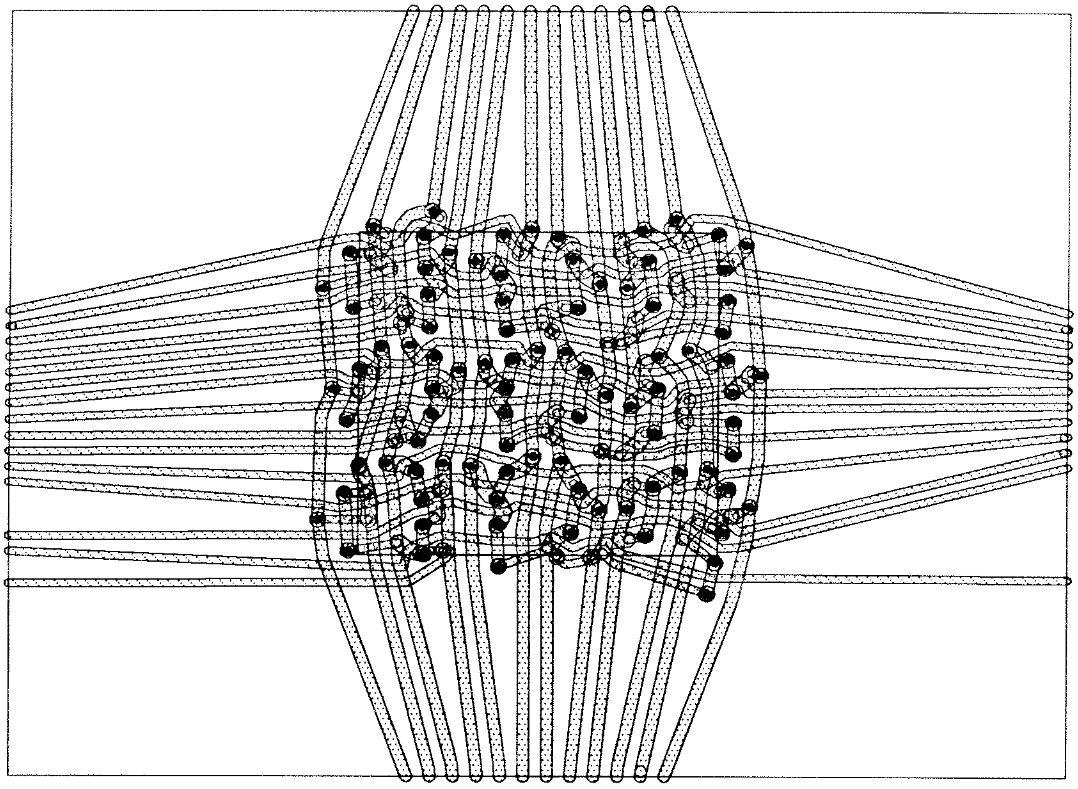


Figure C.19 3 by 3 Cell - end of second stage

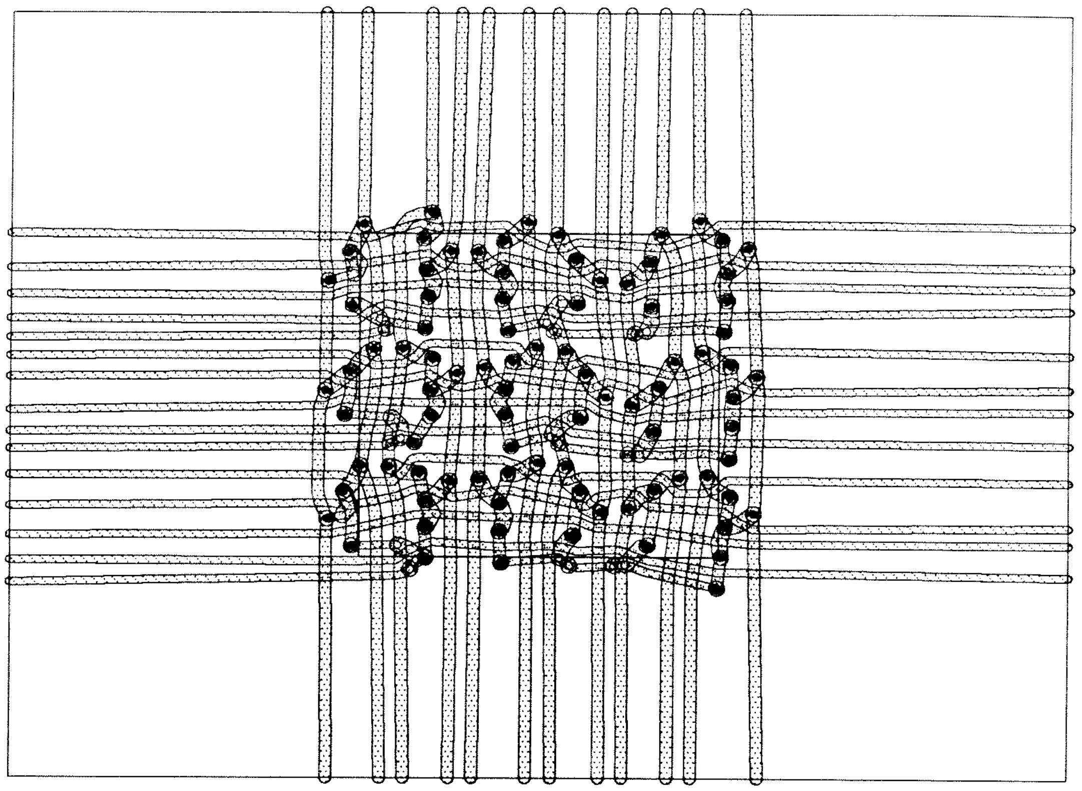


Figure C.20 3 by 3 Cell - end of final stage

Bibliography

[Ackland 83] Bryan Ackland and Neil Weste.
"An Automatic Assembly Tool for Virtual Grid Symbolic Layout".
Proceedings of the IFIP TC 10/WG 10.5 International Conference on Very Large Scale Integration, Trondheim, Norway, 16–19, August 1983, pages 457–466.

[Akers 70] S.B. Akers, J.M. Geyer, and D.L. Roberts.
"IC Mask Layout with a Single Conductor Layer".
Proceedings of the 7th Design Automation Workshop, San Francisco, 1970, page 7–16.

[Ayres 85] Ron Ayres.
A New MOSIS Service: FUSION.
Information Science Institute, Marina del Rey, California, 1985.

[Bales 82] Mark W. Bales.
Layout Rule Spacing of Symbolic Integrated Circuit Artwork.
Masters Dissertation, University of California, Berkeley, 1982.

[Barton 80] E.E. Barton, I. Buchanan.
"The Polygon Package".
"Computer Aided Design", Volume=12(3), page 3-11, January, 1980.

[Bentley 80] J.L. Bentley, D. Haken, and R.W. Hon.
"Statistics on VLSI Designs".
Carnegie-Mellon University, April 1980.

[Birtwistle 73] G.M. Birtwistle, O-J Dahl, B. Myhrhaug and K. Nygaard.
SIMULA begin.
Auerbach Publishers Inc., 1973.

[Buchanan 80] Irene Buchanan.
Modeling and Verification in Structured Integrated Circuit Design.
PhD Dissertation, Computer Science, University of Edinburgh, Edinburgh Scotland, 1980.

[Bryant 81] Randal E. Bryant.
A Switch-Level Simulation Model for Integrated Logic Circuits.
PhD Dissertation, Massachusetts Institute of Technology 1981.

[Bryant 82] R. Bryant, M. Schuster, and D. Whiting.
"Mossim II: A Switch-Level Simulator for MOS LSI User's Manual".
California Institute of Technology, Computer Science Department, Technical Report #5033, Pasadena California, August 1982.

[Cho 77] Y.E. Cho, A.J. Korenjak, and D.E. Stockton.
"FLOSS: An Approach to Automated Layout for High Volume Designs".
Proceedings of the 14th Design Automation Conference, June 1977, pages 138–141.

[Cho 85] Y. Eric Cho.
"A Subjective Review Of Compaction".
Proceedings of the 22nd Design Automation Conference, June 1985, pages 396–403.

[Dunlop 79] A.E. Dunlop.
Integrated Circuit Mask Compaction.
PhD Dissertation, Carnegie-Mellon University, 1979.

[Dunlop 80] A. Dunlop.
"SLIM- The Translation of Symbolic Layouts into Mask Data".
Computer Aided Design, Volume 12, Number 6, June 1980, pages 595–602.

[Fairbairn 78] D.G. Fairbairn, J.A. Rowson.
"ICARUS: An Interactive Integrated Circuit Layout Program".
Proceedings of the 15th Design Automation Conference, pages 188–192 1978.

[Franco 81] D. Franco, and L. Reed.
"THE CELL DESIGN SYSTEM".
Proceedings of the 18th Design Automation Conference, pages 240–247 1981.

[Frey 83] A. Frey.
Rabbit Chip.
California Institute of Technology, Pasadena California, 1983.

[Garey 79] M.R. Garey and D.S. Johnson.
COMPUTERS AND INTRACTABILITY: A guide to the Theory of NP–Completeness.
W.H. Freeman and Company, San Francisco, 1979.

[Hedges 82] T.S. Hedges, K.H. Slater, G.E. Clow, and T. Whitney.
"The Siclops Silicon Compiler".
Proceedings of IEEE ICCC, September 1982, Pages 277–280.

[Hsueh 80] Min-Yu Hsueh.
Symbolic Layout and Compaction of Integrated Circuits.
PhD Dissertation, University of California, Berkeley, Memorandum No. UCB/ERL M79/80, 1980.

[Johannsen 81] David Lawrence Johannsen.
Silicon Compilation.
PhD Dissertation, California Institute of Technology, Technical Report #4530, Pasadena California, 1981.

[Johnson 84] David S. Johnson.
"Optimization By Simulated Annealing: An Experimental Evaluation".
Workshop on Statistical Physics in Engineering and Biology, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, April 26–7, 1984.

[Kingsley 82] C. Kingsley.
"Earl: An Integrated Circuit Design Language".
Masters Dissertation, California Institute of Technology, Computer Science Department, Technical Report #5021, Pasadena California,1981.

[Kingsley 84] Christopher Kingsley.
"A Hiererachical, Error–Tolerant Compactor".
Proceedings of the 21st Design Automation Conference, June 1984, pages 126–132.

[Kirkpatrick 83] S. Kirkpatrick, C.D. Gelatt, Jr., M.P. Vecchi.
"Optimization by Simulated Annealing".
SCIENCE, Volume 220, May 13, 1983, pages 671–680.

[Kirkpatrick 84a] Scott Kirkpatrick.
"Optimization by Simulated Annealing: Quantitative Studies".
IBM Research, Yorktown Heights, N.Y. 10598 1984.

[Kirkpatrick 84b] Scott Kirkpatrick.
List of Abstracts.
Workshop on Statistical Physics in Engineering and Biology, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, April 26–7, 1984.

[Lattin 79] Bill Lattin.
"VLSI Design Methodology: The Problem of the 80's for Microprocessor Design".
Very Large Scale Integration, California Institute of Technology, Charles L. Seitz, Pasadena, California, January 22–24 1979.

[Lien 81] Sheue-Ling Lien.
"Toward a Theorem Proving Architecture".
Masters Dissertation, California Institute of Technology, Computer Science Department, Technical Report #4653, Pasadena California, 1981.

[Locanthi 78] B. Locanthi.
"LAP: A Simula Package for IC Layout".
California Institute of Technology, #1862, Pasadena California, 1978.

[Lundy 84b] M. Lundy.
"Applications of the Annealing Algoritm".
Handout at the Workshop on Statistical Physics in Engineering and Biology, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, April 26–7, 1984.

[Lutz 84] Chris Lutz, Steve Rabin, Chuck Seitz, and Don Speck.
"Design of the Mosaic Element".
Conference on Advanced Research in VLSI, Massachusetts Institute of Technology, Paul Penfield, Jr., Editor, January 1984.

[Maley 85] F.M. Maley.
"Compaction with Automatic Jog Introduction".
1985 Chapel Hill Conference on Very Large Scale Integration, Henry Fuchs, Editor, Computer Science Press, Inc. 1985.

[Mead 80] C.A. Mead and L.A. Conway.
Introduction to VLSI Systems.
Addison Wesley, 1980.

[Mead 83] Carver A. Mead.
"Structural and Behavioral Composition of VLSI".
Proceedings of the IFIP TC 10/WG 10.5 International Conference on Very Large Scale Integration, Trondheim, Norway, 16–19, August 1983, pages 3–8.

[Metropolis 53] Nicholas Metropolis, Arianna W. Rosebluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller.
"Equation of State Calculations by Fast Computing Machines".
The Journal of Chemical Physics, Volume 21, Number 6, June 1953, pages 1087–1092

[Milne 84] George Milne.
"Towards Verifiably Correct VLSI Design".
University of Edinburgh, Internal Report CSR–164–84, The King's Buildings, Mayfield Road, Edinburgh, EH9 3JZ, 1984.

[Minter 80] C. Minter.
"Charles Terminal Care Package".
California Institute of Technology, Silicon Structures Project, SSP Report #3804, Pasadena California, 1980.

[Moore 79] Gordon E. Moore.
"Are We Really Ready for VLSI?".
California Institute of Technology, *Very Large Scale Integration*, Charles L. Seitz, Pasadena, California, January 22–24 1979.

[Mosis 84] The MOSIS Project.
The MOSIS System(what it is and how to use it).
USC/Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, California 90292-6695, March 1984.

[Mosteller 81] R.C. Mosteller.
"A Leaf Cell Design System".
Masters Dissertation, California Institute of Technology, Computer Science Department, Technical Report #4317, Pasadena California, 1981.

[Mosteller 81] R.C. Mosteller.
"REST A leaf Cell Design System".
Very Large Scale Integration, University of Edinburgh, Academic Press, Edinburgh, Scotland, ISBN 0–12–296860–3, John P. Gray, August 1981.

[Mosteller 82] R.C. Mosteller.
"Coma".
California Institute of Technology, Silicon Structures Project Spring Review, Pasadena California, 1982.

[Mosteller 82] R.C. Mosteller.
"An Experimental Composition Tool".
Conference on Microelectronics, The Institution of Engineers, Australia, 1982.

[Mosteller 84] R.C. Mosteller.
"The 2-D Compaction Problem".
Workshop on Statistical Physics in Engineering and Biology, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, April 26–7, 1984.

[Ousterhoust 81] J.K. Ousterhoust.
"Caesar: An Interactive Editor for VLSI Layouts".
VLSI Design, Fourth Quarter, pages 34–41, June 1981.

[Ousterhoust 84] J.K. Ousterhoust, Gt.T. Hamachi, R.N. Mayo, W.S. Scott, and G.S. Taylor.
"Magic: A VLSI Layout System".
Proceedings of 21^{st} D.A. Conference, pages 152–159, June 1984.

[Rabin 84] Steve Rabin.
"Mosaic Memory Design Notes".
California Institute of Technology, Computer Science Department, Technical Report 5162:DF:84, Pasadena California, 1984.

[Robertson 81] P.S. Robertson.
The Production of Optimized Machine–Code for High–Level Languages using Machine–Independent Intermediate Codes.
PhD Dissertation, University of Edinburgh, November 1981.

[Rowson 80] J.A. Rowson.
Understanding Hierarchical Design.
PhD Dissertation, California Institute of Technology, Pasadena California, 1980.

[Rupp 81] C.R. Rupp.
"Components of a Silicon Compiler System".
Very Large Scale Integration, University of Edinburgh, Academic Press, Edinburgh, Scotland, ISBN 0-12-296860-3, John P. Gray, August 1981.

[Sastry 82] Sarma Sastry and Alice Parker.
"The Complexity of Two–Dimensional Compaction of VLSI Layouts".
IEEE International Conference on Circuits and Computers, ICCC 82, September 1982.

[Schiele 83] W.L. Schiele.
"Improved Compaction By Minimized Length Of Wires".
Proceedings of the 21^{st} Design Automation Conference, June 1984, pages 121–127.

[Segal 80] R. Segal, C. Carroll, G. Tarolli, S. Trimberger, R. Sproull, R. Lyon, D. Lang.
"SSP Basic Software Package".
California Institute of Technology, Silicon Structures Project, SSP Report #4024, Pasadena California, 1980.

[Segal 84] R. Segal.
"Structure, Placement and Modeling".
Masters Dissertation, California Institute of Technology, Computer Science Department, Technical Report #5132, Pasadena California, 1984.

[Seitz 85a] Chuck Seitz.
"The Cosmic Cube".
Communications of the ACM, Volume 28, Number 1, January 1985.

[Seitz 85b] Chuck Seitz.
Private Communication.
California Institute of Technology, Pasadena California, 1985.

[Speck 85] Don Speck.
Private Communication.
California Institute of Technology, Pasadena California, 1985.

[Sproull 79] R. Sproull and R. Lyon revised S. Trimberger.
"The CALTECH INTERMEDIATE FORM for LSI LAYOUT DESCRIPTION".
California Institute of Technology, Silicon Structures Project, SSP MEMO #2686, Pasadena California, 1979.

[Stallman 80] R. M. Stallman.
"EMACS Manual for TWENEX Users".
Massachusetts Institute of Technology, Artificial Intelligence Laboratory, AI Memo #555 1980.

[Steele 85] Craig S. Steele.
"Placement of Communication Processes on Multiprocessor Networks".
Masters Dissertation, California Institute of Technology, Computer Science Department, Technical Report #5184:TR:85, Pasadena California, 1985.

[Suaya 84] Roberto Suaya.
"Simulated Annealing and VLSI Control Structures".
Workshop on Statistical Physics in Engineering and Biology, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, April 26–7, 1984.

[Sutherland 78]	I.E. Sutherland. "The Polygon Package". California Institute of Technology, Computer Science report 1438, Pasadena California, 1978.
[Tanner 83]	John Tanner. Motion Detection Chip. California Institute of Technology, Pasadena California, 1983.
[Tompa 80]	M. Tompa. "An Optimal Solution to a Wire-Routing Problem". *Proceedings of the Twelfth annual ACM Symposium on Theory of Computing*, 28–30 April 1980.
[Trimberger 80]	S. Trimberger. "The Proposed Sticks Standard". California Institute of Technology, Silicon Structures Project, SSP MEMO #3487, Pasadena California, 1980.
[Trimberger 81]	S. Trimberger, J. Rowson, C. Lang, and J.P. Gray. "A Structured Design Methodology and Associated Software Tools". *IEEE Transactions on Circuits and Systems.* CAS-28, 7, July 1981.
[Trimberger 82]	S. Trimberger, and J. Rowson. "Riot - - A Simple Graphical Chip Assembly Tool". Proceedings of the 19th Design Automation Conference, June 1982, pages 371–376.
[Trimberger 83]	S.M. Trimberger. *Automated Performance Optimization of Custom Integrated Circuits.* PhD Dissertation, California Institute of Technology, 1983.
[Vecchi 83]	Mario P. Vecchi and Scott Kirkpatrick. "Global Wiring by Simulated Annealing". IEEE Transactions on Computer-Aided Design, Volume CAD–2, Number 4, October 1983.
[Watanabe 83]	Gershen Kedem and Hiroyuki Watanabe. "Graph–Optimization Techniques for IC Layout and Compaction". Proceedings of the 20th Design Automation Conference, June 1983, pages 113–120.

[Watanabe 84] Hiroyuki Watanabe.
IC Layout Generation and Compaction Using Mathematical Optimization.
PhD Dissertation, Computer Science, The University of Rochester, New York 1984.

[Williams 77] John Williams.
"Sticks - A New Approach to LSI Design".
Masters Dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1977.

[Wipfli 78] J. Wipfli.
"A SIMULA Graphic Package".
California Institute of Technology, Silicon Structures Project, SSP Report #1929, Pasadena California, 1978.

[Weste 81] N. Weste.
"Virtual Grid Symbolic Layout".
Proceedings of the 18th Design Automation Conference, pages 225–233, 1981.

[Weste 81a] N. Weste and Bryan Ackland.
"A Pragmatic Approach to Topological Symbolic IC Design".
Very Large Scale Integration, University of Edinburgh, Academic Press, Edinburgh, Scotland, ISBN 0–12–296860–3, John P. Gray, August 1981.

[White 84] Steve R. White.
"Concepts of Scale in Simulated Annealing".
Proceedings of ICCD, October 1984, Pages 646–651.

[Whitney 82] T. Whitney, and T. Hedges.
"Pooh User's Manual",
California Institute of Technology, Computer Science Department Technical Report # 5029, Pasadena California, 1982.

[Whitney 83] T. Whitney, and C.A. Mead.
"Pooh: A Uniform Representation for Circuit Level Designs".
Proceedings of International Conference on VLSI, Trondheim, Norway, August 1983 Pages 401–411.

[Whitney 85] T.E. Whitney.
Hierarchical Composition Of VLSI Circuits.
PhD Dissertation, California Institute of Technology, Pasadena California, 1985.

[Wilcox 85] C.R. Wilcox, M.L. Dageforde, and G.A. Jirak.
Mainsail *Language Manual Version 9.0.*
Xidak, Inc., Menlo Park, California, 1985.

[Wong 83] M. Schlag,Y.Z. Liao and C.K. Wong.
"An Algorithm for optimal two-dimensional compaction of VLSI layouts".
North Holland INTEGRATION, the VLSI journal 1, 1983, pages 179–209.

[Wong 83a] Y.Z. Liao and C.K. Wong.
"An Algorithm to Compact a VLSI Symbolic Layout with Mixed Constraints".
Proceedings of the 20th Design Automation Conference, June 1983, pages 107–112.