Hierarchical Composition of VLSI Circuits

Thesis by

Telle Elizabeth Whitney

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

1985

(Submitted May 21, 1985)

to my parents
Din, Beverly, and June

Acknowledgments

Special acknowledgment is due to Dr. Ivan Sutherland whose ideas on using circles and lines for representing geometry, from his polygon package, inspired the work presented in this thesis.

I'd also like to express my deepest appreciation to Carver Mead, my advisor, for his warmth, his insights, his constant support and patience. He has been a tremendous inspiration and has contributed greatly to my life through the expression of his philosophy in our many lengthy discussions, his views on science, and his incredible humanism.

I'd like to thank my committee, each contributed to this thesis in a unique way: to Chuck Seitz for his concern and help over the years, to Dick Lyon whose careful study and extensive comments made the final weeks a real joy, and to Al Barr and Dr. David Rutledge for their comments and help.

The formation of "the group" created an environment of comradeship that allowed scientific study, and I am grateful for the chance to participate. I'd like to thank all of my fellow students for their friendship and support. An extra special thanks is due Jim Campbell for doing all those things that need to be done in order to maintain a productive lab.

Thanks to the many participants of the Silicon Structures Project, who are too numerous to mention. The people associated with the project made the initial years of my graduate study both productive and a pleasure.

I am truly indebted to John Wawrzynek for both his constant support and remarkable insights. He always provided a reason for trying a little harder when things were going wrong. He was a sounding board for my ideas, and examined each one carefully and thoroughly. He pointed out when I was wrong, and sometimes applauded when I made progress. He was always there.

This thesis was formatted using Donald Knuth's TEX, and printed on an Apple LaserWriter, with figures described in Postscript directly incorporated into the text using Textset software. My thanks to the people at Adobe systems, Inc., and Textset, Inc. for their help in making this formatting approach possible. Glenn Gribble deserves special appreciation since he pulled the various software pieces together, and locally made it possible. Also, thanks to Lounette Dyer for originating and suggesting this formatting approach, and providing software support. An extra special thanks to Calvin Jackson, who was always there to answer questions, offer suggestions, and provide technical support.

I owe many thanks:

to my colleagues in the field who provided fruitful interactions over the years,

to my family who always encouraged me to strive for the best,

and to my friends who made the day-to-day worthwhile.

This work was supported by the System Development Foundation.

Abstract

A transistor level representation for VLSI circuits is presented. This representation is simple but general, technology independent, hierarchical, and maintains connectivity, circuit schematic information, and the information for mask geometry.

A transistor level cell is represented as the interconnection of devices along with their types, sizes and placement, and the cell's typed ports. Connection is represented explicitly by shared connection points. The ports describe the interface between this cell and other cells. This representation, together with a set of synthesis and analysis rules, enforces the description of strictly legal designs. The synthesis rules ensure that each structure is correct by construction. The analysis rules check for geometrical design rule violations which cannot, by their nature, be enforced by construction.

A file of technology dependent information indicates how to implement each transistor type, interconnect type and connection point type, as well as how structure types may interact.

Cells described in this representation may be composed hierarchically to form larger cells. Given a valid composition, the topology, geometry and connectivity of the composite structure is guaranteed to be legal.

A working system supporting this hierarchical representation is also described. This system currently supports design rules for nMOS and cMOS/bulk, and has produced chip descriptions that have been both fabricated and tested.

Table of Contents

List of Figures

1. Introduction

The design of a correct integrated circuit has become an increasingly difficult problem as the complexity has increased [Moore 79]. Current Very Large Scale Integrated Circuit (VLSI) chips contain over 100,000 transistors, and million transistor chips are foreseen in the near future. Using the well-known structured design methodology [Mead 80, Mead 83], the complexity of designing these large systems is drastically reduced by adopting a hierarchical design methodology and by maintaining consistent design representations throughout the design process.

A hierarchical design methodology defines a large system design problem in terms of a set of smaller more manageable problems. A hierarchical integrated circuit design is defined recursively as a cell where a cell is the interconnection of cells and transistors.

This definition allows us to view each cell as an abstraction. Each cell is represented by its ports and port characteristics. With this information, cells may be interconnected without detailed knowledge of their internal structures. The interconnection then produces a new cell abstraction, as shown in Figure 1-1. Internally, each cell is characterized by three unique representations: (1) a functional specification, (2) a timing model, and (3) a structural description.

The functional specification of the cell is an abstraction of the behavior of the cell circuit. Initially, this functional description may be a circuit description, but a low level circuit description quickly becomes unwieldy. Instead, there is often a concise description that completely characterizes the cell behavior. The key to the functional hierarchy is cell functional descriptions that are composed to form the functional description at the next level of the hierarchy [Chen 83]. Thus the behavior of a cell may be derived from that of its sub-cells and is used to describe and simulate the design. Often cells are semantic units [Chen 83], where only the fixed point behavior of the cell affects connected cells. Otherwise, the cells are syntactic units, and are treated differently during system simulation. In either case, functional descriptions are composed hierarchically, and may be used for simulation or formal verification.

The timing model is a description of the dynamic delay thorough a cell [Lin 84]. At the lowest level, the timing may be derived directly from the circuit. As with the functional description, the timing information is composed to form the timing description at the next level in the hierarchy. Each cell, at every level in the hierarchy, includes a timing abstraction that fully characterizes the cell and may be used to evaluate the delay through the design.

The structural description of the cell is a description of the cell topology. Each cell is an interconnection of transistors and other cells. A structural cell abstraction indicates the topology of interest during composition. As with the functional description and

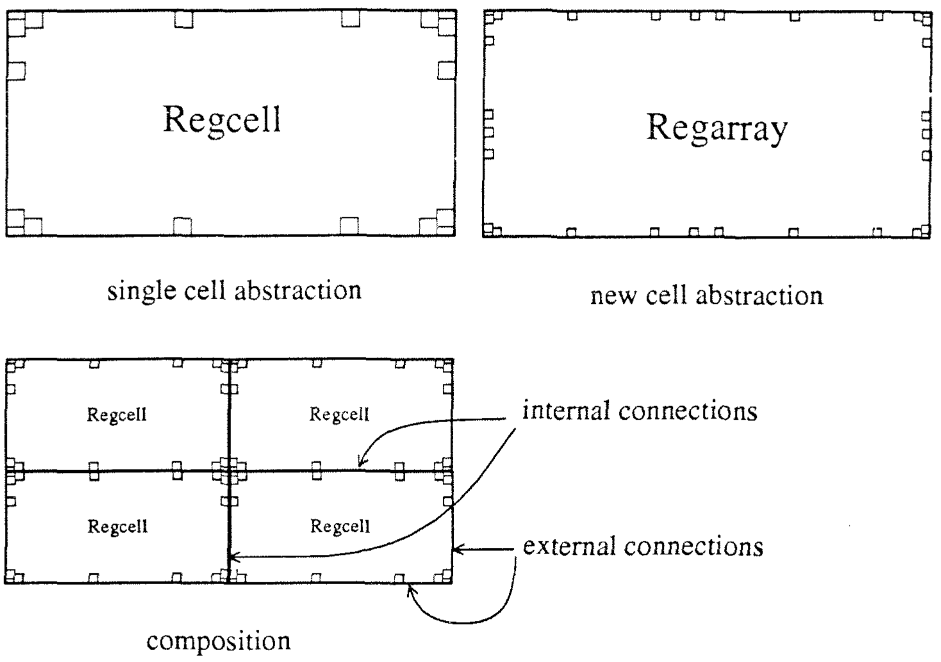


Figure 1-1: *Cells and Interconnection of Cells*

the timing abstraction, the cell structural abstraction is composed to form the structural description at the next level in the hierarchy.

In the structured design methodology described in this thesis, cells are defined by these three abstractions. Assuming consistent representations at the lowest level of the hierarchy, composition produces consistent representations at all other levels. This thesis focuses on the structural aspects of a cell, but the structural representation always has a corresponding functional and timing definition.

1.1 Structural Design Representations

The design of a cell spans many structural design representations. Each representation incorporates information important to the overall design. Often the difference between two related descriptions, or "levels" of the design, is that one contains an abstraction of the other, which has implementation details. A typical structural approach to a VLSI design incorporates the following design representations: (1) architectural floorplan, (2) logic diagram, (3) circuit diagram, and (4) mask geometry description. Figure 1-2 illustrates an example of the four design representations of a simple register. Larger designs will, of course, have more levels. An architectural floorplan is a definition of the large blocks which together comprise the system. A logic diagram is the interconnection of logic blocks. Each logic block is a small unit such as a multiplexor or an adder. A circuit diagram is the interconnection of transistors. The mask geometry description is a collection of colored polygons.

Given multiple levels, a designer is faced with ensuring consistency between

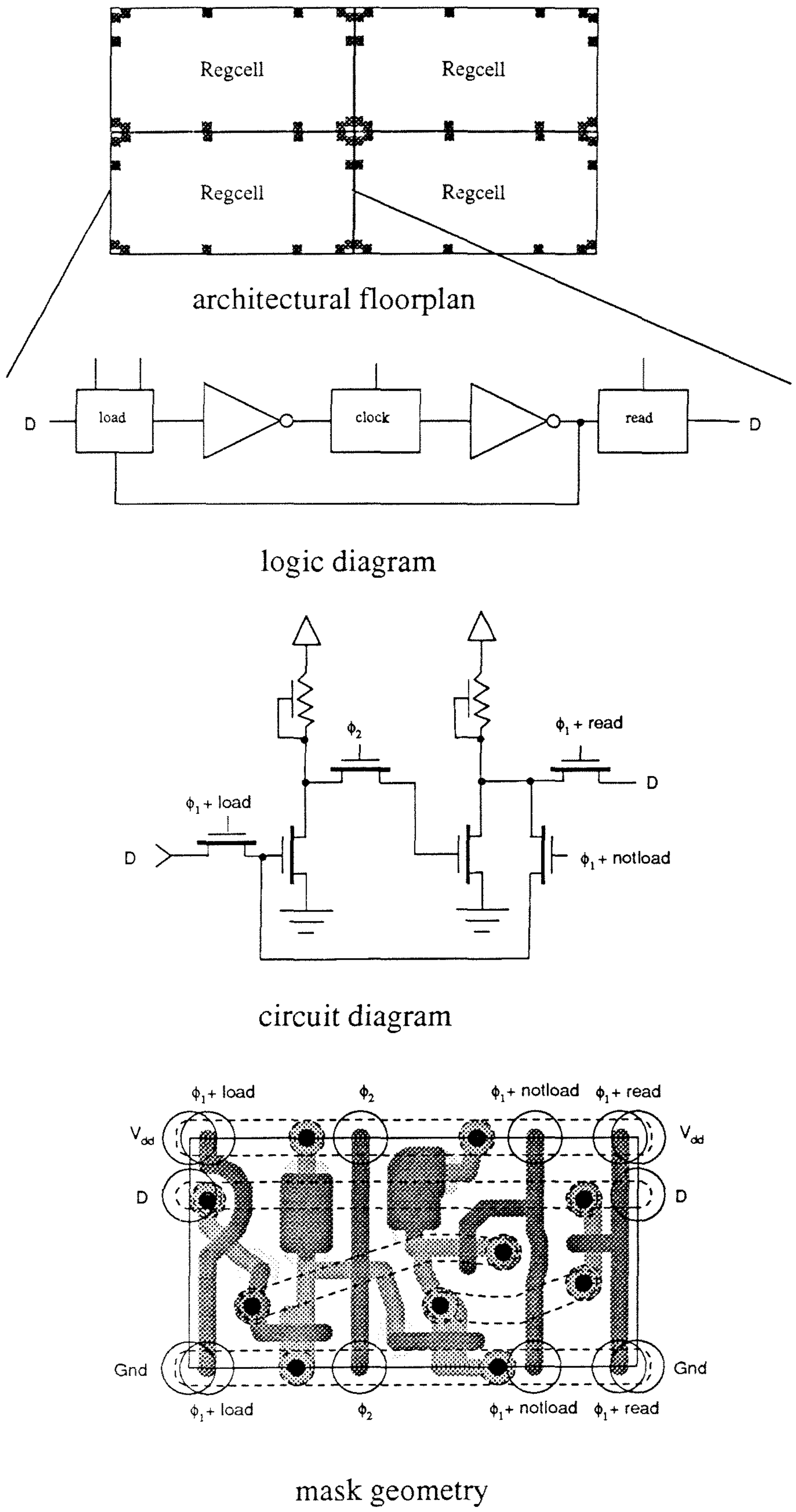


Figure 1-2: Design Representations

the different abstractions. It is absolutely necessary that the logic diagram correctly implements the architectural specification, that the circuit correctly implements the logic diagram, and that the layout corresponds to the circuit.

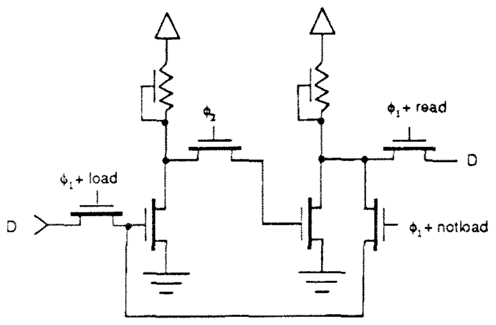
Given the ability to view a single cell as an abstraction, we now focus on the problems of specifying the implementation of the abstraction at the lowest level. The problems are of the same type the designer encounters at every level of design: ensuring that one level of representation or refinement matches the other. At the circuit level, it is essential that the mask geometry implements the circuit.

Several representations are used in generating mask geometry from a circuit diagram (shown in Figure 1-3): (1) Circuit Diagram, (2) Sized Schematic, (3) Topological Sized Schematic, (4) Sticks Diagram, and (5) Mask Geometry. Each is a refinement of the previous one. The process can be thought of as a successive binding of the necessary attributes of a finished design. First, a designer must assign sizes to each of the circuit's transistors, to form a sized schematic. Then, the designer decides on the relative placement of the cell's interfaces or ports. The ports determine a large portion of the topology of the cell. Next, the designer worries about relative placement of the transistors to create the topological sized schematic. Then, layers are assigned to the circuit elements and the interconnect to form the sticks diagram. Finally, the designer generates a set of boxes and polygons on different fabrication layers to define the mask geometry. In actual practice expert designers often skip one of more of these steps, and carry the information informally.

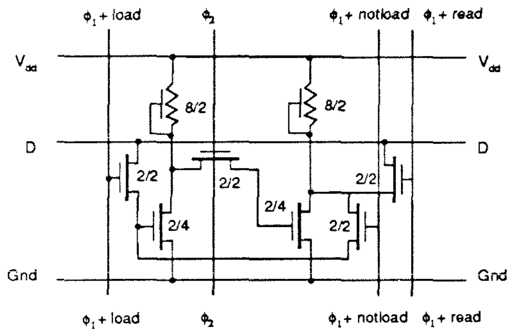
The designer then faces the task of determining whether the mask geometry implements the sized schematic. This job is typically done by extracting the circuit [Baker 80] from the geometrical description and then checking to see if the sized schematic and the extracted circuit are the same. Finally, after ensuring circuit consistency, the designer processes the geometry using a Design Rule Checker (DRC) program to find geometrical design rule (GDR) violations. The methods available for checking consistency and performing DRCs tend to be time consuming and error prone.

This thesis describes an alternative approach in which both a sized schematic and the mask geometry are generated from a single representation. Rather than performing the time-consuming and error-prone task of checking the consistency between these two levels of design representation, circuit diagram and mask geometry, a designer specifies the circuit in a uniform representation. The problem of checking the consistency between the two representations is eliminated. Also, the representation embodies a natural but precise way of expressing rules dictated by the fabrication technology. Design Rules such as device sizing and GDRs may be expressed at the level of the structures of the technology, namely, devices and interconnection.

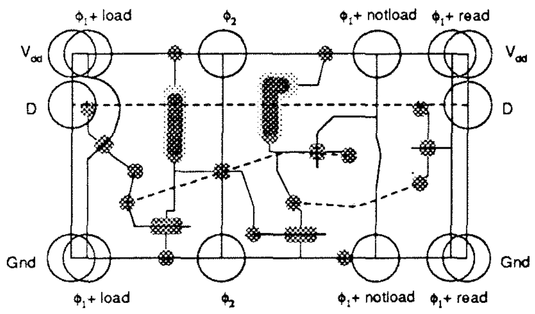
The approach of describing a circuit at a symbolic level was first addressed by John Williams in his "sticks" representation [Williams 77]. He proposed a scheme in which a cell is described in terms of structures such as devices and interconnection wires. He then proposed a "compaction" scheme where a program moves structures as close together as possible according to the GDRs. Several compactors are currently working [Mosteller 81, Hsueh 80]. These systems have many nice properties, but they have not been widely accepted. A fundamental limitation of existing compaction approaches is that they separate the inherently two-dimensional problem of geometrical design rule



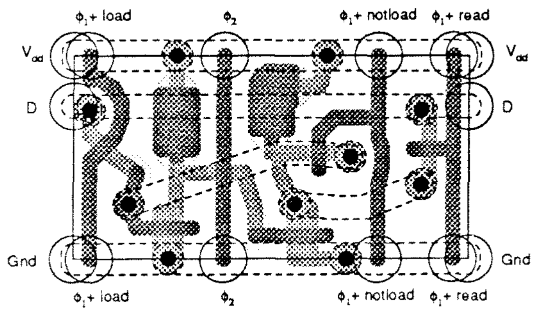
circuit diagram



topological sized schematic



stick diagram



mask geometry

Figure 1-3: Circuit to Geometry Representations

checking into two separate one-dimensional problems. Compaction is done first in the x-direction and then in the y-direction (or vice-versa). Rules that are two-dimensional in nature are either ignored or mapped into one of the two static dimensions. Another peculiar quirk of current sticks systems is that devices may extend in only one dimension. Thus if the length of the device is in the x-direction, then the width can only be in the y-direction and the transistor may not bend.

This thesis describes a symbolic "sticks-like" representation and a set of operational algorithms called Pooh [Whitney 83]. It has the following attributes: (1) simple but general (2) independent of any particular technology, (3) capable of expressing the designer's intent [McGrath 80], (4) expresses the design rules in terms of the representation primitives, (5) represents high density designs, (6) contains connectivity, circuit schematic, and mask geometry information, and (7) is hierarchically composible.

Figure 1-4 indicates how the Pooh representation is used. Pooh provides a service to the user by defining an automatically checked representation for designs at the circuit level. From this representation switch level simulator input, circuit level simulator input, geometry, and a schematic diagram are easily generated. The user interface may either be graphical or language based. Alternatively, Pooh is used as the base level cell representation for a silicon compiler.

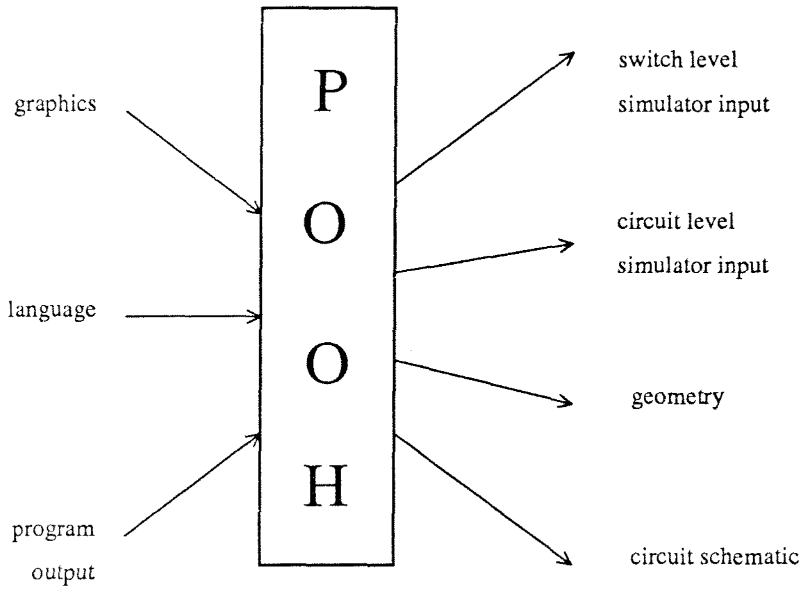


Figure 1-4: The Pooh System

Pooh was designed with MOS technology in mind. Currently both nMOS and cMOS/bulk are supported, though other technologies are easily expressed. All the simple illustrative examples in this thesis are nMOS. Some of the actual designs are cMOS.

1.2 Overview of the Thesis

There are four pieces of work presented in order to establish the representation presented in this thesis:

1) the representation of transistors and interconnection,

2) the synthesis algorithms for constructing these elements,
3) the analysis algorithms for detecting illegal interactions between elements, and
4) a general technique for abstracting the topology of a cell and composing cells into topologically and geometrically correct high-level cells.

Chapter 2 is an overview of the transistor level Pooh representation. The primitive elements in the Pooh representation are lines, arcs and circles. Transistors, interconnection wires, and connection points are defined in terms of these elements.

There are three types of algorithms necessary to ensure the correct description of the topology, geometry, and connectivity of the Pooh representation. Chapter 3 describes the Pooh synthesis algorithms. The synthesis algorithms allow the construction of GDR correct transistors, wires and connections points. Chapter 4 describes the algorithms for propagating the node information of the circuit. Chapter 5 introduces the Pooh analysis algorithms. The analysis algorithms check the interactions between the elements necessary in order to guarantee the GDR correctness of the circuit. Since each element is constructed according to the GDRs, these interactions are typically simple and small in number.

The composition of circuits is the key to making the Pooh representation a viable solution for describing VLSI. Chapter 6 describes the composition of Pooh cells. First, an abstraction of the cell is derived. Then cells are composed to form the next level in the hierarchy. In Pooh, a legal composition guarantees the topology, geometry and connectivity of the composite structure.

Chapter 7 addresses some of the considerations of a system that supports the Pooh representation. An actual Pooh system uses a circle approximation instead of a true circle, and the affect of the circle approximation is explored. Interactions between the Pooh paths and points must be quickly detected to make Pooh a useful tool. A data structure that allows easy access to the Pooh elements is described. This chapter also describes a simplified version of the Pooh algorithms. These algorithms support vertical, horizontal and forty-five degree lines, and are integer based.

Chapter 8 concludes this thesis by describing an implementation of a Pooh system. This implementation supports two separate interfaces. The original Pooh system was an embedded language, used as the base level representation for the Siclops silicon compiler. The second system described is Tigger, a graphical circuit editor that enforces GDRs interactively.

2. The Pooh Representation

2.1 The Pooh Topological Representation

The standard circuit-level representation of basic cells for a VLSI design is the topological sized schematic. To date, the information conveyed by this level in the design has had no formal representation. It incorporates relative placement, transistor sizes and interface ports. In addition, layer information is usually attached to interconnect by either convention or context. Interconnect either connects to a transistor, or goes around it. A sized schematic does not depend on absolute placement, but is drawn with relative placement. The relative placement of the interface ports in a cell determine a large portion of its internal topology. Thus the step from a topological sized schematic to the Pooh representation is a small one.

A topological sized schematic cell is specified by its devices, interconnection wires and ports. Each device is a transistor with a position and an orientation. Each interconnection wire is a single electrical node between two or more devices. Each port is a connection point of one of the cell's electrical nodes to nodes of other cells. Figure 2-1 illustrates a Pooh topological sized schematic.

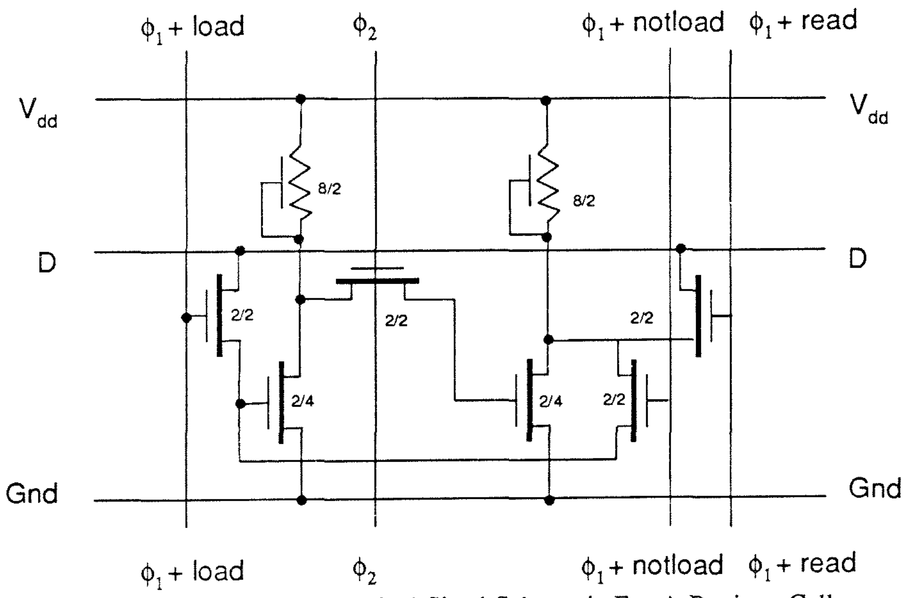


Figure 2-1: Topological Sized Schematic For A Register Cell

A topological sized schematic is expressed in Pooh using three primitive structures: (1) a typed point, (2) a segment and (3) a path. A typed point is a position along with associated type information. It is used to represent both devices—where the type indicates the type of transistor, and connection points. Each transistor definition includes strength information in the form of Length/Width, an orientation, and three electrical nodes—source, drain and gate. A segment is a directed line segment and an arc. The end of the line segment is the tangent point on a circle of radius r centered at the segment end point. The arc is defined by the end of the line segment, the segment end point, and a direction. If the arc radius r is zero, then the arc is of zero length, and the end of the line segment is the segment end point. Paths are sequences of segments. An interconnection wire is represented as a set of connected paths of the same electrical node. A connection point is a typed point (or "dot") where a connection occurs between either two paths, or a path and a device. Ports are represented as typed points where the type indicates what type of connection may occur at this port. Figure 2-2 shows examples of these elements.

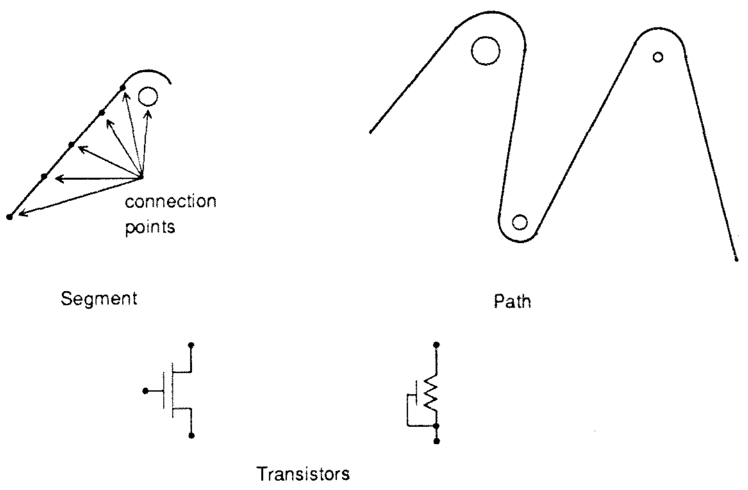


Figure 2-2: Paths, Segments and Transistors

A topological sized schematic differs from a standard circuit schematic in several important ways. In particular, additional information is present in a topological sized schematic that is not in the normal schematic.

A circuit schematic may be represented as a bipartite graph, with vertex partitions representing the nodes and the transistors. The edges between the two partitions represent the connectivity. The information conveyed by this circuit schematic can be fully captured using a one-dimensional representation. For example, let $n_1 \ldots n_m$ denote the node vertices, and $t_1 \ldots t_p$ denote the transistor vertices. Then, we may represent the circuit as two arrays: N of length m, and T of length $3p$, and represent the connectivity as stored indices from one array into the alternate array. Obviously, this representation is one dimensional, and yet it contains sufficient information to fully represent the circuit graph.

A topological sized schematic incorporates information beyond the normal schematic. The device sizes are necessary attributes of the transistor vertices in any actual design. The crucial step, however, is the relative placement of the ports and the interconnect. This step takes the schematic from the realm of a one-dimensional representation to the two-dimensional world of integrated circuits.

Relative placement is the process of basing the position of an object, in this case interconnect or ports, on the placement of other objects. Previous work has focused on automatically generating a stick diagram from a circuit schematic coupled with device sizes and a partially ordered set of ports [Ng 84, Wolf 83]. This work separates the potentially non-planar circuit graph into a set of planar graphs, which must correspond to the different interconnection layers on the target chip. The next step is to create two constraint graphs from these circuit graphs, which describe the ordering of the components in the x-dimension and in the y-dimension, similar to the constraint graphs used in one-dimensional sticks compactors [Mosteller 81, Hsueh 80].

In fact, the ordering inherent in the topological sized schematic is not one-dimensional at all. A topological sized schematic contains lines, representing interconnection, and components, representing transistors and ports. This schematic's relative placement information is two-dimensional in nature, and is naturally representing by lines "missing" or "going around" components.

A stick diagram [Williams 77] is a topological sized schematic with "colored" interconnect and devices. The color indicates the layer on the silicon implementation for the interconnect and transistors. A stick diagram is shown in Figure 2-3. A stick transistor level design is represented as the interconnection of transistors along with their types, sizes and placement. The primitives of the representation are transistors, interconnect and connection points. Transistors are the devices of the technology, each of which have a length and a width. Each transistor is either a path or a point. Interconnect is implemented by typed paths connected by two or more connection points. Connection points are points, with associated type information, where an electrical connection occurs between two or more paths.

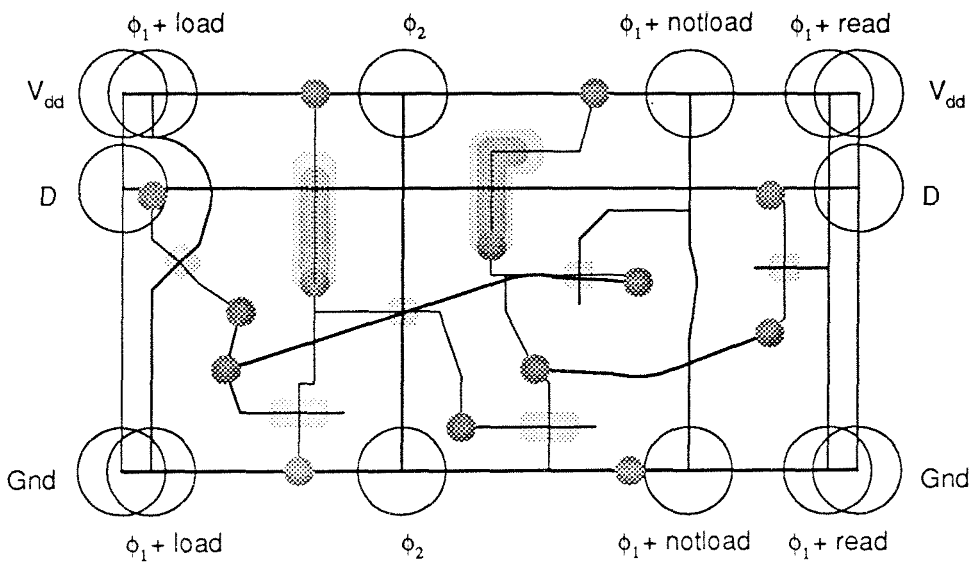


Figure 2-3: Register Cell Stick Diagram

In the Pooh topological representation, the notion of relative placement is implemented in the following manner: transistors and interconnect are represented as a series of segments as shown in Figure 2-4. Each segment may either connect to a point, or go around it. Connectivity is represented explicitly by multiple references to the same point. If a segment goes around a point, the distance is a "miss distance" and forms an arc around the point. Small transistors are often conveniently formed by the intersection of two segments as shown in Figure 2-4.

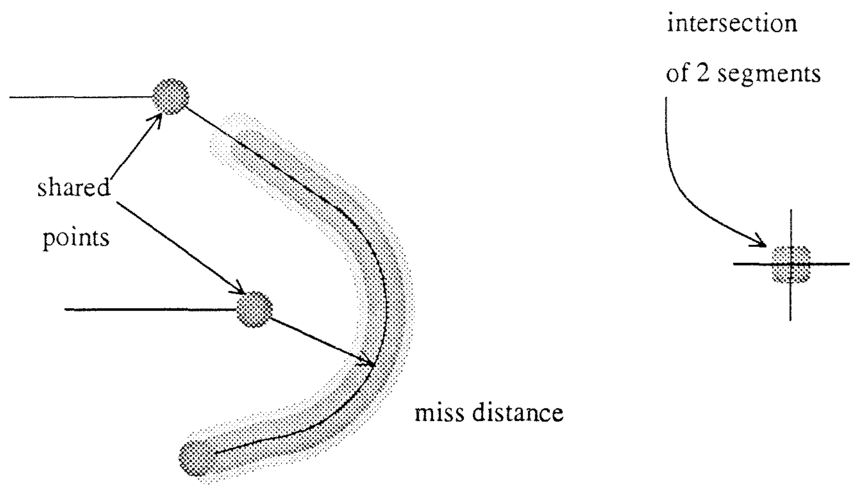


Figure 2-4: Segments, Points and Transistors

A circuit described in this manner maintains the connectivity information through shared points and represents a properly connected set of transistors. This information is more than enough to represent a topological sized schematic and a stick diagram, and is nearly sufficient to construct design-rule correct mask geometry.

The Pooh representation formalizes the notions of shared points, segments, and "miss" distances into a complete topological circuit representation. This representation, together with the Pooh synthesis and analysis rules, form the Pooh algebra of circuits.

Interconnect and transistor paths are a sequence of path segments. Every path segment is a directed line segment and an arc. The placement of each path segment is defined by two connection points. Other connection points may be attached along the straight portion of a path segment.

Connection and transistor points express the connectivity of the paths. A transistor point is formed at the intersection of the channel and gate path segments. A connection point is formed either by placing the end point of a path segment or by connecting one path to another.

This representation coupled with the Pooh synthesis and analysis rules ensure the description of strictly legal designs. The synthesis rules allow only valid interconnection between points and paths and construct transistor paths such that they meet the length and width constraints. The analysis rules ensure that path segments cross only at valid connection and transistor points.

2.2 GDR Representation

A stick diagram differs from a schematic in that the stick interconnect wires, transistors and connection points are "colored" according to the Geometrical Design Rules (GDRs).

GDRs are rules dictated by the fabrication technology that govern how structures may be placed together. A stick diagram demonstrates some of the assumptions about a particular technology, such as that there are multiple interconnection layer types, and multiple transistor types.

Since a chip description is an interconnection of transistors, it make sense to represent the GDRs in terms of transistors and their connections instead of the traditional technique of representing GDRs in terms of the fabrication layers. Thus the Pooh GDR representation contains a list of technology dependent information that describes how to create transistors, interconnect and connection points, and how to connect these structures together.

The GDR descriptions are generic definitions for a particular technology. Each element in a cell is an instance of one of the definitions. The GDRs fall into two categories—descriptions of legal structures and descriptions of legal interactions between structures. The descriptions of individual structures include a logical layer type. This layer type is symbolic as opposed to a physical or geometrical layer [Williams 77]. For example, in nMOS, polysilicon, diffusion and metal are logical layer types. Each of these have a single physical layer—polysilicon, diffusion and metal. A Low Resistance Connector (LRC) is an nMOS logical layer that is comprised of three physical layers—polysilicon, diffusion and buried. In cMOS, there are at least four logical layers—polysilicon, metal, p-diffusion, and n-diffusion. Both the p-diffusion and the n-diffusion logical layers are often composed of multiple geometrical layers.

Each interconnect type description consists of:

1) a logical layer type,
2) a minimum width,
3) the geometrical layers that comprise this logical layer, and
4) the geometrical overlap rules.

Each transistor type description has a:

1) transistor type,
2) channel logical layer,
3) gate logical layer,
4) minimum gate width,
5) minimum channel width,
6) minimum extension of gate and channel layers beyond the gate region,
7) the geometrical layers that comprise this transistor, and
8) the geometrical overlap rules.

Connection points are connections between one or more paths. If these paths are of the same type, then the connection point rules are defined by the transistor and interconnect rules. However if the connection point is between paths of different types, then the connection point is called a contact and has its own unique GDRs. Each contact description consists of:

1) the connection point type,
2) the number of physical connection points,

3) for each point, the legal logical connection layers,
4) the legal angle or angles of connection for each legal interconnect type, and
5) reference to the geometrical implementation description.

The GDR interaction rules include both spacing and angle rules. Each spacing description includes both a rule that applies to structures of different electrical nodes, and a second rule that applies to structures of the same node. Angle rules are always between paths of the same node. Spacing rules are:

1) Minimum Interconnect to Interconnect Spacing,
2) Minimum Interconnect to Transistor Spacing,
3) Interconnect to Contact Spacing,
4) Transistor to Transistor Spacing,
5) Transistor to Contact Spacing, and
6) Contact to Contact Spacing.

Angle rules are:

1) Minimum Interconnect to Interconnect Angle,
2) Minimum Interconnect to Transistor Angle, and
3) Minimum Transistor to Transistor Angle.

2.3 Geometrical and Topological Representation

Pooh constructs the geometry of a circuit in a manner aimed at reducing the design-rule enforcement task. Devices and interconnect are expressed as a center-line, and are generated algorithmically. The geometry of transistors and interconnect is formed by maintaining a constant radius around each line segment and point. Connection points are expressed as points and are constructed as circles. We can use simple Euclidean geometrical operations between center-lines and points to perform all design rule calculations. The problem of performing operations between the complex edges of mask polygons is completely avoided. This section describes these instances of the GDR definitions.

Transistors are either a path or a point. A "wide" transistor path is defined with:

1) a transistor type,
2) a name,
3) a list of the segments that form the path of the gate,
4) a length and width,
5) one or more transistor points where connections occur,
6) two or more connection points,
7) the distance from the start of the path to the transistor's gate region,
8) the point indicating the end of the gate region, and
9) three electrical node numbers (source, drain and gate).

A "long" transistor path is similarly defined with the channel, rather than the gate, as the center-line layer. A transistor point has the following attributes:

1) a transistor type,

2) a name,
3) an X,Y coordinate,
4) the channel segment or segments,
5) the gate segment, and
6) three electrical node numbers.

Each interconnect path has the following attributes:

1) an interconnect type,
2) a width,
3) a list of the segments that form the path,
4) multiple connection points,
5) an electrical node number, and
6) the electrical capacitance of this path.

Each connection point has these attributes:

1) a type,
2) an X,Y coordinate,
3) an orientation,
4) a node number, and
5) a list of the path segments that share this point.

2.4 Circuit Synthesis Rules

There are several types of synthesis rules in Pooh. Some rules are inherent to the Pooh system and govern how points and segments are composed to form structures. For example: a path segment may "miss" a connection point, but is restricted to connect ("miss" distance equal to zero) to a transistor. The other types of rules govern both how structures are connected together and how structures are physically made. These rules are kept in the technology description, and include information such as the actual distance a path segment must "miss" a connection point, and how to make a particular type of connection point or transistor.

Pooh enforces the construction of correct interconnect and transistor paths as shown in Figure 2-5. It enforces a minimum width on all interconnect paths, and a minimum width and length on transistor paths. For a transistor path, the length and width parameters coupled with the transistor overlap rules yield the end point of the transistor gate area. Such a path then continues as an interconnect path with the logical layer of the center-line.

Pooh applies a set of rules to the construction of the sequence of segments that form the interconnect and transistor paths. Each segment may include many connection points, but two points define its physical placement—the first and last points. A segment, rather than connecting to these two points, may instead "miss" them. The arc of this segment is defined by its last point, a "miss" distance and the next segment. The sign of the "miss" distance indicates the sense in which the segment goes around the point. If the "miss" distance is zero, then the last point is indeed a connection point, and the arc is of zero length. Pooh then ensures that the path layer is one of the legal logical

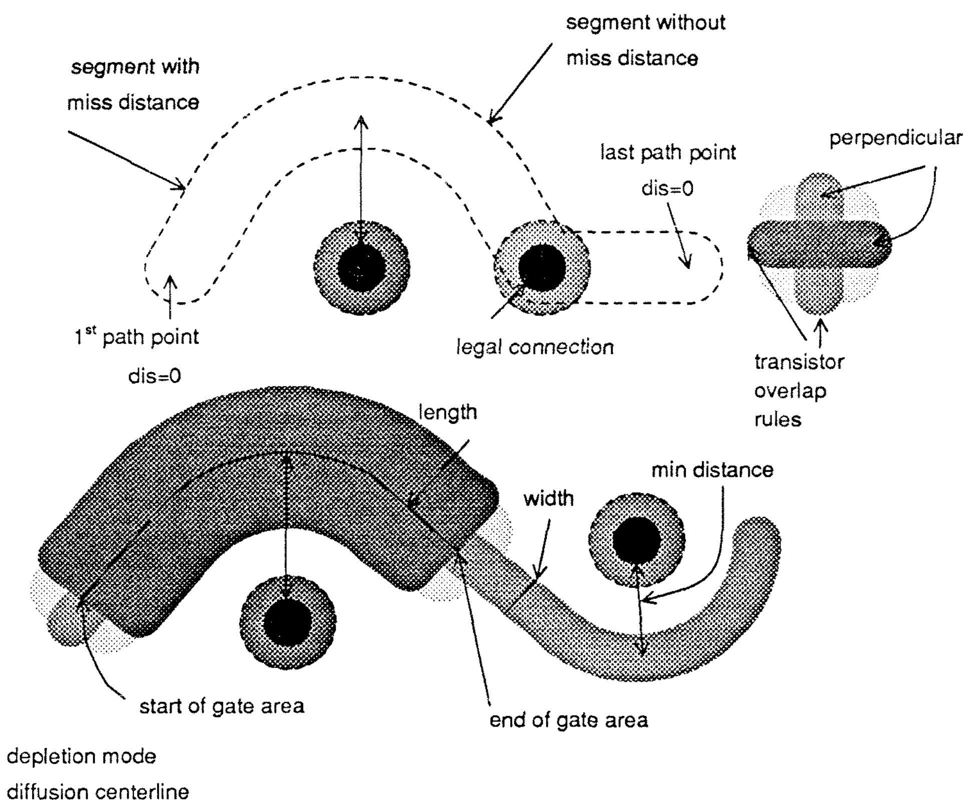


Figure 2-5: Synthesis Rules

connection layers for the point type, and adjusts the angle of the connection point to match the legal angle of connection as shown in Figure 2-6. The first and the last point of the path must have a zero "miss" distance. Figure 2-7 illustrates the use of the "miss" distances.

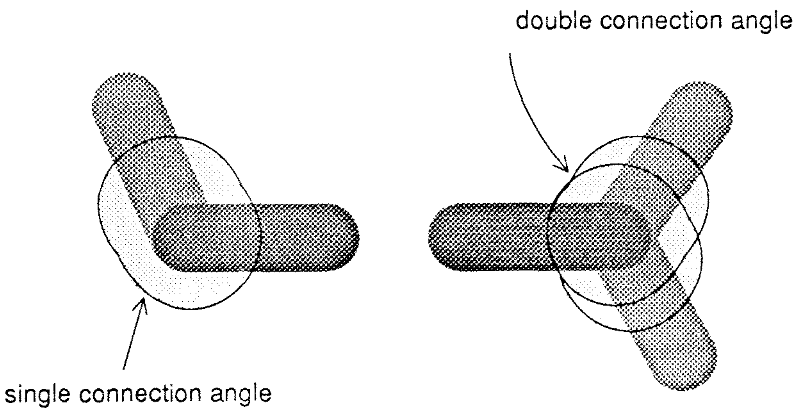


Figure 2-6: Legal Angle of Connection

The spacing rules govern the segment to point "miss" distances based on the path to point spacing and the path to path spacing. If the current segment is the first segment to "miss" a point, or if there isn't a spacing rule between this segment and the point's other segments, then the "miss" distance is calculated as the spacing rule between this segment's path type and the point's type. Otherwise Pooh finds the point's segment with the largest "miss" distance that interferes with this segment and adds to the "miss" distance the spacing rule between this segment's path type and the interfering segment's path type.

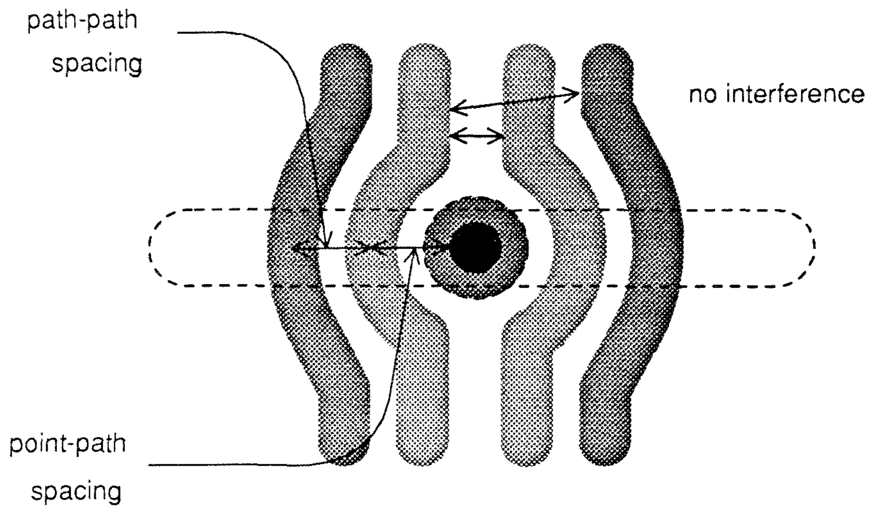


Figure 2-7: "Miss" Distance

Pooh enforces an additional set of rules in the construction of transistor points. The channel and the gate segments must connect ("miss" distance equal to zero) to a transistor point. If the transistor point is not part of a transistor path, the gate segment and the channel segment must be perpendicular as shown in Figure 2-5.

There are paths that define entire regions: the well in cMOS is one example. These "surround" paths are similar to other pooh paths, except that the polygons described by the paths are closed — the first point is equivalent to the last point. Pooh constructs these paths by enforcing a direction during segment synthesis; all surround paths are clockwise.

2.5 Circuit Analysis Rules

Most GDRs are enforced correct by construction during the creation of the Pooh paths. Some analysis is necessary to enforce the remaining GDRs. Analysis determines illegal interactions between interconnect paths, transistor paths and connection points. The synthesis rules ensure the correct construction of the individual structures, so what remains at the analysis phase is ensuring correct spacing and angles between adjacent structures. During the analysis phase Pooh only looks at structures that are "close" to the current structure, i.e., within the maximum GDR bounding box that surrounds the current structure. Figure 2-8 illustrates possible interactions.

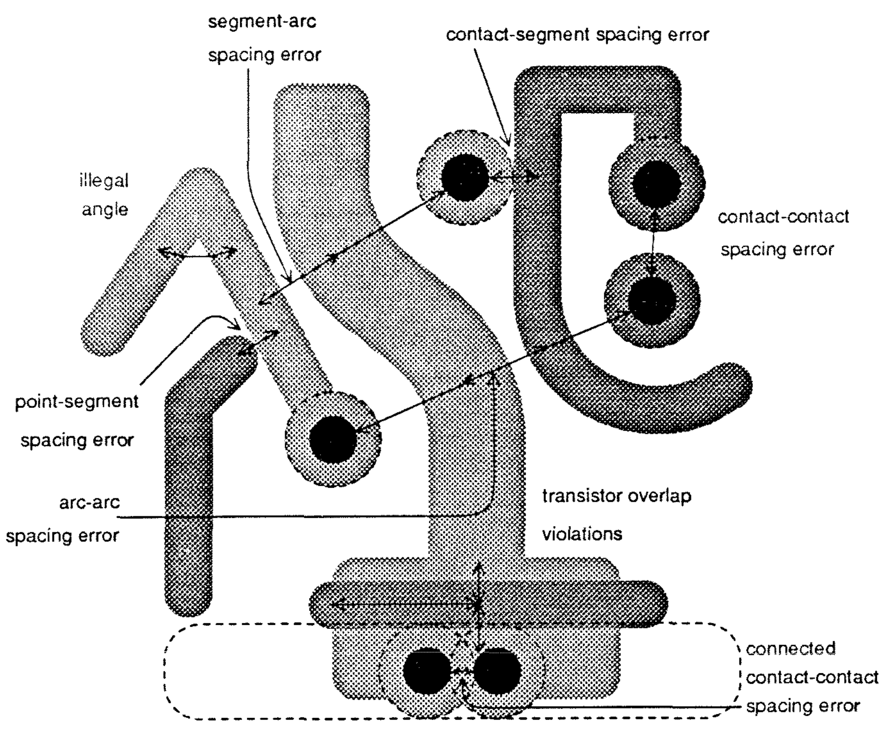

segment-arc spacing error
contact-segment spacing error
illegal angle
contact-contact spacing error
point-segment spacing error
arc-arc spacing error
transistor overlap violations
connected contact-contact spacing error

Figure 2-8: Analysis Rules

The distance from each point to surrounding connected and unconnected points and lines is compared to the minimum spacing rules. If the point is a transistor, then the four closest points are checked for valid overlap. Often the transistor overlap is GDR correct by construction, but existing points may violate transistor overlap rules and thus must be checked. If the point is a contact, then the closest connected contacts are checked for valid connected contact spacing. Finally the point's distance from surrounding segments and points is calculated and compared to both the minimum point to point (contact–contact) and point to path (contact–interconnect and contact–transistor) distances.

Pooh checks to ensure that the distance from each segment to surrounding points and segments is greater than or equal to the minimum GDR spacing rules. First, it marks all directly connected segments, ensuring that connected path–path spacing rules are met. Next, it checks that this segment's distance from surrounding contacts is greater than the appropriate point–path spacing rule. It then checks that this segment's distance from the two end points of all unmarked surrounding segments is greater than or equal to the appropriate path to path spacing rule, and that the segments do not cross. If either of the two segments contain non-zero arcs, the arc–arc or line–arc distance is calculated and compared to the spacing rule. Finally, for each of this segment's connection points, Pooh checks the angle between this segment and the point's other connecting segments. The angle between two segments that connect to the same point must be greater than or equal to the minimum path to path angle.

The analysis of surround paths is similar to that of other paths with a single exception: distances are signed. If points and segments occur inside the surround path, with a negative distance, then the distance must be greater than or equal to the inside GDR spacing rule. Otherwise, the points and segments occur outside the surround path, with a positive distance, and the distance must be greater than or equal to the outside GDR spacing rule.

2.6 Simulation Interface

Simulation is a necessary and important part of the verification of a design. The GDR algorithms guarantee design rule correct circuits, and the typed ports provide a mechanism for statically checking the composition of signals. But, it is still necessary to simulate a circuit, and have some degree of assurance that the geometry is equivalent to the simulated circuit. Simulation input is easy to generate from the same Pooh representation from which geometry is generated.

Pooh supports three different kinds of simulation—depending on the level of detail a designer is interested in. The three types are:

1) an analog device simulator, such as SPICE [Nagel 75],
2) a logical switch level simulator such as Mossim [Bryant 82], and
3) a dynamic timing simulator such as the one developed by Tzu-Mu Lin, at Caltech [Lin 84].

The information necessary to interface to these various simulators is easily derived, since Pooh maintains a list of the transistors, along with their types and strengths, and the node numbers of all the interconnection wires, along with their sizes.

2.7 Geometry Generation

At the completion of a design, a geometrical description may be generated from the Pooh representation. Since Pooh enforces GDRs, a designer may use a strict Pooh representation up until the time of mask description for fabrication. Therefore geometry generation is a very small part of the computation necessary to complete a design.

For each technology, Pooh maintains the information needed to go from the Pooh representation to geometry. This information is:

1) for each logical layer and transistor type the geometrical layers and the overlap rules between the geometrical layers, and
2) for each connection point a reference to the geometrical object that implements this point.

Points are modeled as fixed geometrical objects. Paths are modeled as algorithmic geometrical objects and therefore each path generates a unique set of polygons. This information is kept in the technology file along with the design rule data.

Often, fabrication houses or silicon foundries find it necessary to modify or process geometry files for a particular technology. The processing often includes bloats, shrinks and scaling. Shrinking geometry while maintaining the electrical connectivity is a very difficult problem for general polygons and wires. Since the centers-lines of Pooh paths are connected, called skeletal connectivity, it is equally easy to perform bloats, shrinks and scaling of designs generated by Pooh.

3. Synthesis Algebra

This chapter describes how we may construct each path such that it meets the GDR constraints. The computational complexity of calculations between paths is minimized by the choice of elementary structures: points, lines and arcs. Points are represented as circles. Lines are represented by their directed and normalized line equation. Interconnection, transistors, ports and connection points are composed from these elements.

In the following section, let P_k denote a Pooh path, where a Pooh circuit is a set of paths $\mathcal{P}$ and $P_k \in \mathcal{P}$. Each path includes an initial point p_0 and a set of m segments $S_k \equiv \{s_1, s_2, \ldots s_m\}$. The placement of each path segment s_i is based on the two points p_{i-1} and p_i and both the segment's arc radius r_i, and the last segment's arc radius r_{i-1}. Given a point p_j, let x_j, y_j denote the point's x-coordinate and y-coordinate respectively.

3.1 Primitive Representations

Circles

A unit circle is a convenient way of representing both points and arcs, of which larger constructs may be composed. A circle allows calculations to be performed between lines and points without introducing polygonal edges. Mask geometry cannot, however, be created using perfect circles—therefore Pooh uses an n-point circle approximation. The polygonal approximation and its effect on the calculations presented in this chapter are discussed in detail in Chapter 7. The next section assumes perfect circles.

Lines

The line segment portion of a segment s_i is represented by the normalized line equation:

$$L_i(x,y) = A_i x + B_i y + C_i \qquad (3\text{-}1)$$

$$\text{where } A_i^2 + B_i^2 = 1$$

$$\text{and, for a point } x, y \text{ on the line:}$$

$$L_i(x,y) = 0.$$

In Figure 3-1, the point p_2 is the end point of the segment s if the segment radius is zero. Otherwise p_2 is the starting point of the segment's arc. This figure illustrates the line calculation for the line $L(x,y)$:

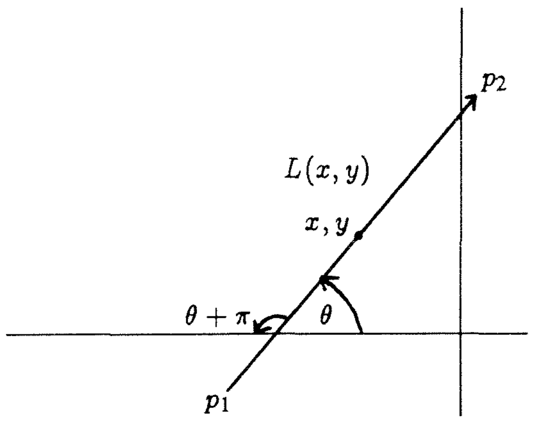


Figure 3-1: Line Calculation

Given two points p_1 and p_2, the normalized coefficients A, B, and C are computed as follows:

$$\begin{aligned} Len &= \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \\ A &= -\sin\theta = (y_1 - y_2)/Len \\ B &= \cos\theta = (x_2 - x_1)/Len \\ C &= -Ax_1 - By_1. \end{aligned} \tag{3-2}$$

The slope of this line is $-A/B$ or $\tan\theta$. An equivalent line $L'(x,y)$, in the opposite direction (from p_2 to p_1), is computed based on the angle $\theta' = \theta + \pi$:

$$\begin{aligned} \sin\theta' &= \sin(\theta+\pi) = -\sin\theta & A' &= -A \\ \cos\theta' &= \cos(\theta+\pi) = -\cos\theta & B' &= -B \\ & & C' &= -C. \end{aligned}$$

The distance D between the line equation $L_i(x,y)$ and a point p_q is found by [Sutherland 78]:

$$D = A_i x_q + B_i y_q + C_i \tag{3-3}$$

where the sign of D indicates the direction of the point with respect to the line. The Pooh line equation was chosen such that a negative D indicates the point is on the right hand side of the line and a positive D indicates the point is on the left hand side of the line.

The intersection point p_{int} between the two lines $L_1(x,y)$ and $L_2(x,y)$ is found as follows [Sutherland 78]:

$$\begin{aligned} \det &= B_1A_2 - A_1B_2 \\ p_{int} &= \Big((B_2C_1 - B_1C_2)/\det,\ (A_1C_2 - A_2C_1)/\det\Big). \end{aligned} \tag{3-4}$$

Arcs

Arcs, like line segments, are directional and the sign of the arc radius r_i indicates its direction. A positive radius indicates a counter-clockwise direction, and a negative radius indicates a clockwise direction. In the following section, let $\mathcal{A}_i$ denote the arc associated with the segment s_i, a_i^b be the arc's starting point, and a_i^e be the arc's ending point.

An arc $\mathcal{A}_i$ is determined by three points p_{i-1}, p_i, p_{i+1}, and three radii r_{i-1}, r_i, r_{i+1}. The pair p_{i-1}, r_{i-1} determines the starting point of the arc a_i^b, the pair p_i, r_i indicates the placement of $\mathcal{A}_i$, and the pair p_{i+1}, r_{i+1} determines the ending point of the arc, a_i^e. The following equations present the calculations necessary for finding the line segment $L_i(x,y)$,that joins the arcs $\mathcal{A}_{i-1}$ and $\mathcal{A}_i$, with $r_i > 0$ and $r_{i-1} < 0$. From there, we may generalize the calculation to all signed radii. Figure 3-2 illustrates this calculation.

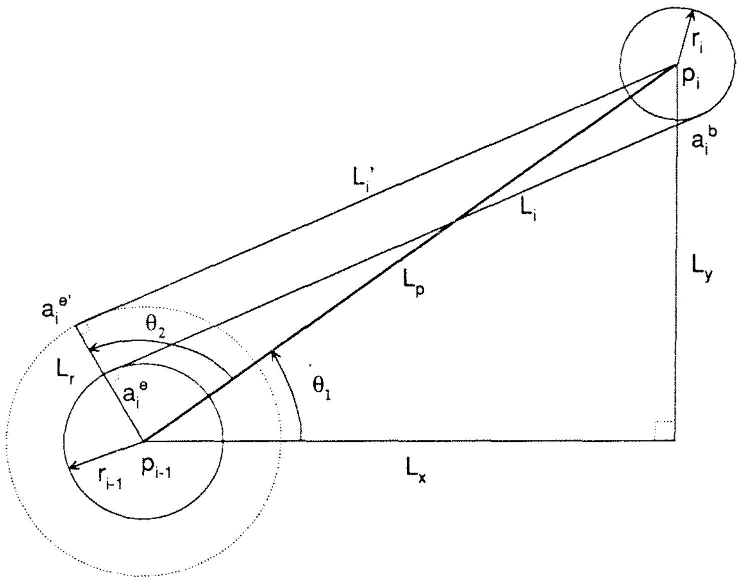


Figure 3-2: Arc Calculation

$$
\begin{aligned}
|L_r| &= r_i - r_{i-1} \\
|L_x| &= x_i - x_{i-1} \\
|L_y| &= y_i - y_{i-1} \\
|L_p| &= \sqrt{|L_x|^2 + |L_y|^2} \\
|L_i'| &= \sqrt{|L_p|^2 - |L_r|^2}
\end{aligned}
\qquad
\begin{aligned}
L_i' &\parallel L_i \\
L_r &\perp L_i
\end{aligned}
$$

$$
\begin{aligned}
\theta' &= \theta_1 + \theta_2 \\
\cos_i(\theta') &= \frac{|L_x||L_r| - |L_y||L_i'|}{|L_p|^2} \\
\sin_i(\theta') &= \frac{|L_y||L_r| + |L_x||L_i'|}{|L_p|^2}
\end{aligned}
$$

The arc starting point $a_i^b = (x_i - r_i \cos_i(\theta'),\ y_i - r_i \sin_i(\theta'))$. Let us define a point on the unit circle:

$$p_i^c = (-\cos_i(\theta'),\ -\sin_i(\theta')) \,. \tag{3-5}$$

Then, the arc beginning point $a_i^b = (x_i + r_i x_i^c,\ y_i + r_i y_i^c) = p_i + r_i p_i^c$ and the last arc's ending point $a_{i-1}^e = p_i - |r_{i-1}| p_i^c$. But, $r_{i-1} < 0$ by definition, therefore $a_{i-1}^e = p_i + r_{i-1} p_i^c$. In general, we may compute the tangent point p_i^c on the unit circle, and then use the sign of the radii to determine the ending point of $\mathcal{A}_{i-1}$, and the beginning point of $\mathcal{A}_i$. The ending point of $\mathcal{A}_i$ is found while calculating the starting point of $\mathcal{A}_{i+1}$. The calculation is:

$$\begin{aligned} a_i^b &= p_i + r_i p_i^c \\ a_{i-1}^e &= p_{i-1} + r_{i-1} p_i^c . \end{aligned} \tag{3-6}$$

The arc representation is $\mathcal{A}_i = (r_i,\ p_i,\ p_i^c,\ \mathcal{A}_{i+1})$ from which we may easily compute a_i^b and a_i^e from (3-6). An illustration of a positive and a negative arc, based on the same unit circle points p_0 and p_1, is shown in Figure 3-3. In practice, the points on the unit circle are chosen from a set of pre-calculated points, as is discussed in Chapter 7.

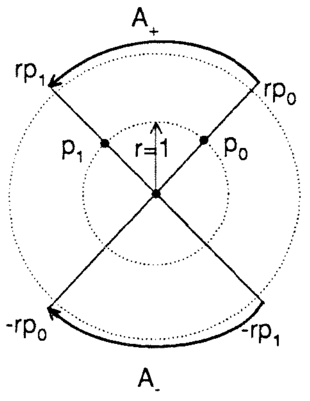


Figure 3-3: Arc Representation

3.2 Path Synthesis

Pooh constructs a path P_k "correct" according to the GDRs. The following definitions of GDRs are introduced:

$$\begin{aligned} mw(T_k) &\equiv \textit{minimum width of a path of type } T_k \\ mr(T_k) &\equiv mw(T_k)/2 \\ minSpace(s_i, s_j) &\equiv \textit{minimum spacing between paths of type } T_k \textit{ and } T_l \\ &\quad\ \textit{where } s_i \in S_k \textit{ and } s_j \in S_l \\ PminSpace(s_i, p_q) &\equiv \textit{minimum spacing between a path of type } T_k \textit{ where } s_i \in S_k \\ &\quad\ \textit{and the point } p_q . \end{aligned}$$

A path P_k is defined by a path type T_k, a width W_k, where $W_k \geq mw(T_k)$, an initial point p_0, and a set of m segments S_k. Each of the path segments is synthesized in sequence to construct a valid path. Each path segment has four points of possible interest. These points are the first and last points of the segment, and the first and last points of the segment's line segment. Let us define four functions to aid in the description of a path segment: $lp(s_j)$ for the last (previous) point of the segment s_j, $lr(s_j)$ for the last radius of the segment s_j, $bp(s_j)$ for the starting point of the line segment $L_j(x,y)$, and $ep(s_j)$ for the ending point of the line segment, as follows:

$$lp(s_j) = \begin{cases} p_0, & j = 1 \\ p_{j-1}, & \text{otherwise.} \end{cases}$$

$$lr(s_j) = \begin{cases} 0, & j = 1 \\ r_{j-1}, & \text{otherwise.} \end{cases}$$

$$bp(s_j) = \begin{cases} lp(s_j), & lr(s_j) = 0 \\ a^e_{j-1}, & \text{otherwise.} \end{cases}$$

$$ep(s_j) = \begin{cases} p_j, & r_j = 0 \\ a^b_j, & \text{otherwise.} \end{cases}$$

A minimum width path P_k is defined by the locus of points exactly half the minimum width, or $mr(T_k)$, distance away from the path center-line. A path P_l whose width is greater than the minimum width is defined by the locus of points exactly $mr(T_l)$ distance away from the path skeleton. This skeleton is the path center-line expanded by the width W_l minus the minimum radius.

A minimum width path segment s_i where $s_i \in S_k$ of the path P_k is shown in Figure 3-4. Notice that the starting point bp(s_i) is surrounded by a half circle whose radius is $mr(T_k)$. In fact, the path P_k is defined by the locus of points exactly $mr(T_k)$ distance away from the path center-line. This path P_k is defined as a set of m segments, $r_m = 0$, where each segment consists of a composition of an arc and a line identical to the one shown in Figure 3-4.

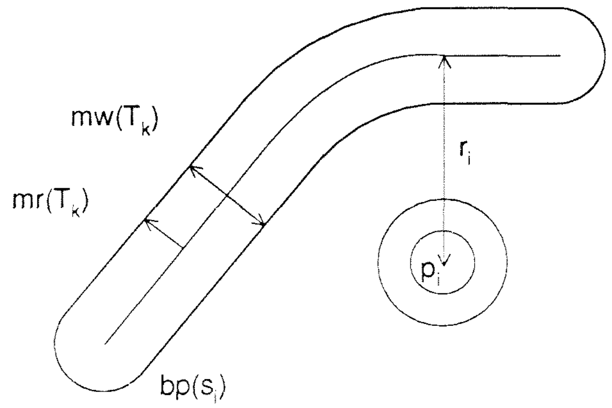


Figure 3-4: Minimum Width Segment

A path P_l where $W_l > mw(T_l)$ is defined by the locus of points exactly $mr(T_l)$ distance away from the path skeleton. The path skeleton surrounds the center-line by a distance $d = W_l/2 - mr(T_l)$ as shown in Figure 3-5. The line segment portion of each segment $s_i \in S_l$ is defined as three parallel lines $L_i(x,y)$, $L_i'(x,y)$ and $L_i''(x,y)$ where $C_i' = C_i + d$, and $C_i'' = C_i - d$. Each path segment arc is identical to the arc of a minimum width path. The line $L_{i\perp}(x,y) = -B_i x + A_i y$. The calculation of a wide path's end conditions at the points p_0 and p_m, is based on $L_1(x,y)$, $L_1'(x,y)$, $L_1''(x,y)$, $L_{1\perp}(x,y)$, and $L_m(x,y)$, $L_m'(x,y)$, $L_m''(x,y)$, $L_{m\perp}(x,y)$, respectively. The offsets from p_0 to the two points p_l and p_r, as shown in Figure 3-5, are:

$$\begin{aligned} p_l &= (A_1 d,\ B_1 d) \\ p_r &= (-A_1 d,\ -B_1 d). \end{aligned} \tag{3-7}$$

from equation (3-4). The point offsets from p_m are similarly calculated.

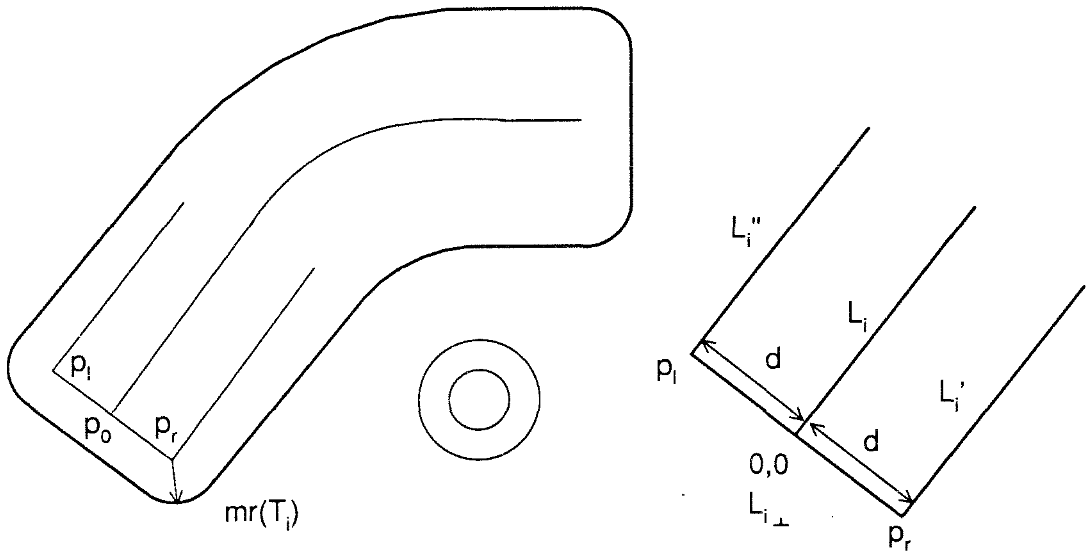


Figure 3-5: Wide Path Segment

Segment Synthesis

Each segment s_i is initially designated by a point p_i, a direction $q_i = \pm 1$, and a connection status. Let $ValidConnect(s_j, p_q)$ denote a function that indicates whether or not there is a valid connection between s_j and p_q. If s_i connects to the point p_i and if $validConnect(s_i, p_i)$, then r_i is the design rule correct connection radius, often equal to zero. Otherwise, if the segment does not connect, r_i is calculated based on the design rules. If other segments have not previously "missed" the point p_i, then r_i is equal to $PminSpace(s_i, p_i)$. Otherwise we must detect interference between this arc $\mathcal{A}_i$ and all other arcs $\mathcal{A}_l$ where $p_l \equiv p_i$, as illustrated in Figure 3-6. In order to detect arc interference, $\mathcal{A}_i$ must be defined, which in turn implies that $r_i \neq 0$ to use equation (3-6). Therefore, initially we let $r_i = q_i$.

Let $U(\mathcal{A}_q)$ denote a unit arc defined as:

$$U(A_q) = (p^c_q,\ p^c_{q+1},\ dir)$$

$$dir = \begin{cases} 1, & r_q \geq 0 \\ -1, & r_q < 0. \end{cases}$$

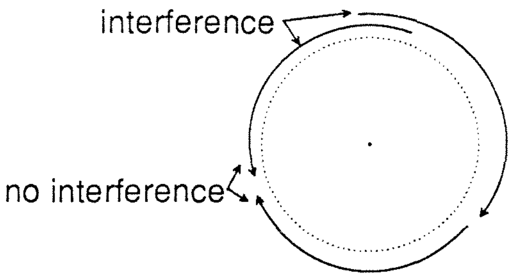


Figure 3-6: Arc Interference

These functions detect interference between two unit arcs U_1 and U_2:

```
define in($U_1$,$U_2$):  boolean
    arc ranges overlap
enddef
define otherhalf($U_1$):  $U$
    $p^c_q \leftarrow -p^c_q$;  $p^c_{q+1} \leftarrow -p^c_{q+1}$;
enddef
define reverse($U_1$):  $U$
    $dir \leftarrow -dir$;
enddef
define interfere($U_1$,$U_2$):  boolean
    interfere $\leftarrow$ if $dir_1 > 0$ and $dir_2 > 0$ then in($U_1$,$U_2$)
            ef $dir_1 > 0$ and $dir_2 < 0$ then in($U_1$,otherhalf(reverse($U_2$)))
            ef $dir_1 < 0$ and $dir_2 > 0$ then in(otherhalf(reverse($U_1$)), $U_2$)
            else in(reverse($U_1$), reverse($U_2$))
            fi
enddef
```

The detection of arc overlap is simplified when defined in terms of the underlying circle approximation. Therefore, the definition of the function in is postponed until the circle approximation is described. Chapter 7 includes the definition of in.

If we assume that the segments that depend on a point p_q are maintained in a list sorted by increasing arc radius $|r|$, then the following algorithm finds the minimum design rule correct "miss" distance between s_q and p_q:

```
define minimum_distance($s_q$, $p_q$):  distance
    var min_dis :  distance;
    min_dis ← $PminSpace(s_q, p_q)$;
    while there is another segment missing $p_q$ do
        $s'$ ← next segment;
        if interfere($U(s_q)$, $U(s')$) then
            if $minSpace(s_q, s') > 0$ then
                min_dis ← $|r'| + minSpace(s_q, s')$
            fi
        fi
    od
    minimum_distance ← if $r_q < 0$ then -min_dis else min_dis fi
enddef
```

Once the segment arc radius r_i is calculated using minimum_distance, A_i is defined by (3-6), and $L_i(x, y)$ is calculated using (3-1). Given this complete segment definition, it is possible to detect and prevent an invalid segment. It is possible to indicate a miss direction such that a segment loops, as shown in Figure 3-7, which is never desirable. The dot product between the line $L_i(x, y)$ and the line connecting a_i^b and a_i^e, shown in the figure as L_c, detects this condition between two consecutive segments. If the segment is found to be invalid, Pooh replaces the two segments by a single segment as shown in the figure.

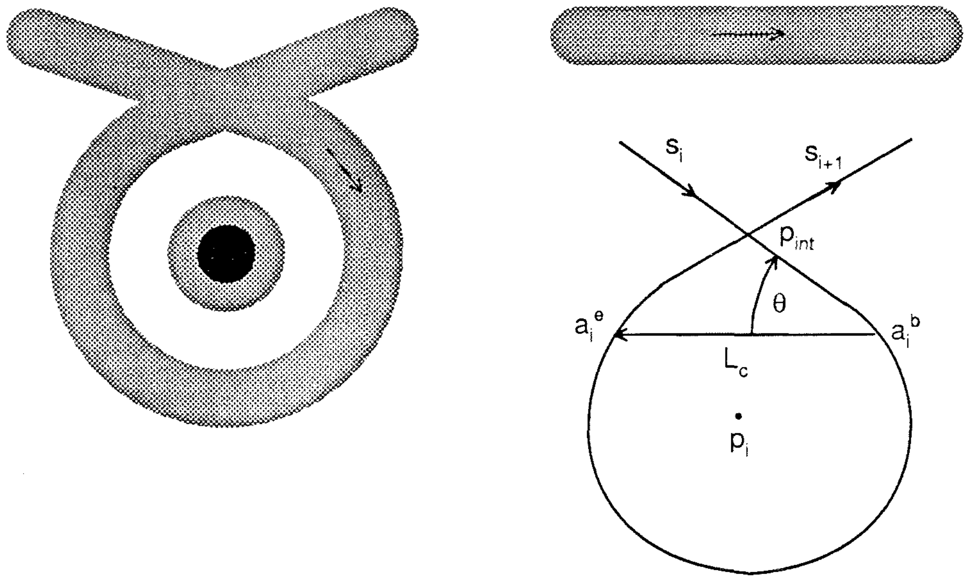


Figure 3-7: An Illegal Segment

The point a_i^b represents the tangent point of the segment s_i on the circle of radius $|r_i|$ centered at p_i, and the point a_i^e represents the tangent point of the segment s_{i+1} on the same circle. In order to prevent these two line segments from crossing, the angle θ

between s_i and the line connecting a_i^b and a_i^e must be greater than or equal to 90°. If $-90° < \theta < 90°$ then s_i is ignored.

$$Dot = s_i \cdot L_c = \cos\theta = A_i A_c + B_i B_c$$

The segment construction rules described thus far allow us to define any legal interconnection path. The rules must be extended to incorporate transistor paths.

Transistor Synthesis

With respect to synthesis algorithms, transistor paths differ from interconnection paths in two ways: (1) the presence of two logical layers—channel and gate, and (2) a Length/Width parameter that determines when one of the logical layers ends. A transistor path is composed of the center-line layer and the outside layer, one of which is the channel, the other the gate. If the center-line layer is the gate, then the outside layer is the channel and the path is a "wide" transistor, otherwise the path is a "long" transistor, as shown in Figure 3-8. Let us call these two layers $Center(T_k)$ and $Outside(T_k)$ for a transistor of type T_k. Let $\mathcal{L}_k$ denote the Length/Width parameter that determines when the $Outside(T_k)$ layer ends. The path width W_k governs the width of the center-line layer as shown in Figure 3-8. The parameter $\mathcal{L}_k$ governs the length of the outside layer $Outside(T_k)$. There is a set of overlap design rules for each transistor type that dictate the distance each of the logical layers must overlap the gate region. The two definitions for the overlap rules are $OutsideOver(T_k)$ for the outside overlap, and $CenterOver(T_k)$ for the center-line overlap.

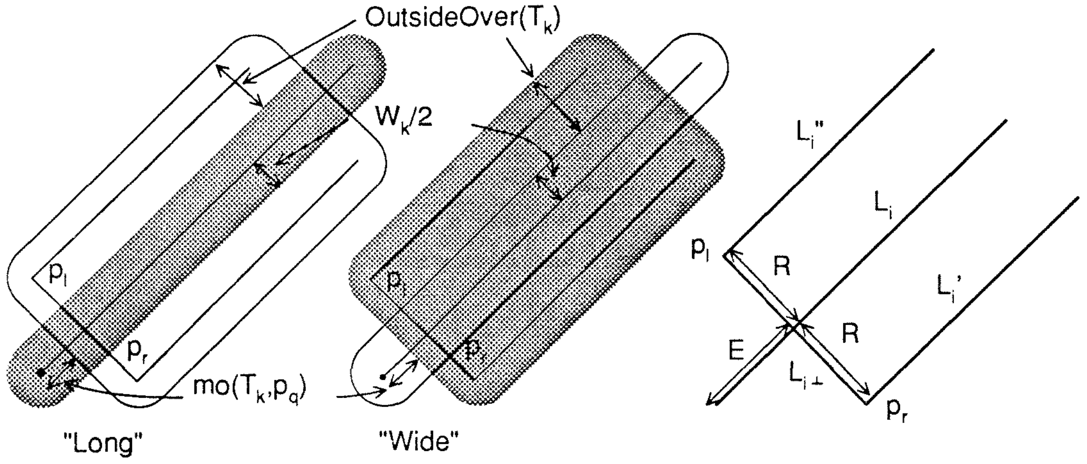


Figure 3-8: Transistor Overlap

Figure 3-8 illustrates the beginning of a transistor path. We must calculate the point offsets p_l and p_r such that the $OutsideOver(T_k)$ and $CenterOver(T_k)$ rules are satisfied. The lines $L_i(x,y)$, $L_i'(x,y)$, and $L_i''(x,y)$ are the same lines used in Figure

3-5. The line $L_{i\perp}(x, y) = -B_i x + A_i y + E$. Let the minimum overlap be defined as:

$$mo(T_k,\ p_q) = \Big(CenterOver(T_k) - mr(Center(T_k))\Big)\ \max\ \Big(PminSpace(Outside(T_k),\ p_q)\Big).$$

The two point offsets p_l and p_r are calculated (from equation (3-4)) by:

$$\begin{aligned} E &= mo(T_k,\ p_q) + mr(Outside(T_k)) \\ R &= W_k/2 + OutsideOver(T_k) - mr(Outside(T_k)) \\ p_l &= (A_i R + B_i E,\ B_i R - A_i E) \\ p_r &= (-A_i R + B_i E,\ -B_i R - A_i E). \end{aligned} \tag{3-8}$$

This equation allows us to calculate the beginning of a transistor path. We must also end the transistor region. Pooh uses the parameter $\mathcal{L}_k$ to determine the segment s_n, such that $1 \le n \le m$, and on which the center-line length achieves $\mathcal{L}_k$.

The parameter $\mathcal{L}_k$ governs the size of the transistor along the center-line. If $Center(T_k)$ is the channel then the parameter is the transistor length, and W_k is the transistor width. If $Center(T_k)$ is the gate then the parameter is the transistor width and W_k is the transistor length. Given that the length is less than the total path length, or $\mathcal{L}_k + mo(T_k, p_0) + mo(T_k, ep(s_n)) \le \sum_{i=1}^{m}(|L_i(x,y)| + arclen(\mathcal{A}_i))$, then we must find the segment s_n, such that:

$$\begin{aligned} &\sum_{i=1}^{n-1}(|L_i(x,y)| + arclen(\mathcal{A}_i)) \\ &\le \mathcal{L}_k + mo(T_k, p_0) + mo(T_k, ep(s_n)) \\ &\le \sum_{i=1}^{n}|L_i(x,y)| + \sum_{i=1}^{n-1} arclen(\mathcal{A}_i). \end{aligned}$$

This condition assumes that $\mathcal{L}_k$ is achieved on the line segment portion of a path segment. A segment that meets these constraints may not exist, if the length is achieved on the arc portion of a segment, which implies that for some q:

$$\begin{aligned} &\sum_{i=1}^{q}|L_i(x,y)| + \sum_{i=1}^{q-1} arclen(\mathcal{A}_i) \\ &\le \mathcal{L}_k + mo(T_k, p_0) + mo(T_k, ep(s_q)) \\ &\le \sum_{i=1}^{q}(|L_i(x,y)| + arclen(\mathcal{A}_i)). \end{aligned}$$

If this second condition is true, then we insert a segment s' that approximates the arc $\mathcal{A}_q$ at the position that $\mathcal{L}_k$ dictates between the segments s_q and s_{q+1}. Then, $\mathcal{L}_k$ is achieved on the segment s' by design, and thus the segment $s_n = s'$.

We may use the segment s_n to determine the two point offsets p_l and p_r that yield a transistor path that meets the specified $\mathcal{L}_k$ constraint. We let

$$E = \sum_{i=1}^{n}(|L_i(x,y)|) + \sum_{i=1}^{n-1} arclen(\mathcal{A}_i) - \mathcal{L}_k - mo(T_k, p_0) + mr(Outside(T_k)),$$

and then compute the points using equation (3-8). The transistor path continues in the center-line logical layer as if it were an interconnection path of type $Center(T_k)$.

Surround Path Synthesis

The synthesis of a surround path P_k differs from other paths in three ways: (1) the path width is always twice the minimum radius, or $W_k = 2 \times mr(T_k)$, (2) the first and last points of the path are the same, or for a path P_k with m segments, $p_0 = p_m$, and (3) the sign of the segment arc radius r_j, for $s_j \in \mathcal{S}_k$, indicates whether the point p_j is inside or outside the path. Surround paths define entire regions and are clockwise paths by construction. Pooh enforces these three restrictions, and otherwise constructs surround paths in the usual manner.

3.3 Point Calculations

The placement of a point determines the placement of one or more path segments, but it may also be determined by one or two line segments using the line intersection equation (3-4). It is possible to designate a point as placed at the intersection of two line segments. The placement of such a point may be computed directly from (3-4). It is even possible to designate a segment s_i as placed perpendicular to segment s_j, where s_i and s_j are in general not members of the same path's set of segments.

Given the segments s_i, s_j and p_1=bp(s_i), we may calculate the point p_i, with r_i=0, ensuring that $s_i \perp s_j$. The point p_i is either at the intersection of s_i and s_j, as shown in Figure 3-9, or there is another point p_q at the intersection of s_i and s_j and p_i is some distance Ext past p_q, as shown in Figure 3-9. This latter type of calculation is used to construct a point that meets the transistor overlap rules, given the transistor point p_q. The calculation to find the point p_i at the intersection of s_i and s_j is:

$$\begin{aligned} L_i(x,y) &= -B_j x + A_j y + C_i \\ C_i &= B_j x_1 - A_j y_1 \\ p_i &= (B_j C_i - A_j C_j,\ -A_j C_i - B_j C_j) \end{aligned} \qquad (3\text{-}9)$$

from equation (3-4). Equation (3-9) may be extended to calculate a point beyond the intersection of s_i and s_j. If s_i is of type T_k, and the transistor point p_q, at the intersection of s_i and s_j, is of type T_l, then the calculation is:

$$\begin{aligned} Ext &= \textit{if}\ T_n = Center(T_t)\ \textit{then}\ CenterOver(T_t)\ \textit{else}\ OutsideOver(T_t)\ \textit{fi} \\ p_i &= (B_j C_i - A_j(C_j + Ext),\ -A_j C_i - B_j(C_j + Ext)) . \end{aligned} \qquad (3\text{-}10)$$

Contacts may or may not be symmetrical, according to the target set of GDRs. If a contact type is asymmetrical, then the point's orientation is important. If the GDRs

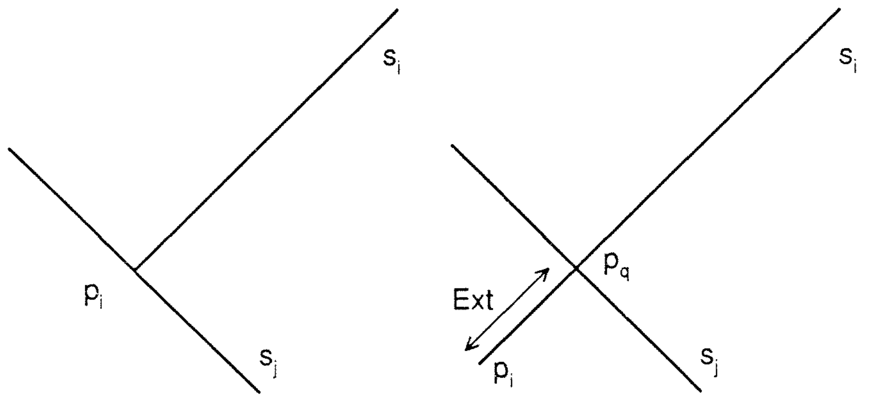


Figure 3-9: *Intersection Points*

dictate that the connection angle of a path type is fixed with respect to a contact, then Pooh orients the contact according to the GDRs. If more than one segment with a fixed connection angle connects to the same contact, then additional points, with different orientations are superimposed. Figure 2-6 illustrates oriented contacts.

The orientation angle φ of the contact point p_q is easy to compute, given the segment s_i of type T_k. If the point p_q is a contact of type c_q, then let the function $fixedAngle(c_q, T_k)$ denote the GDR specified connection angle of c_q with respect to s_i. Then:

$$\theta = \arccos(B_i) \qquad \text{from (3-2)}$$
$$\varphi = \theta + fixedAngle(c_q, T_k).$$

In this chapter I have described how, given a minimal amount of information, we may construct each path such that it meets the design rule constraints. This section has alluded to the circuit connectivity several times. The next chapter discusses synthesizing the connectivity of a collection of paths. I will then examine the calculations necessary to detect problems between paths.

4. Node Synthesis

The Pooh representation must express the connectivity of the transistors and their interconnection. There are two aspects to this problem. First, the description must contain sufficient information to provide the possibility of detecting whether or not two elements are connected. An element, in this context, is either a path or a point. Second, given any two elements within a Pooh circuit, there must be a quick way of detecting their underlying connectivity.

The first condition affects the composition of Pooh paths and points. The underlying philosophy is that any circuit level representation must express the designer's intent [McGrath 80]. In Pooh, any two elements that are connected share at least one reference to the same point. That is, connectivity is represented explicitly in the representation.

A simple example of shared references is shown in Figure 4-1. Two metal paths, P_1 and P_2, each consisting of a single segment, are shown in the figure. These two metal paths are obviously connected since at one point they cross. The Pooh composition rules state that P_1 is connected to P_2 if and only if there exists a point p_q such that p_q is contained in both the P_1 and P_2 path definitions. Otherwise P_1 is not connected to P_2 and the Pooh analysis rules (described in Chapter 5) detect a spacing violation between P_1 and P_2.

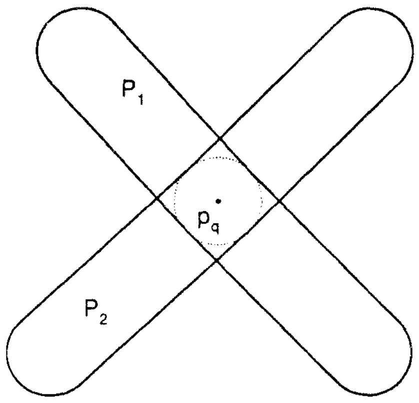


Figure 4-1: Connected Paths

The second condition mandates that there is a quick algorithm for determining whether or not any two elements are connected. Pooh uses Martin Rem's Equivalence class algorithm [Dijkstra 76] to note the elements' connectivity. Pooh elements are separated into equivalence classes using this algorithm, and then connectivity is detected by checking whether two elements are in the same equivalence class.

4.1 Node Equivalence

Pooh assigns each element a unique increasing positive node number, and then uses Rem's Algorithm to record connected elements. For each collection of paths there is an array `nodenumbers` that indicates the connectivity. Two elements with node numbers N_1 and N_2 are connected if and only if `nodenumbers[`N_1`]=nodenumbers[`N_2`]`. Given the array `nodenumbers`, the algorithm for asserting connectivity can be expressed:

```
define equivalence($N_1$, $N_2$ : integer)
    var a1, a2 : integer;
    a1 $\leftarrow$ nodenumbers[$N_1$];
    a2 $\leftarrow$ nodenumbers[$N_2$];
    while a1 $\neq$ a2 do
        if a2 < a1 then
            nodenumbers[$N_1$] $\leftarrow$ a2;
            $N_1$ $\leftarrow$ a1;
            a1 $\leftarrow$ nodenumbers[a1];
        else
            nodenumbers[$N_2$] $\leftarrow$ a1;
            $N_2$ $\leftarrow$ a2;
            a2 $\leftarrow$ nodenumbers[a2];
        fi
    od
enddef
```

The algorithm works by equating both node numbers N_1 and N_2, two nodes of a tree, to the smallest integer k reachable from either N_1 or N_2. Initially all node numbers are in separate equivalence classes, represented as trees of numbers. Then, each time `equivalence(`N_1`, `N_2`)` is invoked, a new tree is implicitly created that includes both N_1 and N_2. If there are n node numbers, then the longest tree's path length may be n, in which case the worst case cost of executing n `equivalence` statements is $O(n^2)$. But, since each time `equivalence` is invoked, it flattens out the reachable tree, the expected height of any given tree remains less than $\log(n)$, in which case the cost of executing n equivalences is $O(n \log(n))$.

Any transistor, either a path or point type, is assigned three node numbers: source, drain and gate. Each interconnection path and connection point is assigned a single node number. Assuming that each time two elements are connected, `equivalence` of the two connecting node numbers N_1 and N_2 is invoked, then detecting whether or not any two elements are connected is simply whether or not `nodenumbers[`N_1`]` = `nodenumbers[`N_2`]`.

4.2 Path Connectivity Representation

This section describes the construction of a set of Pooh paths such that they fully maintain the circuit connectivity. It uses the notation introduced in Chapter 3. A Pooh circuit consists of a set of paths $\mathcal{P} \equiv \{P_1, P_2, \ldots P_z\}$, and a set of points $\mathbf{p} \equiv \{p_1, p_2, \ldots p_r\}$. Each transistor, either a path P_k or a point p_k, includes three node numbers $source_k$, $drain_k$, and $gate_k$. Each interconnect path P_l or connection point p_l includes a single node number $node_l$. Let $transistor(P_k)$ denote a function that indicates whether or not the path P_k is a transistor path, and let $ptran(p_q)$ denote a function that indicates whether or not a point p_q is a transistor point.

In the following discussion, a path segment is designated for convenience as $s_{k,j}$. The notation $s_{k,j}$ is shorthand for the j^{th} segment of the path P_k, or $s_j \in S_k$ in the path P_k.

Each path segment maintains a set of points $\mathbf{p}_{k,j} \equiv \{p_1, p_2, \ldots p_n\}$, and a connection status for the points $\mathbf{cs}_{k,j} \equiv \{cs_1, cs_2, \ldots cs_n\}$. The points are maintained strictly for connectivity reasons, with the exception of p_n, that also determines the segment's placement. The point p_n may or may not be connected, all other points p_j, $j < n$, are connected by construction. This point $p_n \equiv p_j$ on segment $s_j \in S_k$ associated with the path P_k, described in the last chapter. Each connection status entry $cs_i \in \{NoConnect, Connect, CSource, CDrain, PSource, PDrain\}$. The first four connection status values denote the following: not connected, connected, connected to the source of a transistor point, and connected to the drain of a transistor point, respectively. The last two status values denote connecting a point to the source of a transistor path and connecting a point to the drain of a transistor path. The Pooh construction rules restrict the exact connection status values a particular point may have associated with it.

Pooh constructs an interconnect path P_k with a set of m segments S_k as a single node. This restriction implies that the connection status cs_q of a point p_q, where $p_q \in \mathbf{p}_k$, may be equal to $CSource$ or $CDrain$ only if p_q is either the first point or last point of the path and if p_q is a transistor point. Each segment $s_{k,j}$ may "miss" its last point p_j in which case the connection status is $NoConnect$. All other connected points may be connection points or transistor points where the segment connects to the gate of the transistor. In either case, the connection status is $Connect$. Figure 4-2 illustrates an nMOS interconnection path P_1, with three segments. In the figure, dotted lines indicate connected path segments that are not elements of P_1, circles indicate connection points, and crosses indicate transistor points.

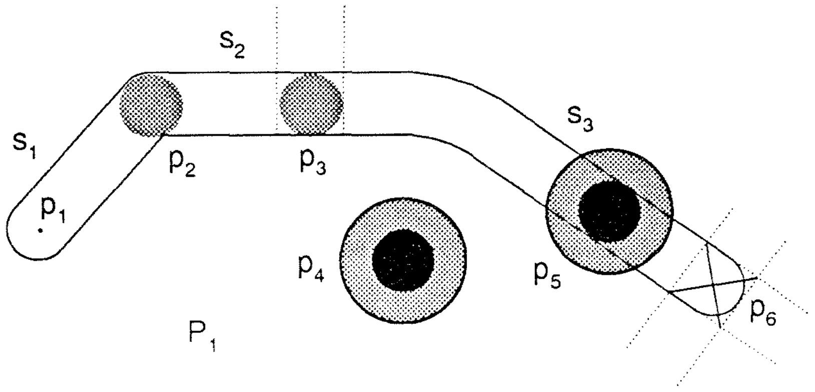


Figure 4-2: Interconnection Path

In this path, the segments, points and connection status have the following values:

$$S_1 = \{s_1, s_2, s_3\}$$

$$\mathbf{p}_{1,1} = \{p_1, p_2\}$$
$$\mathbf{cs}_{1,1} = \{Connect, Connect\}$$
$$\mathbf{p}_{1,2} = \{p_3, p_4\}$$
$$\mathbf{cs}_{1,2} = \{Connect, NoConnect\}$$
$$\mathbf{p}_{1,3} = \{p_5, p_6\}$$
$$\mathbf{cs}_{1,3} = \{Connect, CSource\}$$

A transistor path P_l of type T_l includes three nodes—$source_l$, $drain_l$, and $gate_l$. The connectivity of the transistor depends on the center layer $Center(T_l)$. If $Center(T_l)$ is the gate layer then possible valid connection regions are shown on the left side of Figure 4-3. Otherwise $Center(T_l)$ is the channel, and the possible valid connection regions are shown on the right side.

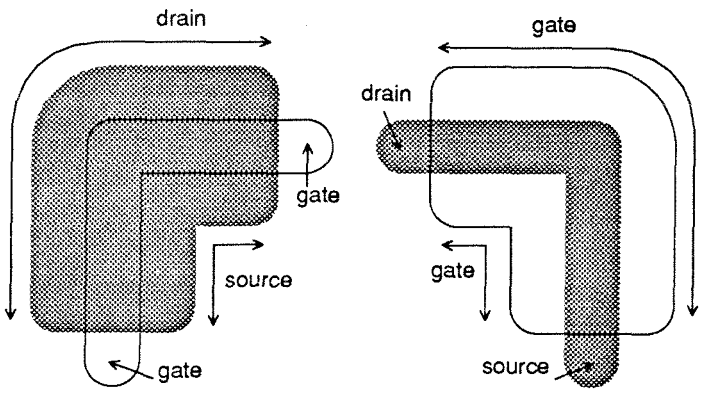


Figure 4-3: Connection Regions

The points that occur along a transistor path fall into two categories: 1) points that occur outside of the transistor region, and 2) points that occur within the transistor region. Points that occur outside the transistor region are constructed in the same manner as points that occur along an interconnection path of type $Center(T_l)$. Notice, however, in Figure 4-3 that there are two points that bound the transistor region whose connection status indicate the connectivity of the regions outside the transistor. The connection status of these two points must be either $Connect$, for connect to the gate of the transistor, $PSource$, to connect to the source, or $PDrain$, to connect to the drain. Points that occur within the transistor region are: 1) transistor points if they occur directly on the line—with connection status equal to $Connect$, or 2) connection points if they occur along the side of the transistor—with connection status equal to $PSource$, $PDrain$ or $NoConnect$ for a gate center transistor, or $Connect$ or $NoConnect$ for a channel center transistor. Figure 4-4 illustrates two transistor paths P_1 and P_2, and their connection status values.

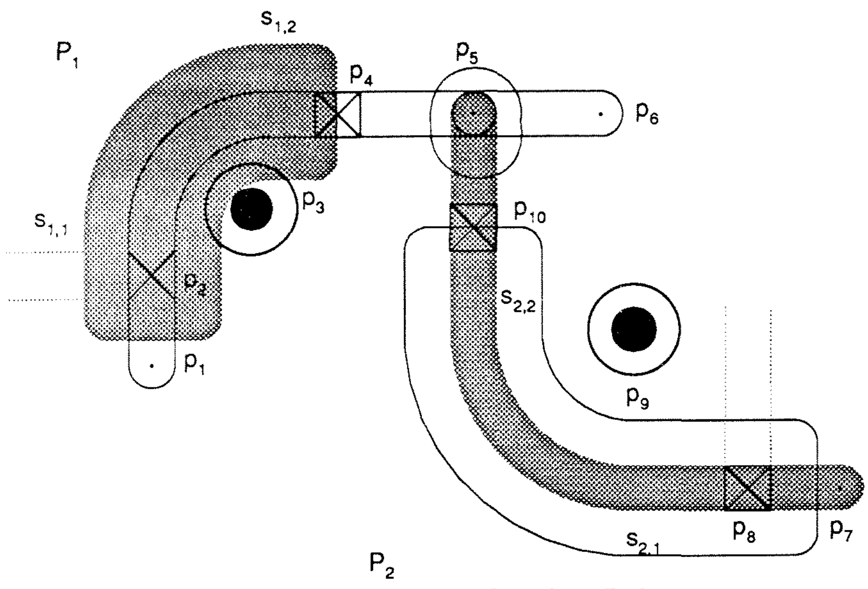


Figure 4-4: Two Transistor Paths

$S_1 = \{s_{1,1}, s_{1,2}\}$	$S_2 = \{s_{2,1}, s_{2,2}\}$
$\mathbf{p}_{1,1} = \{p_1, p_2, p_3\}$	$\mathbf{p}_{2,1} = \{p_7, p_8, p_9\}$
$\mathbf{cs}_{1,1} = \{Connect, Connect, PSource\}$	$\mathbf{cs}_{2,1} = \{PSource, Connect, NoConnect\}$
$\mathbf{p}_{1,2} = \{p_4, p_5, p_6\}$	$\mathbf{p}_{2,2} = \{p_{10}, p_5\}$
$\mathbf{cs}_{1,2} = \{Connect, Connect, Connect\}$	$\mathbf{cs}_{2,2} = \{PDrain, Connect\}$

The possible connection status designations allow the description of all valid interconnection and transistor paths. Pooh assigns the actual connection status during the circuit synthesis. It is always possible to detect the appropriate valid connection status from the circuit element context.

4.3 Node Propagation Algorithm

Pooh assigns nodes numbers to each element, and then propagates these node numbers, based on the connectivity, using Rem's `equivalence` algorithm. Pooh assigns each point and each path unique node numbers. Then, it uses the following algorithm to ensure that two elements, represented by the node numbers N_1 and N_2, are connected if and only if `nodenumbers`$[N_1] =$ `nodenumbers`$[N_2]$.

Given a point p_q, a node number N_k, and a connection status cs_q, then the following algorithm ensures that the node number of point p_q is equivalent to N_k.

```
define point_equivalence($N_k$, $cs_q$, $p_q$)
    if $ptran(p_q)$ then
        if $cs_q = CSource$ then equivalence($N_k$, $source_q$)
        else if $cs_q = CDrain$ then equivalence($N_k$, $drain_q$)
        else equivalence($N_k$, $gate_q$)
        fi
    else equivalence($N_k$, $node_q$)
    fi
enddef
```

The following algorithm ensures that connected paths maintain the same node numbers:

```
define interconnect_equiv($P_k$ : interconnect)
    for $s_j \in S_k$ do
        for $p \in \mathbf{p}_{k,j}$, $cs \in \mathbf{cs}_{k,j}$ do
            point_equivalence($node_k$, $cs$, $p$);
        od
    od
enddef

define transistor_equiv($P_k$ : transistor)
    $EndOfTransistor \leftarrow$ false;
    for $s_j \in S_k$ do
        for $p \in \mathbf{p}_{k,j}$, $cs \in \mathbf{cs}_{k,j}$ do
            if $EndOfTransistor$ then
                point_equivalence($node_k$, $cs$, $p$)
            else
                if $ptran(p)$ then equivalence all three nodes
                else if $cs = PSource$ then
                    point_equivalence($source_k$, $cs$, $p$)
                else if $cs = PDrain$ then
                    point_equivalence($drain_k$, $cs$, $p$)
                else point_equivalence($gate_k$, $cs$, $p$)
                fi
                if $EndOfTransistorPoint(p)$ then
                    $EndOfTransistor \leftarrow$ true;
                    $node_k \leftarrow$ appropriate node number based on $cs$;
                fi
            fi
        od
    od
enddef

define path_equiv($P_k$)
    if $transistor(P_k)$ then transistor_equiv($P_k$)
    else interconnect_equiv($P_k$) fi
enddef
```

The top level node propagation algorithm takes a description of a Pooh circuit—z paths and r points. First it assigns a unique node number to each path and point. Thus the number of node numbers is $\sim z+r$. Then, for each of the z paths, $P_k \in \mathcal{P}$, the algorithm invokes `path_equiv`. The number of times `equivalence` is invoked is within a factor of three of the number of times `point_equivalence` is invoked. For a path P_k with m segments, each with an average number of n points, `point_equivalence` is invoked nm times. Both n and m are small, therefore we may define a constant K that is the average constant cost of executing `path_equiv`. The expected cost of executing `equivalence` is $\log(z+r)$. Thus the time complexity of the node propagation algorithm is $C = z \times K \times \log(z+r)$ or $O(z \log(z+r))$.

The algorithms described in this chapter allow us to construct valid interconnections of circuit elements. These algorithms coupled with the previous algorithms ensure GDR correct circuit elements and valid declared interconnection. In order to ensure a fully functional circuit, we need to be able to detect illegal interactions between elements. The next chapter describes these algorithms.

5. Analysis Algebra

This chapter describes a set of operations that allow the detection of illegal interactions between the Pooh structures. Pooh creates the interconnection and transistor paths, contacts and transistor points in a way guaranteed to meet the GDRs by construction. Most of the difficult and context sensitive geometrical rules are met during the synthesis phase. There are a few simple interaction rules between elements that must be checked in order to guarantee the GDR correctness of the Pooh circuit. The GDRs that Pooh must detect are spacing and angle rules between transistors, interconnection wires and contacts. The primitive operations that allow Pooh to detect possible violations are point-point, line-point, arc-line and arc-arc interference detection. The first section presents the primitive operations, and the second section shows how the circuit GDR rules are mapped onto these operations.

5.1 Primitive Operations

The first and most straightforward operation is point-point distance calculation. The distance between any two points p_1 and p_2, according to Pythagoras, is simply:

$$PointDis(p_1,p_2)= \sqrt{(x_1 - x_2)^2+(y_1 - y_2)^2}. \tag{5-1}$$

The following function decides whether or not two point are too close:

```
define points_too_close($p_1$, $p_2$, min_dis):  boolean
    points_too_close $\leftarrow$ $PointDis(p_1,p_2) <$ min_dis;
enddef
```

The second operation is deciding if a point p_1 is too close to a line $L_2(x,y)$. The line is represented by the normalized line equation, described in equation (3-1), $L_i(x,y)= A_ix + B_iy + C_i$. The distance from a point to the line, as shown in equation (3-3), is:

$$LineDis(L_i,p_q)= A_ix_q + B_iy_q + C_i.$$

The line equation represents an infinite line but the line $L_2(x,y)$ is a finite line segment. This check must ensure that either the point really falls between the line's two end points, in which case the distance is valid, or the point is too close to one of the end points. Pooh calculates the point $p_{\perp int}$ on the line $L_2(x,y)$ where a line from $p_{\perp int}$ to p_1 is perpendicular to $L_2(x,y)$, and then uses this point to detect a true violation. This straightforward calculation, as shown in equation (3-9), is:

$$C_j = B_i x_q - A_i y_q$$
$$p_{\perp int}(L_i, p_q) = (B_i C_j - A_i C_i,\ -A_i C_j - B_i C_i).$$

The following function determines if there is a GDR violation:

```
define between_end_points($p_1$, $p_2$, $p_{int}$: point): boolean
    between_end_points ← $(x_1 \min x_2) \le x_{int} \le (x_1 \max x_2)$ &
                        $(y_1 \min y_2) \le y_{int} \le (y_1 \max y_2)$;
enddef

define point_line_too_close($L_1$: line; $bp_1$, $ep_1$, $p_2$: point;
                            min_dis: distance): boolean
    if $|LineDis(L_1, p_2)|$ < min_dis then
        point_line_too_close ← between_end_points($bp_1$, $ep_1$, $p_{\perp int}$)
            OR $PointDis(bp_1, p_2)$< min_dis OR $PointDis(ep_1, p_2)$< min_dis;
    else point_line_too_close ← false;
    fi
enddef
```

Pooh must detect the GDR interference between two arcs, $\mathcal{A}_1$ and $\mathcal{A}_2$. An arc, as defined in Section 3.1, is $\mathcal{A}_i = (r_i,\ p_i,\ p_i^c,\ \mathcal{A}_{i+1})$. Pooh uses both arc radii, r_1 and r_2, and the point distance between p_1 and p_2 to determine if the arcs could conceivably be too close, as shown in Figure 5-1. If the distance between the circles determined by r_1 and r_2 is less than the GDR spacing rule, then the arcs themselves must be examined. The two arcs $\mathcal{A}_1$ and $\mathcal{A}_2$ interfere if and only if the line from p_1 to p_2 intersects $\mathcal{A}_2$ and the line from p_2 to p_1 intersects $\mathcal{A}_1$.

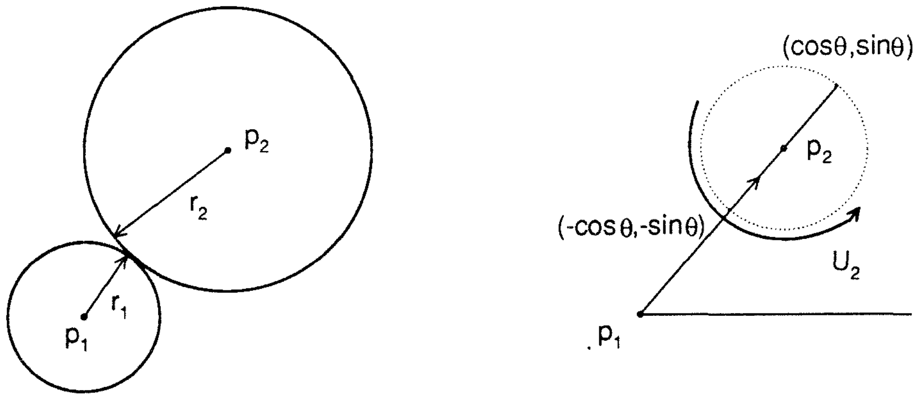


Figure 5-1: Arc Overlap Detection

A line from the point p_1 to p_2 intersects the arc $\mathcal{A}_2$ if the point on the unit circle $(-\cos\theta, -\sin\theta)$ is on the unit arc $U(\mathcal{A}_2)$, as shown in Figure 5-1. The definition of a unit arc $U(\mathcal{A}_q)$, as described in Section 3.2, is:

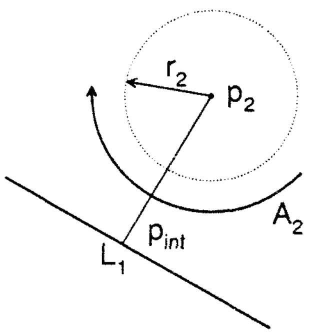


Figure 5-2: Line-Arc Proximity Detection

$$U(\mathcal{A}_q) = (p_q^c,\, p_{q+1}^c,\, dir)$$
$$dir = \begin{cases} 1, & r_q \geq 0 \\ -1, & r_q < 0. \end{cases}$$

The functions `otherhalf`, and `reverse` from the same section are used in the following algorithm. The function `point_in` is defined in terms of the circle approximation. The actual definition of this function is described in Chapter 7. This algorithm detects interference between two arcs $\mathcal{A}_1$ and $\mathcal{A}_2$:

```
define point_in($U_1$, $p_q$):  boolean
    arc overlaps with point
enddef

define arc_inbetween($p_i$, $p_j$, $U_j$; point_dis:  distance):  boolean
    if $dir_j$<0 then $U_j \leftarrow$ otherhalf(reverse($U_j$)); fi
                                   "$(-\cos\theta, -\sin\theta)$"
    arc_inbetween $\leftarrow$ point_in($U_j$,
                             [($x_i - x_j$)/point_dis,  ($y_i - y_j$)/point_dis]);
enddef

define arcs_interfere($\mathcal{A}_1$, $\mathcal{A}_2$:  arc; min_dis:  distance):  boolean
    var point_dis:  distance;
    point_dis $\leftarrow$ $PointDis(p_1, p_2)$;
    if (point_dis$-|r_1|-|r_2|$) < min_dis then
        arcs_interfere $\leftarrow$ arc_inbetween( $p_1$, $p_2$, $U(\mathcal{A}_2)$, point_dis)
            AND arc_inbetween( $p_2$, $p_1$, $U(\mathcal{A}_1)$, point_dis);
    else arcs_interfere $\leftarrow$ false;
    fi
enddef
```

Pooh must also detect GDR interference between a line $L_1(x, y)$ and an arc $\mathcal{A}_2$. A possible violation may be detected using the line-point distance method, and representing the arc $\mathcal{A}_2$ as a circle of radius r_2, as shown in Figure 5-2. Given a possible violation, the line-arc interference check is half of the arc-arc detection check, using the calculated point p_{int}, as shown in the figure.

The function that performs the line-arc check is as follows:

```
define line_arc_interfere($L_1$, $A_2$; min_dis:  distance):  boolean
   if ($|LineDis(L_1,p_2)| - |r_2|$)< min_dis then
      if between_end_points($bp_1$, $ep_1$, $p_{\perp int}$) then
         line_arc_interfere ← arc_inbetween(
                     $p_{\perp int}$, $p_2$, $U(A_2)$, $|LineDis(L_1,p_2)|$);
      else
         line_arc_interfere ← ($PointDis(bp_1,p_2)$<min_dis AND
                     arc_inbetween($bp_1$, $p_2$, $U(A_2)$, $PointDis(bp_1,p_2)$))
            OR ($PointDis(ep_1,p_2)$<min_dis AND arc_inbetween(
                     $ep_1$, $p_2$, $U(A_2)$, $PointDis(ep_1,p_2)$));
      fi
   else line_arc_interfere ← false;
   fi
enddef
```

Finally, Pooh must calculate the angle between two line segments $L_1(x,y)$ and $L_2(x,y)$. The dot product of two lines $L_1(x,y)$ and $L_2(x,y)$ emanating from a common point yields the $\cos\theta$ of the angle θ between the two lines. This result may then be compared to $\cos\varphi$, where φ is the minimum legal angle, to detect an illegal angle. Notice that the value $\cos\varphi$ is calculated once by Pooh, and not each time an angle comparison is make. The dot product is simply:

$$DotProd(L_1,L_2)= L_1 \cdot L_2 = \cos\theta = A_1A_2 + B_1B_2.$$

The function that detects an illegal angle is:

```
define check_angle($L_1$, $L_2$, $\cos\varphi$):  boolean
   ensure the directions of  $L_1$  and  $L_2$  are the same
   check_angle ← $DotProd(L_1,L_2)< \cos\varphi$;
enddef
```

The errors Pooh must detect are distance and angle errors. The four functions:

1) points_too_close,
2) point_line_too_close,
3) arcs_interfere, and
4) line_arc_interfere,

provide Pooh with the capability of detecting distance errors between lines, points, and arcs. The spacing analysis of transistors, interconnection and contacts is performed using these functions. The function check_angle allows Pooh to detect angle errors between lines. Interconnection and transistor angle checks are performed using this function.

5.2 The Geometrical Design Rules

The geometrical design rules, as mentioned before, are the rules dictated by the fabrication technology that govern how elements may be constructed, and how they may legally interact. The elements are transistors, interconnection wires and contacts. A contact is a connection point between paths of different logical layer types. Transistors are the devices of the technology, and interconnection wires are the paths that connect the devices together. Surround paths are interconnection wires that define entire regions. This section presents the interaction GDR definitions. These definitions are extensions of the synthesis GDR definitions, presented in Section 3.2.

First, let us define some Boolean functions that differentiate between the different kinds of Pooh elements. In Pooh, points may either be transistors, contacts or g-points. A g-point is a connection point between one or more paths of the same logical layer type. Paths may either be transistors or interconnection wires. Let $transistor(T_k)$ and $ptran(p_q)$ denote functions that indicate whether or not a path type or a point is a transistor. Let *contact*(p_q) and *g-point*(p_q) indicate whether a point is a contact or a g-point respectively.

Every element in Pooh is typed. As described earlier, a path P_k has a path type T_k. Points are also typed. Since a g-point is always part of a path, the type of a g-point p_q is a path type T_q, specified by the function $PathType(p_q)$. Since transistor points represent the same type of devices as transistor paths, the type of a transistor point p_r is a path type T_r, denoted by the function $TranType(p_r)$. Finally contacts have unique types c_q. This type represents the different contacts in the target technology. Let $ContactType(p_q)$ denote a function that indicates the contact type of a contact p_q.

The following are the definitions of Pooh's GDR spacing rules:

$minUnCSpace(T_k, T_l) \equiv$ *minimum edge to edge spacing between unconnected paths of type* T_k *and* T_l

$minCSpace(T_k, T_l) \equiv$ *minimum edge to edge spacing between connected paths of type* T_k *and* T_l

$minInsideSpace(T_k, T_l) \equiv$ *minimum inside spacing between a surround path of type* T_k *and a path of type* T_k

$minOutsideSpace(T_k, T_l) \equiv$ *minimum outside spacing between a surround path of type* T_k *and a path of type* T_l

$minUnCContactSpace(c_q, c_r) \equiv$ *minimum unconnected point-point spacing between contacts of type* c_q *and* c_r

$minCContactSpace(c_q, c_r) \equiv$ *minimum connected point-point spacing between contacts of type* c_q *and* c_r

$minUnCCPSpace(T_k, c_q) \equiv$ *minimum unconnected spacing between a path edge of type* T_k *and a contact of type* c_q

$minCCPSpace(T_k, c_q) \equiv$ *minimum connected spacing between a path edge of type* T_k *and a contact of type* c_q

$minInsideCSpace(T_k, c_q) \equiv$ *minimum inside spacing between a surround path of type* T_k *and a contact of type* c_q

$minOutsideCSpace(T_k, c_q) \equiv$ *minimum outside spacing between a surround*

$$\begin{aligned} & \textit{path of type } T_k \textit{ and a contact of type } c_q \\ minAngle(T_k, T_l) & \equiv \textit{minimum angle between paths of type} \\ & T_k \textit{ and } T_l. \end{aligned}$$

The spacing GDRs indicate legal distances between the edges of the paths. The spacing rules are defined in this manner because center-line distance depends on a particular path radius. Thus we define a set of functions that gives the actual spacing rules for the particular elements we are analyzing.

The distance between two line segments is determined by the distance between a line $L_1(x,y)$ and a point p_q, plus the line radius and the point radius. The radius of the line $L_{k,j}(x,y)$ is $W_k/2$, and the radius of the two parallel lines $L'_{k,j}(x,y)$ and $L''_{k,j}(x,y)$, as shown in Figure 3-5, is $mr(T_k)$. The radius of a point depends on the kind of point. The two ends of a path are constructed according to the minimum radius of the path type $mr(T_k)$ rather than the actual radius W_k, as shown in Figure 5-3. The minimum radius is maintained to allow perpendicular connection between paths of different widths. The circle surrounding p_0 of radius $r = W_k/2$ does not properly represent the path end. Thus Pooh must examine the two points p_{0l} and p_{0r} to fully determine the dimensions of the path end. These two points are intermediate points, and not actually represented, but rather calculated from the point p_0 and the line $L_1(x,y)$, using equation (3-7). Every path has at least four such points for the two end conditions—p_{0l}, p_{0r}, p_{ml}, and p_{mr} for a path of m segments. In a minimum width path the intermediate points are redundant, i.e., $p_0 \equiv p_{0l} \equiv p_{0r}$ and $p_m \equiv p_{ml} \equiv p_{mr}$. Let $IntPoint(p_q)$ denote a function that indicates whether a point is an intermediate point, and $IntLine(L_j(x,y))$ denote a function that indicates whether a line is an intermediate line $L'_j(x,y)$ or $L''_j(x,y)$.

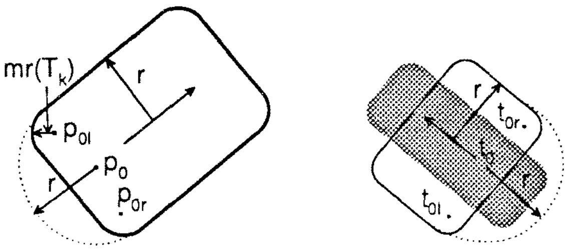


Figure 5-3: Path End Conditions

Two functions that yield the radius of the point $p_{k,q}$, meaning the point p_q on the path P_k, and the radius of the line $L_{k,j}$, meaning the line segment $L_j(x,y)$ associated with the segment s_j on the path P_k, are:

$$Pathprad(p_{k,q}) = \begin{cases} mr(T_k), & IntPoint(p_{k,q}) \\ W_k/2, & \text{otherwise.} \end{cases}$$

$$Pathlrad(L_{k,j}) = \begin{cases} mr(Outside(T_k)), & IntLine(L_{k,j}) \wedge transistor(T_k) \\ mr(T_k), & IntLine(L_{k,j}) \\ W_k/2 + OutsideOver(T_k), & transistor(T_k) \\ W_k/2, & \text{otherwise.} \end{cases}$$

The design rule functions are:

$$\begin{aligned}
ppspace(L_{k,j}, p_{l,q}) &= minUnCSpace(T_k, T_l) + Pathlrad(L_{k,j}) + Pathprad(p_{l,q}) \\
Cppspace(L_{k,j}, p_q) &= minCSpace(T_k, T_l) + Pathlrad(L_{k,j}) + Pathprad(p_{l,q}) \\
Ippspace(L_{k,j}, p_q) &= InsideSpace(T_k, T_l) - Pathprad(p_{l,q}) \\
Oppspace(L_{k,j}, p_q) &= OutsideSpace(T_k, T_l) + Pathprad(p_{l,q}) \\
cpspace(L_{k,j}, p_q) &= minUnCCPSpace\Big(T_k, ContactType(p_q)\Big) + Pathlrad(L_{k,j}) \\
Ccpspace(L_{k,j}, p_q) &= minCCPSpace\Big(T_k, ContactType(p_q)\Big) + Pathlrad(L_{k,j}) \\
ccspace(p_q, p_r) &= minUnCContactSpace\Big(ContactType(p_q), \\
&\qquad ContactType(p_r)\Big) \\
Cccspace(p_q, p_r) &= minCContactSpace\Big(ContactType(p_q), ContactType(p_r)\Big) \\
cosAngle(L_{k,j}, L_{l,i}) &= \cos\Big(minAngle(T_k, T_l)\Big).
\end{aligned}$$

5.3 Point Analysis

Pooh must check that its points' interactions with other elements do not violate any GDRs. There are three types of points: contacts, transistors, and g-points. There are also two types of checks: point-point and point-line. The possible combinations of the types with the checks are large, but fortunately unnecessary. G-points are only used to represent the ends of path segments, thus the checks associated with the g-points may be ignored in this section since interaction checks between paths will detect any possible violations involving g-points. Transistor point analysis is necessary only in a very local area. Thus most of the point checks involve contacts.

Transistor Points

Transistor points either represent a connection on a transistor path, as described in Section 4.2, or they occur at the intersection of interconnect segments. If a transistor point occurs along a transistor path, it has no GDR significance, since the checks applied to the transistor path will catch any GDR violation. Thus the only transistor point of interest to the analysis phase is the transistor point at the intersection of interconnect segments. These interconnect segments must be perpendicular as part of the construction criteria enforced by Pooh.

The transistor overlap rules, as described in Section 3.2, dictate how far the channel and the gate layers must extend beyond the gate region. Pooh either calculates the four points surrounding a transistor point p_q using equation (3-10), or checks to see if an existing point is too close to a new transistor point. Examples of these four points are shown in Figure 5-4.

Given a transistor point p_q, there must be at least three segments: s_{gate_q}, s_{source_q}, and s_{drain_q}, with the line equations $L_{source_q}(x, y) = L_{drain_q}(x, y)$. The line $L_{gate_q}(x, y)$

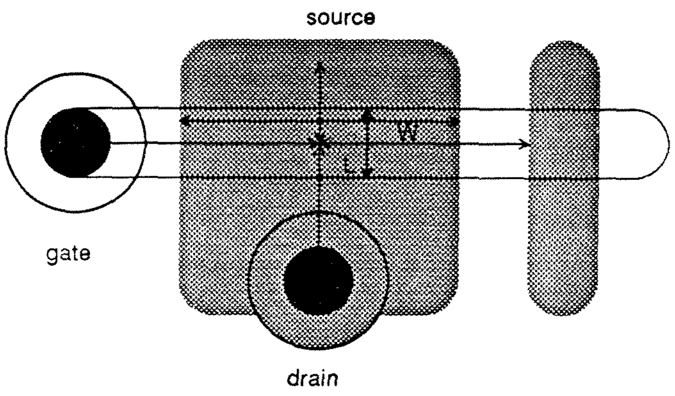


Figure 5-4: A Transistor Point

is perpendicular to the lines $L_{source_q}(x,y)$ and $L_{drain_q}(x,y)$. These constraints are maintained by Pooh during the construction of any transistor point. Pooh finds the four closest interesting points by looking along each of the three segments, and then checks that valid overlap rules are met.

The following algorithm checks for valid transistor overlap:

$$TOverLap(p_q,p_r,w) = \begin{cases} GateOver\Big(TranType(p_q)\Big) & g\text{-}point(p_r) \wedge \\ -\,mr\Big(Gate\big(TranType(p_q)\big)\Big) + w, & Gate(p_r,p_q) \\ ChannelOver\Big(TranType(p_q)\Big) & g\text{-}point(p_r) \wedge \\ -\,mr\Big(Channel\big(TranType(p_q)\big)\Big) + w, & Channel(p_r,p_q) \\ minCCPSpace\Big(TranType(p_q), & \\ ContactType(p_r)\Big) + w, & contact(p_r) \\ minCSpace\Big(TranType(p_q), & \\ TranType(p_r)\Big) + w + Width(p_r) & ptran(p_r). \end{cases}$$

```
define point_illegal($p_q$, $p_r$, twidth):  boolean
    if *calculated point* then point_illegal ← false
    else point_illegal ←
            points_too_close($p_q$, $p_r$, $TOverLap(p_q,p_r,twidth)$) fi
enddef

define transistor_overlap_illegal($p_q$):  boolean
    transistor_overlap_illegal ←
        point_illegal($p_q$, $NextInterestingPoint(s_{source_q})$, $Length(p_q)/2$) OR
        point_illegal($p_q$, $NextInterestingPoint(s_{drain_q})$, $Length(p_q)/2$) OR
        point_illegal($p_q$, $LastInterestingPoint(s_{gate_q})$, $Width(p_q)/2$) OR
        point_illegal($p_q$, $NextInterestingPoint(s_{gate_q})$, $Width(p_q)/2$);
enddef
```

Once Pooh ensures that the overlap rules for a transistor point p_q are met, the point does not need to be included in any other GDR check. Since the segments s_{gate_q}, s_{source_q}, and s_{drain_q} surround p_q on four sides, and since Pooh's model of a transistor dictates that connections to a transistor point must be perpendicular to the other connecting segments, it is not possible for any other segment to be connected to p_q except through s_{gate_q}, s_{source_q}, or s_{drain_q}. Any element that is connected to the transistor through these segments will be on the other side of the four surrounding points, since Pooh used the first interesting point along each segment to check overlap. Thus any other connected element's design rule distances will be limited by the transistors four overlap points, and not by the transistor itself. Assuming that the design rule spacing for the transistor point to unconnected elements is less than or equal to the transistor overlap rules plus the overlap point spacing to the same elements, then all unconnected elements will be in violation with one of the four overlap points or the segments s_{gate_q}, s_{source_q}, and s_{drain_q} before there is GDR violation with the transistor point. This assumption is reasonable for all known design rules. Thus we may exclude all transistor points from the remaining analysis.

Contacts

Contacts are structures in the technology that allow connections between the different interconnection layers. They have a unique set of GDRs. The contact checks however are straightforward. First, contact-contact spacing rules use point-point distance calculations. Second, contact-path spacing rules use point-line and point-arc distance calculation. The point-arc comparison is a degenerate case of an arc-arc comparison.

Pooh's model of legal interactions involving contacts is simple. Two contacts must be the connected contact spacing distance apart—*Cccspace*, if their node numbers are the same, denoted as $SameNode(p_q, p_r)$, and the unconnected contact spacing, *ccspace*, apart otherwise. Often these two spacing rules are the same. A contact must be the connected contact-path distance apart, *Ccpspace*, if the two share a common node, otherwise they must be contact-path spacing, *cpspace*, apart. Figure 5-5 illustrates the contact spacing rules.

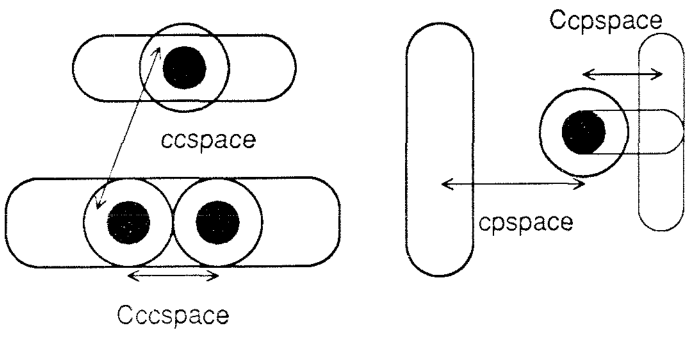


Figure 5-5: Contacts Spacing

The algorithms to analyze contacts and contacts and lines are:

```
define contact_contact_illegal($p_q$, $p_r$): boolean
    if $SameNode(p_q, p_r)$ then
        contact_contact_illegal ← points_too_close($p_q$, $p_r$, $Cccspace(p_q, p_r)$)
    else
        contact_contact_illegal ← points_too_close($p_q$, $p_r$, $ccspace(p_q, p_r)$)
    fi
enddef

define contact_path_illegal($p_q$, $P_k$): boolean
    error_found ← false;
    for $s_j \in S_k$ do
        if $SameNode(p_q, P_k)$ then
            error_found ← error_found OR
                point_line_too_close( $L_j$, $bp(s_j)$, $ep(s_j)$, $p_q$,
                        $Ccpspace(L_j(x,y), p_q)$)
                OR arcs_interfere($A_j$, [$p_q$, 0], $Ccpspace(L_j(x,y), p_q)$);
        else
            error_found ← error_found OR
                point_line_too_close( $L_j$, $bp(s_j)$, $ep(s_j)$, $p_q$,
                        $cpspace(L_j(x,y), p_q)$)
                OR arcs_interfere($A_j$, [$p_q$, 0], $cpspace(L_j(x,y), p_q)$);
        fi
        if $transistor(T_k)$ AND $EndOfTransistor(s_j)$ then $T_k$ ← $Center(T_k)$; fi
    od
    contact_path_illegal ← error_found;
enddef
```

5.4 Path Analysis

Pooh must ensure that each path's interactions with other elements is legal. The algorithm that detects a path's illegal interactions with points was described in the last section. What remains is interactions between paths. There are two types of paths: interconnection and transistors, and three types of path interactions: interconnect-interconnect, transistor-interconnect, and transistor-transistor. All of these interaction checks are very similar. For a particular path, Pooh performs two different checks: checks between connected segments, and checks between unconnected segments.

Connected Segments

Pooh has a model of what constitutes a legal interaction between unconnected and connected path segments, independent of a particular set of GDR values. The angle between two segments $s_{k,i}$ and $s_{l,j}$ sharing a common point must be greater than the minimum angle rule $minAngle(P_k, P_l)$. Two segments $s_{k,i}$ and $s_{l,j}$ connected through another segment, whose shortest distance is less than the minimum connected spacing are GDR correct if: 1) the angle between $L_{l,j}(x,y)$ and a constructed line $L'_{k,i}(x,y)$, parallel to $L_{k,i}(x,y)$ through the closest end point, is greater than a fixed angle ϑ, and 2) the end points $bp(s_{l,j})$ and $ep(s_{k,i})$ are both on the same side of the line $L_{k,i}(x,y)$. Figure 5-6 illustrates legal and illegal connected segments. Pooh uses the angle $\vartheta = 45°$. In fact

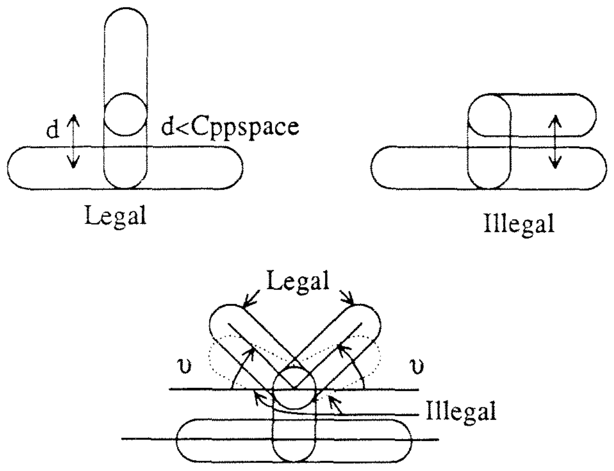


Figure 5-6: Legal and Illegal Segment Connections

this model not only applies to segments that share the same node, it also applies to segments connected through a point transistor. Figure 5-7 illustrates examples of legal interactions between segments. All other pairs of segments must meet the specified path-path spacing rules. If two segments share a common node, then they must be *Cppspace* apart, otherwise they must be *ppspace* apart.

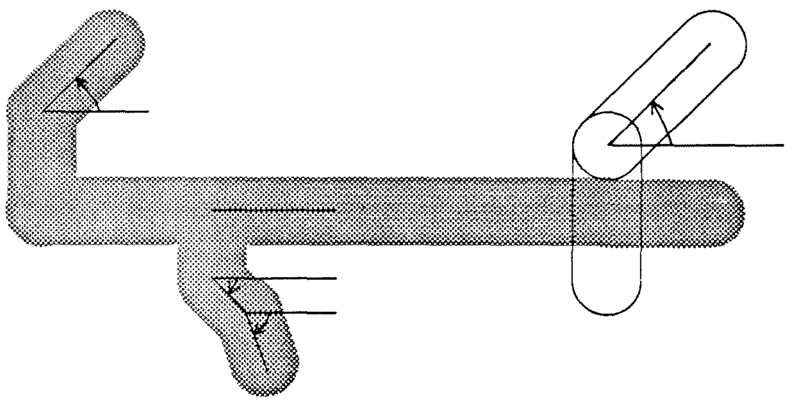


Figure 5-7: Legal Connections between Segments

The algorithm to check all connected interactions for the segment $s_{k,j}$ is:

```
define check_line_angle($L_{k,i}$, $L_{l,j}$, PointInMiddle):  boolean
   check_line_angle ← if pointInMiddle then
         check_angle( $L_{k,i}$, $L_{l,j}$, $cosAngle(L_{k,i}, L_{l,j})$)
         AND check_angle([$-A_{k,i}, -B_{k,i}$], $L_{l,j}$, $cosAngle(L_{k,i}, L_{l,j})$)
         else check_angle( $L_{k,i}$, $L_{l,j}$, $cosAngle(L_{k,i}, L_{l,j})$) fi
enddef
```

```
define segment_connections_illegal($s_{k,j}$):  boolean
    error_found ← false;
    for $p \in \mathbf{p}_{k,j}$ do
        for $s_i \in$ segments connecting to $p$ do
            error_found ← error_found OR
                check_line_angle($L_{k,j}$, $L_i$, $p \neq lp(s_{k,j}) \wedge p \neq p_{k,j}$);
            if not error_found then
                for $p_2 \in \mathbf{p}_i$ do
                    error_found ← error_found OR
                            removed_segments_illegal($s_{k,j}$, $p_2$);
                od
            fi
        od
    od
    segment_connections_illegal ← error_found;
enddef
```

Segment Interactions

Pooh must detect path segments that violate the spacing rules maintained in the GDR description. Thus for each path segment, Pooh examines "interesting" other segments that may create potential spacing violation. How exactly Pooh might decide what constitutes an "interesting" segment is not addressed in this chapter. Chapter 7 describes an approach to this segment proximity problem. This section describes how, given two segments, Pooh detects a GDR violation.

Each Pooh path is composed of path segments. Each path segment includes a line segment and an arc. Figure 5-8 illustrates examples of interconnection and transistor paths.

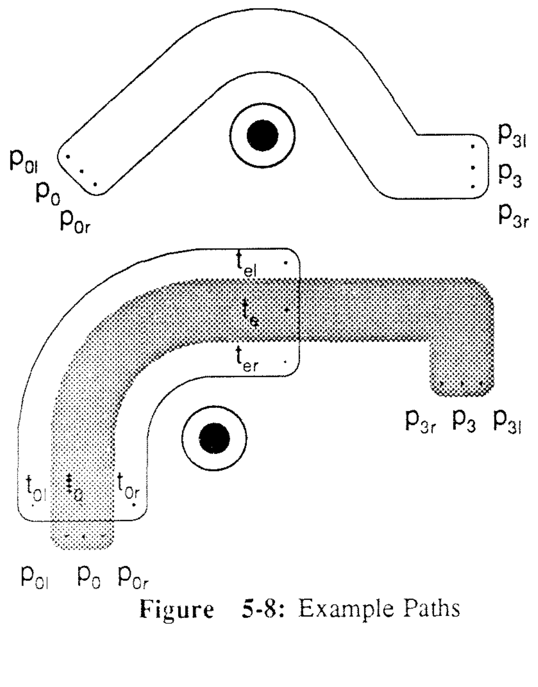


Figure 5-8: Example Paths

Pooh compares two segments s_1 and s_2 by comparing the two lines $L_1(x,y)$ and $L_2(x,y)$, the two arcs $\mathcal{A}_1$ and $\mathcal{A}_2$, and the arcs to the lines $L_1(x,y)$ and $\mathcal{A}_2$, and $L_2(x,y)$ and $\mathcal{A}_1$. Obviously, if one or both of the arcs is of zero radius, then some or all of the arc checks are unnecessary. The `arcs_interfere` algorithm detects an arc-arc violation, and the `line_arc_interfere` algorithm detects a line-arc violation. Two lines may be compared by noticing: given two non-intersecting lines $L_1(x,y)$ and $L_2(x,y)$, the closest point to $L_1(x,y)$ on $L_2(x,y)$ will always be at one of the end points of $L_2(x,y)$. Thus, Pooh may detect a spacing violation between two line segments $L_1(x,y)$ and $L_2(x,y)$ by first checking for intersection. If there is no intersection, then Pooh compares the end points of the $L_1(x,y)$ line to $L_2(x,y)$, and vice-versa. Intersection may be determined by using the signed distance $LineDis(L_i(x,y), p_q)$. Two lines intersect if and only if:

$$Sign\Big(LineDis(L_1(x,y), bp(L_2(x,y)))\Big) \neq Sign\Big(LineDis(L_1(x,y), ep(L_2(x,y)))\Big).$$

The radius of the beginning and ending of a Pooh path P_k is constructed according to the minimum radius of the path, $mr(T_k)$, rather than the path radius $W_k/2$. This construction allows perpendicular connection between paths of different widths. While this approach to path ends has been found to be extremely convenient, it adds some complexity to the analysis of two path segments. If the first or the last point of an m segment path, points p_0 or p_m, is found to be too close to another line segment $L_j(x,y)$, based on the GDR *ppspace*, which uses the point radius $Pathprad(p_0)$, an error might be detected where none exists. If this condition is detected, Pooh must recalculate the distance using the intermediate points p_{0l} and p_{0r} or p_{ml} and p_{mr}. A similar situation occurs with the line $L_i(x,y)$, at which point $L_i'(x,y)$ and $L_i''(x,y)$ must be introduced.

Finally, an additional level of complexity is introduced by transistor paths. A transistor path P_k includes two logical layers, introduced as $Center(T_k)$ and $Outside(T_k)$ in Section 3.2. If Pooh were to assume that the transistor path radius was $W_k/2$, then there are GDR violations that would not be caught, since this width is only the width of the center layer and does not take into account the outside layer. But the radius of a transistor path calculated by the function $Pathlrad(L_{k,j})$, assumes the worst case radius everywhere. The transistor region of a transistor path begins on the first segment, or s_1. For an m segment transistor path there is a segment, denoted s_e, where $1 \leq e \leq m$, on which the transistor region ends. Figure 5-8 illustrates a transistor path example where the number of segments $m = 3$, and where the transistor region ends on the second segment, or $e = 2$. If a potential error is detected on the segments s_1 or s_e, Pooh must examine the segment more closely. In fact the segment s_e may be decomposed into two separate line segments: (1) the line $L_e(x,y)$ with beginning point equal to $bp(s_e)$ and ending point equal to $ep(s_e)$, path radius equal to $W_k/2$, and (2) the line $L_e(x,y)$ with beginning point $bp(s_e)$, and ending point equal to the transistor end point t_e, path radius equal to $W_e/2 + OutsideOver(T_k)$. Given these two line segments, Pooh can detect the existence of a true GDR violation. Segment s_1 receives similar treatment. All path segments included in a transistor path beyond the segment s_e are identical to an interconnect path of type $Center(T_k)$, as shown in the figure.

The functions that compare two lines $L_1(x,y)$ and $L_2(x,y)$, and introduce the intermediate lines and points as necessary, are:

```
define compare_lines_to_point($L_{k,j}$, $p_{l,i}$):  boolean
    compare_lines_to_point ←
        point_line_too_close($L'_{k,j}$, $bp(L'_{k,j})$, $ep(L'_{k,j})$, $p_{l,i}$, ppspace($L'_{k,j}$, $p_{l,i}$))
        OR point_line_too_close($L''_{k,j}$, $bp(L''_{k,j})$, $ep(L''_{k,j})$, $p_{l,i}$,
                ppspace($L''_{k,j}$, $p_{l,i}$));
enddef

define compare_line_to_points($L_{k,j}$, $L_{l,i}$, $p_{l,i}$):  boolean
    compare_line_to_points ←
        point_line_too_close($L_{k,j}$, $bp(L_{k,j})$, $ep(L_{k,j})$, $p_{left_i}$($L_{l,i}$, $p_{l,i}$),
                ppspace($L_{k,j}$,$p_{left_i}$($L_{l,i}$, $p_{l,i}$)))
        OR point_line_too_close($L_{k,j}$, $bp(L_{k,j})$, $ep(L_{k,j})$, $p_{right_i}$($L_{l,i}$, $p_{l,i}$),
                ppspace($L_{k,j}$,$p_{right_i}$($L_{l,i}$, $p_{l,i}$)));
enddef

define compare_lines_to_points($L_{k,j}$, $L_{l,i}$, $p_{l,i}$):  boolean
    compare_lines_to_points ←
        compare_lines_to_point($L_{k,j}$, $p_{left_i}$($L_{l,i}$, $p_{l,i}$))
        OR compare_lines_to_point($L_{k,j}$, $p_{right_i}$($L_{l,i}$, $p_{l,i}$));
enddef

define compare_two_lines($L_{k,j}$,$L_{l,i}$, $kfirstorlast$, $lfirst$, $llast$):  boolean
    error_found ← segments_cross( $L_{k,j}$,$L_{l,i}$);
    if $lfirst$ then
        error_found ← error_found OR
            if $kfirstorlast$ then compare_lines_to_points ($L_{k,j}$, $L_{l,i}$, $bp(L_{l,i})$)
            else compare_line_to_points($L_{k,j}$, $L_{l,i}$, $bp(L_{l,i})$) fi;
    else
        error_found ← error_found OR
            if $kfirstorlast$ then compare_lines_to_point ($L_{k,j}$, $bp(L_{l,i})$)
            else point_line_too_close($L_{k,j}$, $bp(L_{k,j})$, $ep(L_{k,j})$ $bp(L_{l,i})$,
                    ppspace($L_{k,j}$, $bp(L_{l,i})$)) fi;
    fi
    if $llast$ then
        error_found ← error_found OR
            if $kfirstorlast$ then compare_lines_to_points ($L_{k,j}$, $L_{l,i}$, $ep(L_{l,i})$)
            else compare_line_to_points($L_{k,j}$, $L_{l,i}$, $ep(L_{l,i})$) fi;
    else
        error_found ← error_found OR
            if $kfirstorlast$ then compare_lines_to_point ($L_{k,j}$, $ep(L_{l,i})$)
            else point_line_too_close($L_{k,j}$, $bp(L_{k,j})$, $ep(L_{k,j})$ $ep(L_{l,i})$,
                    ppspace($L_{k,j}$, $ep(L_{l,i})$)) fi;
    fi
    compare_two_lines ← error_found;
enddef
```

The algorithm that compares two interconnection paths for possible design rule violations is:

```
define wire_in_error($P_k$):  boolean
    error_occured ← false;
    for $s_j \in S_k$ do
        for $s_{l,i}$ not connected and close enough to be a potential violation do
            error_occured ← error_occured OR
                compare_two_lines($L_{k,j}$, $L_{l,i}$, first or last segment in $P_k$,
                        first segment in $P_l$, last segment in $P_l$);
            if $r_j \neq 0$ then
                error_occured ← error_occured OR
                    line_arc_interfere($L_{l,i}$, $A_j$, ppspace($L_{k,j}$, $L_{l,i}$)); fi
            if $r_{l,i} \neq 0$ then
                error_occured ← error_occured OR
                    line_arc_interfere($L_{k,j}$, $A_i$, ppspace($L_{k,j}$, $L_{l,i}$)); fi
            if $r_j \neq 0$ AND $r_{l,i} \neq 0$ then
                error_occured ← error_occured OR
                    arcs_interfere($A_j$, $A_{l,i}$, ppspace($L_{k,j}$, $L_{l,i}$)); fi
        od
    od
    wire_in_error ← error_occured;
enddef
```

Algorithms that find violations involving transistor paths and surround paths are similarly defined. Algorithms for transistor paths split the first segment of a path and the segment where the transistor region ends s_e into two segments if a potential violation is discovered. The algorithm for surround paths uses a signed distance while detecting potential violations.

These algorithms provide Pooh with the capability of detecting an illegal interaction, either spacing or angle, between any Pooh elements. Thus the Pooh representation has the capability of: (1) defining legal transistor elements and legal interconnect wires, (2) fully synthesizing the transistor connectivity, and (3) detecting an illegal interaction between two elements. Therefore Pooh can fully represent any transistor level design.

6. Pooh Hierarchical Composition

The Pooh design methodology supports and encourages the use of hierarchy. A cell is the building block of a Pooh design. Pooh cells are composed together to form larger cells, which in turn are composed with other cells. Previous chapters have discussed how to describe transistors and their interconnection, and how to: (1) ensure that these circuits meet the GDRs, and (2) maintain the circuit connectivity. Circuits of 100,000 devices are common today, and the device count is on its way up. It is no longer possible for a single designer or even a team of designers to comprehend every detail in designs of this magnitude without an abstraction capability. A design representation that merely detects GDR violations and maintains connectivity on a fully instantiated design would not be a useful tool. The massive amount of information present in a large design makes the detection process too time consuming. Once an error is found, the designers must sort through the design and fix the problem. Each iteration is extremely costly at best and eventually impossible. But, if there is a way to connect two or more cells together and derive information about the composite, then an abstraction is possible. Pooh supports this type of design approach.

This chapter describes how to compose cells described in the Pooh representation. The first section describes the representation of Pooh cell definitions. The next three sections describe the composition algorithms that provide three important functions: 1) verify logical connectivity, 2) maintain the electrical node information about the new cell being formed by the composition, and 3) ensure the Geometrical Design Rule correctness of the new cell. The last section presents a proof of closure under composition.

6.1 Cell Representation

Internal Topology

A Pooh cell definition, denoted as C_i, contains enough information to represent both its internal topology and its external interface. The topology of a leaf cell [Rowson 80], is a set of paths $\mathcal{P}_i$, and a set of points $\mathbf{p}_i$. A cell's external interface is defined by a set of sides $\mathcal{Q}_i \equiv \{Q_1, Q_2, \ldots Q_n\}$ that form a simply connected polygon. The shorthand notation $Q_{i,j}$ is used to mean a side Q_j on the cell C_i, or $Q_j \in \mathcal{Q}_i$. Each directed side $Q_{i,j}$ has a corresponding set of ports $\mathcal{O}_{i,j} \equiv \{o_1, o_2, \ldots o_m\}$ and a line equation $L_{i,j}(x, y)$, where the ports are guaranteed by Pooh to be both on the line, and between the two end points of the side. Figure 6-1 illustrates a cMOS leaf cell included in one bit of a bit serial multiplier [Mead 85]. The figure shows both the internal topology and the external interface.

In a hierarchical design, a cell's internal topology includes not only paths but instances of other cells. Thus, the internal topology of the cell C_i includes not only the sets $\mathcal{P}_i$ and $\mathbf{p}_i$, but a set of instances $\mathcal{I}_i \equiv \{I_1, I_2, \ldots I_n\}$, and a set of internal connections

$\mathbf{ic}_i \equiv \{ic_1, ic_2, \ldots ic_n\}$. The notation $I_{i,k}$ is used to mean the k^{th} instance in the cell C_i, or $I_k \in \mathcal{I}_i$ and $ic_{i,l}$ is the l^{th} internal connection in the cell C_i, or $ic_l \in \mathbf{ic}_i$. Each cell instance $I_{i,k}$ is a placed version of another cell definition, in this discussion denoted as $cell(I)$. Each internal connection $ic_{i,l}$ notes a connection between two instances I and I' where $I, I' \in \mathcal{I}_i$. Thus a Pooh cell C_i is fully characterized as $[\mathcal{P}_i, \mathbf{p}_i, \mathcal{Q}_i, \mathcal{I}_i, \mathbf{ic}_i]$. When a cell is in the process of being defined, the values of these sets are considered to be the current *state* of the cell C_i.

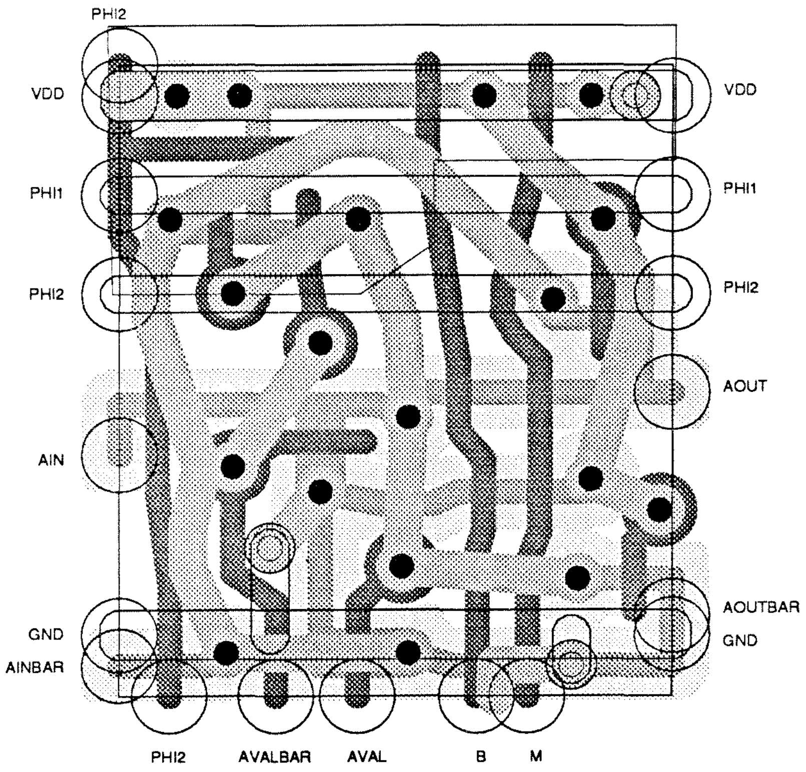


Figure 6-1: Bit Serial Multiplier Leaf Cell

The cell ports $\mathcal{O}_i \equiv \mathcal{O}_1 \cup \mathcal{O}_2 \cup \ldots \mathcal{O}_n$, for a cell with n sides, are the external connection points of the cell C_i. The ports dictate how the cell interfaces to all other cells. Each port is typed; the port type indicates the kind of signal the cell expects at the port. The port types are analogous to data types in a programming language [Jensen 74]. A linear set of ports are grouped together as $\mathcal{O}_{i,j}$ along a side $Q_{i,j}$. A side is characterized by a line segment and provides the mechanism by which two cells are composed. The cell C_i is composed with C_k by connecting two sides Q_j and Q_l, where $Q_j \in \mathcal{Q}_i$, and $Q_l \in \mathcal{Q}_k$. The result of this composition forms the current *state* of a new cell C_q.

A hierarchical cell C_i is incrementally defined by adding instances I_q, $q = 1 \ldots n$ to the existing instances of the cell C_i. Each composition takes the cell's current *state* to a next *state* based on information derived from the instance. The instance I_q includes placement information about its cell definition or $C \equiv cell(I_q)$. There is at least one internal connection ic_r, where $ic_r \in \mathbf{ic}_i$, that represents the connectivity between this instance and all other instances, $I \in \mathcal{I}_i$. Finally the cell C contributes zero or more ports to the sides $\mathcal{Q}_i$ in the next *state* of C_i.

The cell instances $\mathcal{I}_i$ indicate the cells composed to form the current topology of the cell C_i. Each instance provides placement information about the cell definition. The placement information is simply represented as a two-dimensional graphics transformation matrix [Newman 79], where the orientation information is provided when composition is defined, and the translation information is derived from the composition. An instance is positioned in order to connect the two sets of ports defining the composition.

Each internal connection $ic_{i,k}$ is an indication that two ports from two distinct instances are electrically equivalent within the cell C_i. Each time a composition occurs between two sides $Q_{i,j}$ and $Q_{q,r}$, an internal connection ic is defined for each pair of ports that connect. The internal connections provide the representation of the cell's internal connectivity and allow the cell's external interface $\mathcal{Q}_i$ to reflect the cell's connectivity. Figure 6-2 illustrates a cell composed from several other cells.

External Cell Interface

Though it is important to fully represent a cell's internal topology, it is perhaps more important to be able to derive a cell abstraction $E(C_i)$. This cell abstraction must fully represent the external interface of the cell C_i, while hiding the details of the cell implementation. Pooh actually uses $E(C_i)$ for composition instead of the cell C_i and guarantees the composition according to the three criteria — logical, electrical and geometrical. The external cell $E(C_i)$ is fully characterized as $[\mathcal{P}_i', \mathbf{p}_i', \mathcal{Q}_i]$. The paths $\mathcal{P}_i'$ and points $\mathbf{p}_i'$ are those of geometrical interest along the cell interface. These sets are derived from both the paths $\mathcal{P}_i$ and points $\mathbf{p}_i$ and other cell abstractions $E(cell(I_{i,q}))$, where $I_{i,q} \in \mathcal{I}_i$. The size of $\mathcal{P}_i'$ and $\mathbf{p}_i'$ is small as compared to an instantiation of C_i. Figure 6-2 illustrates the information present during the definition of the cell, and the cell abstraction of the new cell. The cells in the figure comprise one bit stage of a cMOS bit serial multiplier [Mead 85].

6.2 A Composition Algorithm

Given the current cell C_i, a composible interface $E(C_q)$, and two or more sides to be composed, Pooh composition is a function that maps the current *state* of C_i $[\mathcal{P}_i, \mathbf{p}_i, \mathcal{Q}_i, \mathcal{I}_i, \mathbf{ic}_i]$ into the next *state* $[\mathcal{P}_i^n, \mathbf{p}_i^n, \mathcal{Q}_i^n, \mathcal{I}_i^n, \mathbf{ic}_i^n]$. Initially the sets $\mathcal{Q}_i = \emptyset, \mathcal{I}_i = \emptyset$, and $\mathbf{ic}_i = \emptyset$. If either $\mathcal{Q}_i$ or $\mathcal{Q}_q$ form a concave polygon, the composition may include more than one pair of sides. For example, in Figure 6-3, C_i connects to $E(C_q)$ by connecting $\mathcal{Q}_{i,4}$ to $\mathcal{Q}_{q,2}$ and $\mathcal{Q}_{i,5}$ to $\mathcal{Q}_{q,1}$. This composition may be performed by iteratively connecting paired sides and then checking for consistency. All side pairs must be connected before Pooh checks the GDR correctness.

Two cells are connected by connecting two sets of sides $\mathcal{Q}_i^c = \{Q_j, Q_{j+1} \ldots Q_n\} \subset \mathcal{Q}_i$ and $\mathcal{Q}_q^c = \{Q_j, Q_{j+1} \ldots Q_n\} \subset \mathcal{Q}_q$. The sets $\mathcal{Q}_i^c$ and $\mathcal{Q}_q^c$ consist of connected edges — that is given the set $\mathcal{Q}$, the sides contained in the sets $\mathcal{Q}^c$ and $\mathcal{Q} - \mathcal{Q}^c$ intersect at two and only two points. The composition function must:

(1) order the sets of ports appropriately and syntactically verify that each pair of ports, $o_j \in \mathcal{O}_i^c, o_j' \in \mathcal{O}_q^c$, may logically be connected,

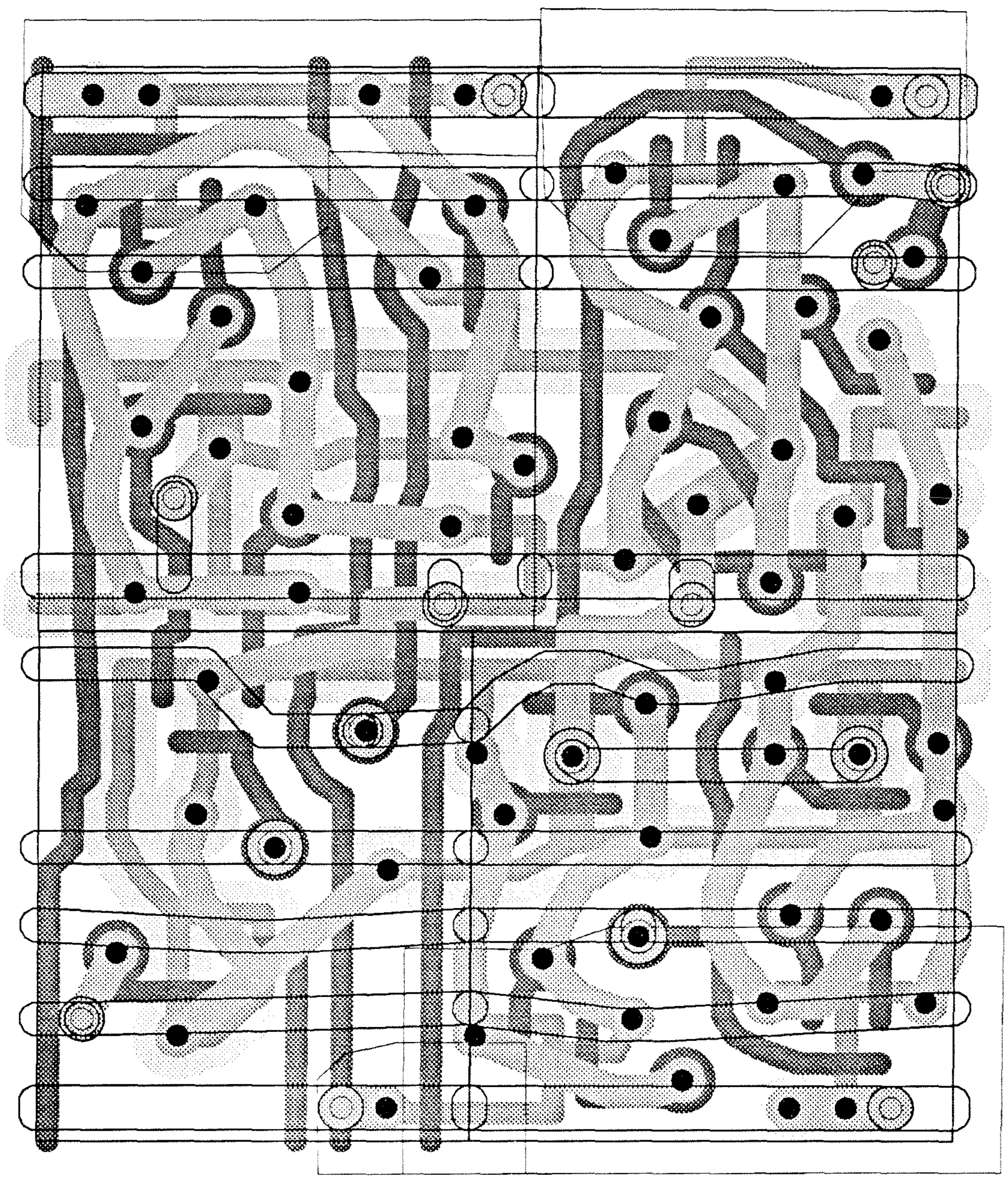


Figure 6-2: (a) Full Geometry Of One Bit Stage

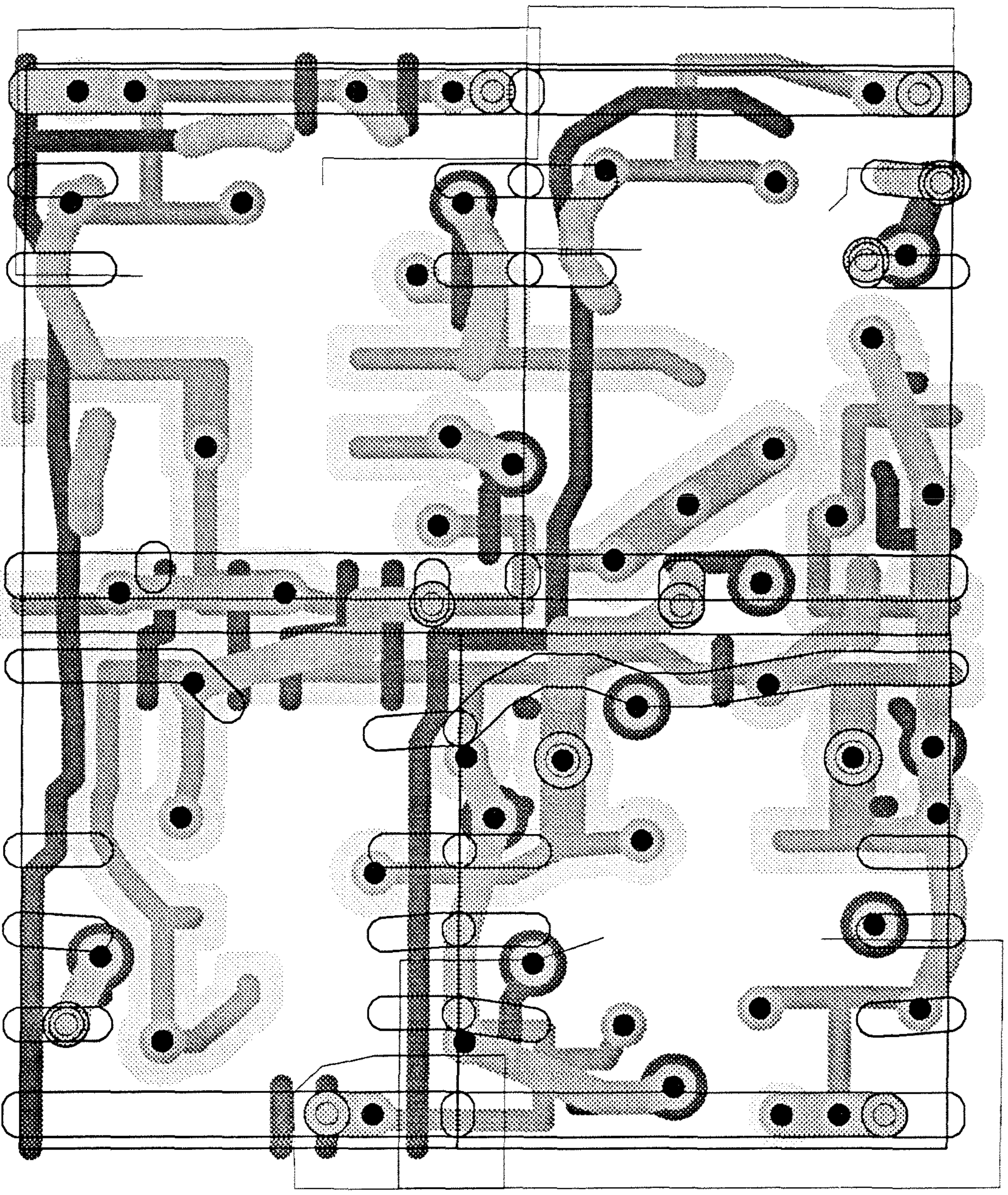


Figure 6-2: (b) Composition of Cells

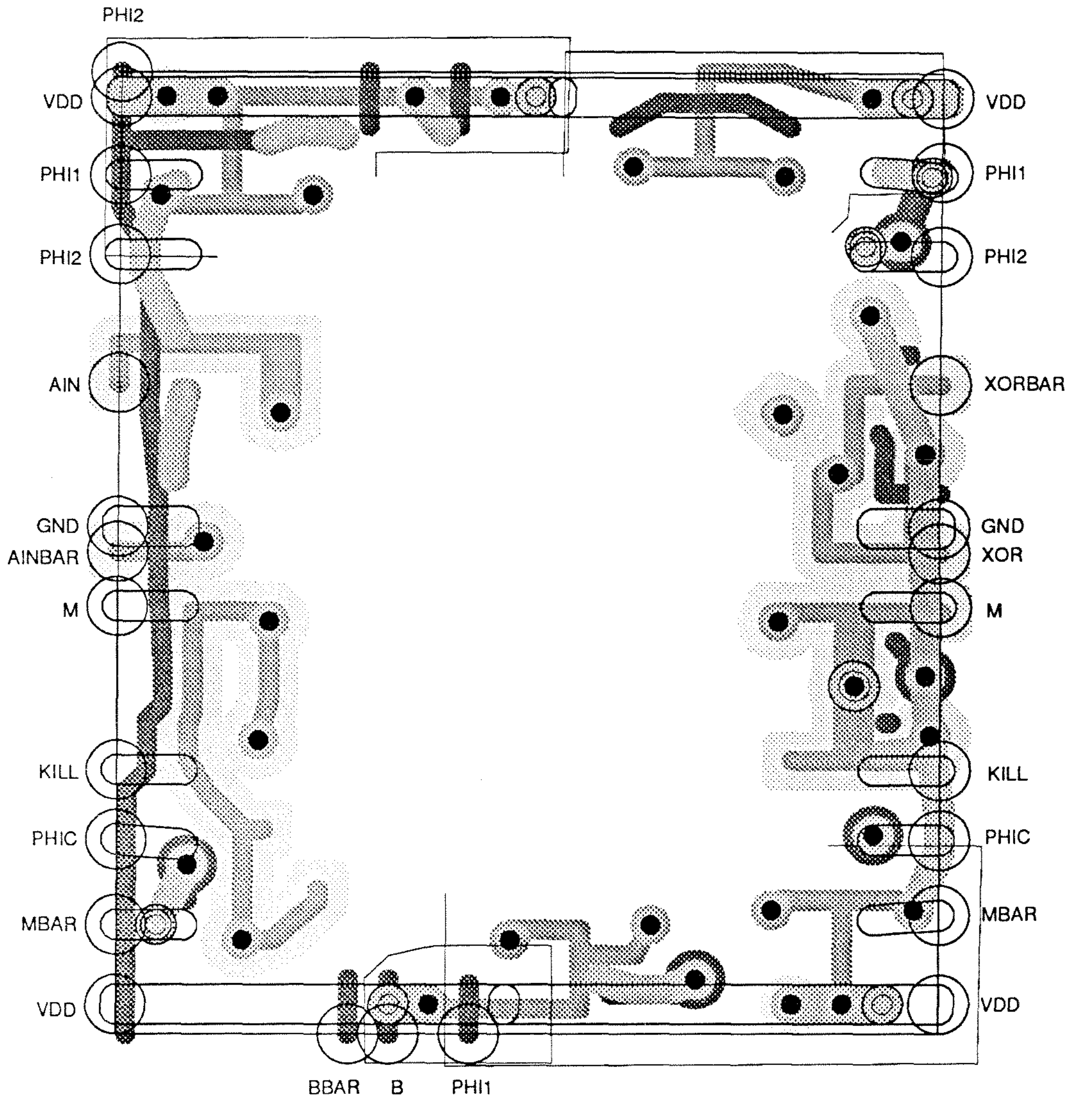


Figure 6-2: (c) Cell Abstraction

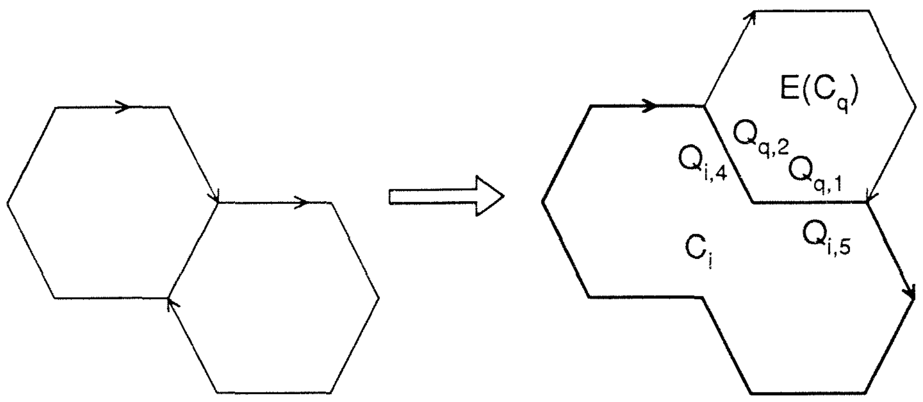


Figure 6-3: *Concave Connecting Sides*

(2) calculate the placement of the cell $E(C_q)$ based on the side abutment,

(3) add an instance I_q representing the cell $E(C_q)$ to the set $\mathcal{I}_i$,

(4) add an internal connection ic for each connection between the two port sets $\mathcal{O}_{i,j}$ and $\mathcal{O}_{q,j}$,

(5) update the connectivity of the cell C_i,

(6) verify the GDR correctness of C_i and if necessary adjust the placement of the new instance, and finally

(7) update the sets p_i, $\mathcal{P}_i$ and $\mathcal{Q}_i$.

Logical Verification

In order to understand Pooh composition, it is easier to simplify the composition function, describe how the simple function works, then introduce the additional functionality. Assume for the moment that Pooh composition takes two side sets $\mathcal{Q}_i^c$ and $\mathcal{Q}_q^c$, checks that the ports may logically connect, adds a new placed instance, and updates the set $\mathcal{Q}_i$. This composition function is strictly a syntactic operation since it no longer maintains the GDR integrity or the cell connectivity. For clarity, let us also assume that composition is done by abutment [Rowson 80] — i.e., once the composition is complete $|L_{i,j}^c(x, y)| = |L_{q,j}^c(x, y)|$.

Each port o is both typed and named. The port type is chosen from a small set of allowable signal types $\{input, output, input/output, clock, vdd, ground\}$. The function $PortConnect(o)$ indicates the set of port types to which this port may connect. The port name, if present, indicates the signal the port expects. For example, the port type may be clock, and the signal may be ϕ_1. The algorithm to verify logical connectivity is as follows:

```
define Connect_Ports($\mathcal{O}_i$, $\mathcal{O}_q$):  boolean
    Connect_Ports ← true;
    for $o_j \in \mathcal{O}_i, o'_j \in \mathcal{O}_q$ do
        if $PortType(o'_j) \notin PortConnect(o_j)$ OR $PortType(o_j) \notin PortConnect(o'_j)$
            then Connect_Ports ← false
        fi
        if $exists(PortName(o_j))$ AND $exists(PortName(o'_j))$ AND
            $\sim CompatibleNames(PortName(o_j), PortName(o'_j))$ then
            Connect_Ports ← false
        fi
    od
enddef
```

The ports $\mathcal{O}_i$ of a cell definition C_i specify where a connection may occur. A particular instance of a cell may not use the complete set of ports, but instead use a subset $\mathcal{O}_i^I \subseteq \mathcal{O}_i$. The unused ports may either be individual ports, where $o_{omit} \in \mathcal{O}_i$, in the cell C_i, or entire sides, where $\mathcal{O}_{omit}$ is the entire set of ports on the side $Q \in \mathcal{Q}_i$. For example in the Figure 6-2, this cell is repeated n times to form an n bit bit-serial multiplier. The last instance, or bit n, does not use any ports on the west side, therefore these ports are omitted during the n^{th} composition. Individual port omissions are common on buses that occur along an edge and cover the intersection of two edges. A port to the same wire may appear on two edges, and different instances omit one or the other. A composition in Pooh requires the cell sides $\mathcal{Q}_q$, the sides to be composed, $\mathcal{Q}_q^c$, and a set of sides with the ports to be omitted $\mathcal{Q}_{omit_q}^c$. The ports in the set $\mathcal{Q}_{omit_q}^c$ may include both ports on the composition sides $\mathcal{Q}_q^c$, and ports not on the composition sides, or $\mathcal{Q}_q - \mathcal{Q}_q^c$. In either case, the ports in the set $\mathcal{Q}_{omit_q}^c$ are not connected and do not remain after the composition. The paths and points used to define these omitted ports become normal Pooh path and points, and are treated by Pooh accordingly.

The function that removes omitted ports from a side $Q_{compose}$ is:

```
define Omit_Ports($Q_{compose}$, $\mathcal{Q}_{omit}$)
    for $Q_o \in \mathcal{Q}_{omit}$ do
        if $L_o(x,y) = L_{compose}(x,y)$ then
            $\mathcal{O}_{compose} \leftarrow \mathcal{O}_{compose} - \mathcal{O}_o$;
        fi
    od
enddef

define Omit_All_Ports($\mathcal{Q}_{compose}$, $\mathcal{Q}_{omit}$)
    for $Q \in \mathcal{Q}_{compose}$ do
        Omit_Ports($Q$, $\mathcal{Q}_{omit}$);
    od
enddef
```

In order to create a new set of sides $\mathcal{Q}_i^n$, Pooh must first compute the placement information for the new instance I_q. The orientation, rotation and mirroring of the instance is specified by the designer, since without this information the composition operation is ambiguous. Assuming that the side $Q_{q,j} \in \mathcal{Q}_q^c$ is oriented correctly, then in order to compose $Q_{q,j}$ with $Q_{i,j} \in \mathcal{Q}_i^c$, either $A_{i,j} = A_{q,j}$ and $B_{i,j} = B_{q,j}$ or $A_{i,j} = -A_{q,j}$ and $B_{i,j} = -B_{q,j}$. It is thus a straightforward transformation to change both side sets into clockwise polygons, as shown in Figures 6-3 and 6-4. Once the directions of the sides are consistent, it is easy to calculate the translation factors Δx and Δy and ensure that all sides $Q_{i,j}$ and $Q_{q,j}$ abut using the translation factor.

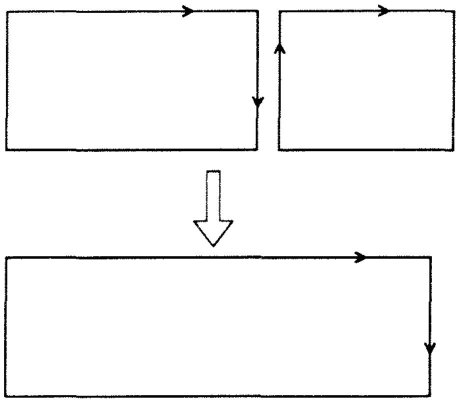


Figure 6-4: Connecting Sides

Once the translation factor is applied to all the ports, the function that takes the two sets $\mathcal{Q}_i$ and $\mathcal{Q}_q$, and maps them into a new set $\mathcal{Q}_i^n$ is:

$$Merge(Q_l, Q_m) = \begin{cases} [L_l(x,y), \mathcal{O}_l \cup \mathcal{O}_m], & A_l = A_m \wedge B_l = B_m \wedge C_l = C_m \\ \{Q_l, Q_m\}, & \text{otherwise.} \end{cases}$$

$$\begin{aligned} &ComposeSides(\mathcal{Q}_i, \mathcal{Q}_q, \\ &\qquad Q_{j\ldots n}, Q_{k\ldots m}) = (\mathcal{Q}_i - \{Q_{j-1}, Q_j, Q_{j+1} \cdots Q_n, Q_{n+1}\}) \\ &\qquad\qquad \cup\, Merge(Q_{j-1}, Q_{m+1}) \cup Merge(Q_{n+1}, Q_{k-1}) \\ &\qquad\qquad \cup\, (\mathcal{Q}_q - \{Q_{k-1}, Q_k, Q_{k+1} \cdots Q_m, Q_{m+1}\}). \end{aligned}$$

The function that composes an external cell $E(C_q)$ to the current *state* of C_i, removes the omitted ports, and produces the next *state* of the cell is:

```
define Compose($C_i$, $E(C_q)$, $\mathcal{Q}_i^c$, $\mathcal{Q}_q^c$, $\mathcal{Q}_{omit_i}$, $\mathcal{Q}_{omit_q}$, orient: transform):
                boolean
    var composible: boolean;
        $I$: instance;
    composible ← true;
    $I$ ← newinstance($E(C_q)$);
    orientCellSides($I(E(C_q))$, $\mathcal{Q}_q^c$, orient);
    make sure $\mathcal{Q}_i$ and $\mathcal{Q}_q$ are clockwise polygons
    for $Q_j \in \mathcal{Q}_i^c, Q'_j \in \mathcal{Q}_q^c$ do
        Omit_Ports($Q_j$, $\mathcal{Q}_{omit_i}$);
        Omit_Ports($Q'_j$, $\mathcal{Q}_{omit_q}$);
        if $A_j \neq -A'_j$ OR $B_j \neq -B'_j$ OR $size(\mathcal{O}_j) \neq size(\mathcal{O}'_j)$ then
            composible $\leftarrow$ false fi
        if $j = 1$ then $\Delta x, \Delta y \leftarrow$ translation($\mathcal{O}_j$, $\mathcal{O}'_j$)
        else if $\Delta x, \Delta y \neq$ translation($\mathcal{O}_j$, $\mathcal{O}'_j$) then composible $\leftarrow$ false
        fi
        apply_translation($\mathcal{O}'_j, \Delta x, \Delta y$);
        composible $\leftarrow$ composible AND Connect_Ports($\mathcal{O}_j$, $\mathcal{O}'_j$);
        $C'_j \leftarrow C'_j - (A'_j \times \Delta x + B'_j \times \Delta y)$;
    od
    for $Q \in (\mathcal{Q}_q - \mathcal{Q}_q^c)$ do
        Omit_Ports($Q$, $\mathcal{Q}_{omitq}$);
        apply_translation($\mathcal{O}, \Delta x, \Delta y$);
    od
    Omit_All_Ports($\mathcal{Q}_i$, $\mathcal{Q}_{omit_i}$);
    if composible then
        $\mathcal{Q}_i \leftarrow ComposeSides(\mathcal{Q}_i, \mathcal{Q}_q, \mathcal{Q}_i^c, \mathcal{Q}_q^c)$;
        $\mathcal{I}_i \leftarrow \mathcal{I}_i \cup I$;
    fi
    Compose $\leftarrow$ composible;
enddef
```

The function Compose simply takes too side sets $\mathcal{Q}_i^c$, $\mathcal{Q}_q^c$, makes sure that the sides are in the same coordinate system, computes the actual placement of the proposed instance, removes the omitted ports from the composition, and then syntactically verifies that these sides may connect. In order to add semantic information to the cell C_i, Pooh must note the connectivity of the composition and verify that the GDRs are maintained.

6.3 Composition Connectivity

Maintaining the connectivity information at the composition level has two important aspects: (1) the representation of the cell's connectivity and (2) the algorithms to deduce a cell's electrical node information. The representation of connectivity of paths was described in Chapter 4. The set $\mathbf{ic}_i$ represents the connectivity between the instances $\mathcal{I}_i$. Each internal connection indicates that two ports defined in separate cells are connected in this cell. Figure 6-5 illustrates examples of internal connections.

$$ic_1 = \langle I_1 : Q_4.o_1 \equiv I_2 : Q_2.o_5 \rangle$$
$$ic_2 = \langle I_1 : Q_1.o_1 \equiv I_3 : Q_3.o_6 \rangle$$
$$ic_3 = \langle I_2 : Q_1.o_9 \equiv I_3 : Q_3.o_7 \rangle$$

Internal connections are a general connectivity representation between the instances $I_{i,j}$ and $I_{i,k}$. The connections between these two instances occur in the cell C_i and not in the cells where the instance's ports are defined. An alternative method of representing this connectivity would be to add a path to the set $\mathcal{P}_i$ whose initial point is

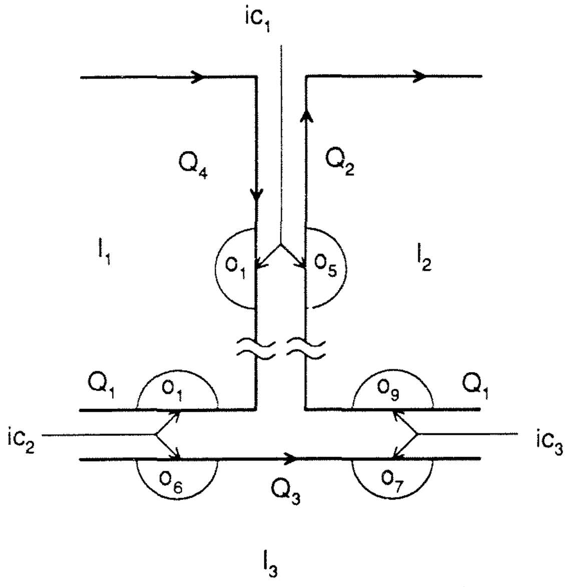


Figure 6-5: Internal connections

the port in $I_{i,j}$ and whose final point is the port in $I_{i,k}$. Since the two instances abut, the two ports occupy the same position. This zero length path is artificial and unnecessary, since the port connection is the only necessary piece of information. The number of internal connections defined is equal to the actual number of connected ports and not to the total number of defined ports, since it is possible for a particular composition to use a subset of the actual number of cell ports. Without the set $\mathbf{ic}_i$, the cell C_i would not include a representation of its internal connections.

Each cell port o has a node number within its original cell definition, denoted $node(o_q)$. The placement of the port is given by a Pooh point p_q that maintains a node N_q, as described in Chapter 4. The port node $node(o_q) \equiv N_q$. This node number is important simply because all connected points have an equivalent node number. Thus two ports o_q and o_r share a common signal if and only if $node(o_q) = node(o_r)$.

Node Propagation Algorithm

The node propagation algorithm for composition level cells uses the `equivalence` function introduced in Section 4.1. The following algorithm indicates how to propagate node information as each composition occurs between the current *state* of the cell C_i and a cell abstraction $E(C_q)$. Since each cell represents its total internal connectivity, it is also possible to design a "batch" node propagation algorithm that incorporates information about the entire set I_i.

The node propagation for an instance of $E(C_q)$ requires valid node numbers in C_i for all ports defined in the external interface $E(C_q)$, and that ports defined in $E(C_q)$, where $node(o) = node(o')$, have the same node number in C_i. The node numbers in the cell C_i are only valid in C_i and are distinct from the original node numbers of the ports in the cell C_q. The function `Ports_Equivalence` uses an array `cellnodes` to store the node numbers of the cell C_i. The indices for this array are the original node numbers of the ports in C_q. The array is a temporary mechanism to ensure that connected ports in C_q have an equivalent node number in C_i. The function is:

```
define Ports_Equivalence($I(E(C_q))$)
    for $Q_j \in \mathcal{Q}_q$ do
        for $o_k \in \mathcal{O}_j$ do
            if cellnodes[$node(o_k)$]$=0$ then
                cellnodes[$node(o_k)$] $\leftarrow$ *generate node number*
            fi
            $N_k$ $\leftarrow$ cellnodes[$node(o_k)$];
        od
    od
enddef
```

Pooh composition connects ports together where every port has a local node number. Node numbers are assigned locally within a cell, and are not valid globally. During the creation or modification of a cell, the array nodenumbers is used to maintain the cell connectivity. When two ports are connected, not only does an internal connection record the event, but the cell node numbers are modified accordingly. The function to connect two port sets is:

```
define Note_Port_Connections($\mathcal{O}_i$, $\mathcal{O}_q$)
    for $o_j \in \mathcal{O}_i, o'_j \in \mathcal{O}_q$ do
        *add an internal connection* $ic_{i,q}$
        equivalence($N_j$, $N'_j$)
    od
enddef
```

6.4 Composition GDR Verification

The final issue composition must address is Geometrical Design Rule (GDR) integrity at the composition level. The external interface $E(C_q)$ must fully represent the transistors, interconnection wires and contacts along the interface of the cell C_q, so that for every composition of $E(C_q)$, Pooh can guarantee the design rule integrity of the composite structure. The amount of information maintained in $E(C_q)$ must be both sufficient and necessary. The external cell is characterized as $[\mathcal{P}'_i, \mathbf{p}'_i, \mathcal{Q}_i]$ where the paths $\mathcal{P}'_i$ and points $\mathbf{p}'_i$ are derived from the sets $\mathcal{P}_i$, $\mathbf{p}_i$ and $\mathcal{I}_i$.

The External Interface

The amount of information maintained in the external interface depends on the restrictions enforced by the composition methodology. For example, one approach is to rigidly enforce a ring of half the maximum design rule distance between all non-interface elements and the edge of a cell [Mosteller 82]. This approach has the advantage that no design rule violations can exist at the composition level, and the obvious disadvantage that connections between cells are always dictated by the worst case design rule and not by the individual compositions. The other extreme is to allow arbitrary overlap between composed cells. In this case, a design rule violation may occur between geometry in any two cells anywhere in the entire design, and therefore a cell abstraction is not possible. This second approach is equivalent to using fully instantiated geometry and is unacceptable in designs of current day complexities.

The approach supported by Pooh is to derive the paths $\mathcal{P}_i'$ and $\mathbf{p}_i'$ within an "applicable" design rule spacing inside the cell sides $\mathcal{Q}_i$, to restrict all path center-lines and points to fall within or on the cell boundary $\mathcal{Q}_i$, and to restrict the paths whose path radius extend outside $\mathcal{Q}_i$ as described in the following discussion. Figure 6-6 illustrates examples of legal and illegal interface elements.

A path P_j center-line may occur along one or more of the cell edges if and only if: (1) the path radius is the minimum path radius $mr(T_j)$, or (2) for each and every composition involving a non-minimum radius path segment, an identical redundant path segment is defined in the cell C_q to which the cell C_i is being composed. An example of a minimum radius path occurring along the edge of a cell is path P_1 shown in the legal cell of Figure 6-6. A composition involving a cell whose definition includes this path is valid if the interaction between the two composible interface regions are GDR correct. An example of a non-minimum radius path whose center-line occurs along a cell edge is path P_2 in the legal cell of the figure. Pooh places some restrictions on the valid compositions of a cell containing a non-minimum path. In particular, given a cell C_i, where the paths $\mathcal{P}_i$ include a non-minimum radius path P whose center-line lies on the side Q_j, $Q_j \in \mathcal{Q}_i$, for each and every composition between this cell and a cell C_q, there must be an identical path P', $P' \in \mathcal{P}_q$. These paths, called merge paths, are single segment path that lie on one side. Multiple segment paths that lie on more than one side may always be split into multiple paths. When composition occurs these two paths P and P' are merged. A corollary to this restriction is that the geometry of a non-minimum radius path may either extend beyond the cell boundary by the minimum radius, or it must lie on the cell boundary. If the path lies on the cell boundary, the path ends may only extend $mr(T_k)$ outside the cell boundary. Path merging is often used for power, ground and other global buses. The capability of composing cells that include paths whose radius is not minimum is important for the same reason ports are important and is treated in a similar manner.

Path merging allows the designer to construct design rule correct cells. Often large buses are shared between parts of the design, and without path merging each cell could not contain a definition of the buses it used. However, allowing arbitrary overlap between cells would destroy the ability to derive a cell abstraction. Path merging allows wires to be defined in the cells where they are used, and still allows Pooh to guarantee GDR integrity at the composition level. In Figure 6-6, path P_1, in the illegal cell, is illegal because its end points do not lie on a single side.

The paths and points $\mathcal{P}_i'$ and $\mathbf{p}_i'$ of the external interface $E(C_i)$ are the Pooh elements of GDR interest along the the edges $\mathcal{Q}_i$. Let D denote the largest minimum design rule distance. If we shrink the polygon described by $\mathcal{Q}_i$ by a distance D, then any Pooh element residing inside the new polygon cannot possibly introduce a design rule violation by the definition of D. Thus all the GDR-interesting elements pass through the D wide region defined by the shrunk polygon and the original polygon $\mathcal{Q}_i$. Figure 6-7 indicates the elements of GDR interest from the cell in Figure 6-1. Since the placement of each side $Q_j \in \mathcal{Q}_i$ is defined by a line equation $L_j(x,y)$, a parallel line $L_j'(x,y)$, D

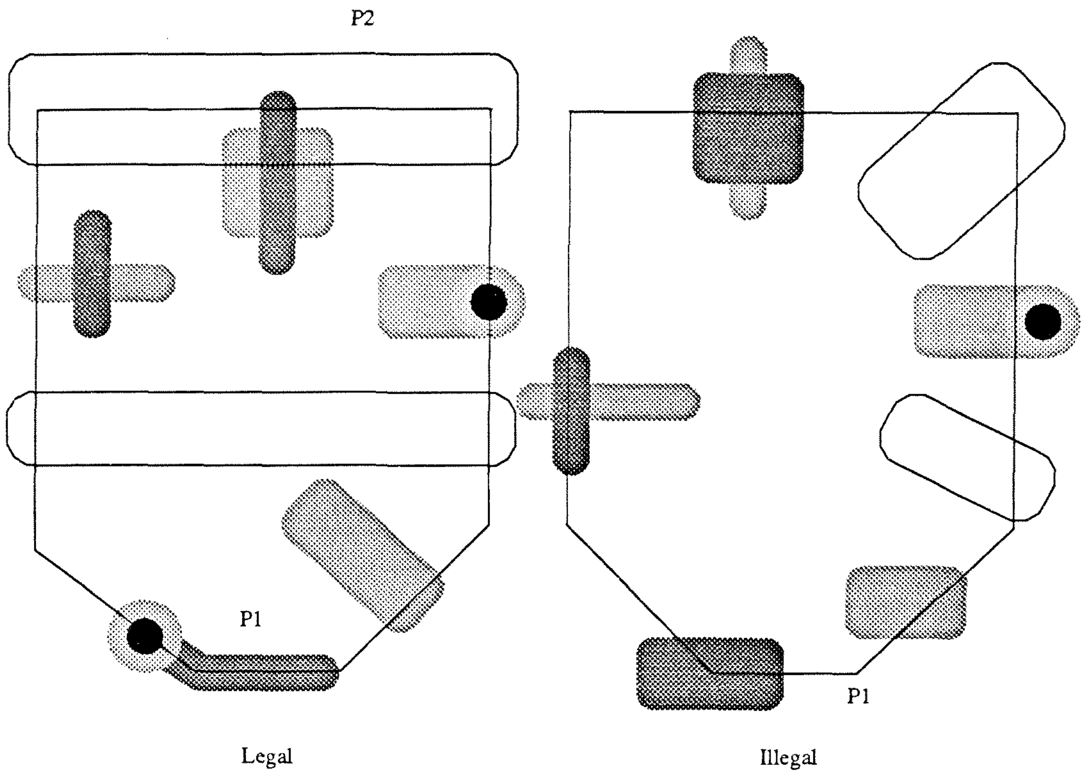


Figure 6-6: Legal and Illegal Cells

distance closer to the cell center is:

$$
\begin{aligned}
A' &= A \\
B' &= B \\
C' &= C + D. \qquad (6\text{-}1)
\end{aligned}
$$

The points $\mathbf{p}_i'$ of external GDR interest are the subset of $\mathbf{p}_i$ that occur between the lines $L_j(x,y)$ and $L_j'(x,y)$, where $Q_j \in \mathcal{Q}_i$. The subset $\mathbf{p}_i'$ may easily be derived simply by testing all point $p_k \in \mathbf{p}_i$ for inclusion. A function to determine whether a point is in the interface region along the side Q_j, using (6-1), is:

```
define Interface_Point($p_k$, $Q_j$, $D$):  boolean
    Interface_Point $\leftarrow (A_j \times x_k + B_j \times y_k + C_j) \leq 0$ &
            $(A_j \times x_k + B_j \times y_k + (C_j + D)) \geq 0$;
enddef
```

The paths of GDR interest in the set $\mathcal{P}_i'$ are composed from the segments of GDR interest. Each path $P_k \in \mathcal{P}_i$ is composed of a GDR correct set of segments $\mathcal{S}_k$. Only the segments of interface interest need be included in the interface description — not the entire path of which these segments are members. Including incomplete paths could create apparent internal GDR violations, but at the composition level Pooh does not heed internal GDR violations. In fact Pooh does not even need to keep the complete definition of interesting interface segments; instead for each interesting interface segment s_k, Pooh

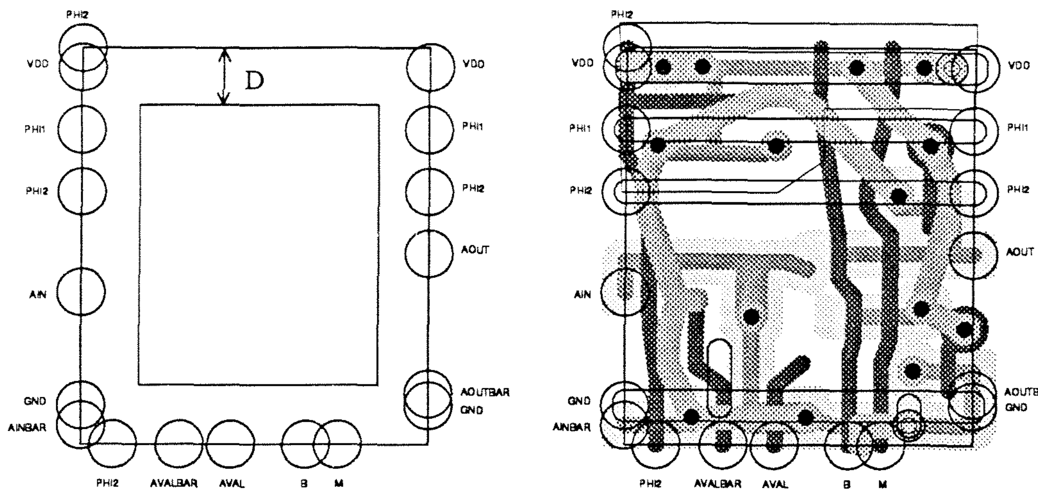


Figure 6-7: GDR Interesting Paths and Points

calculates a segment s'_k that lies completely in the GDR interface region. Figure 6-7 illustrates a GDR interface region with a width of D.

Given a segment s_k that intersects the modified side $L'_j(x,y)$, it is possible to calculate the point of intersection, using equation (3-4). The point of intersection is:

$$\det = B_k A_j - A_k B_j$$
$$p_{int}(s_k, L_j, D) = \Big((B_j C_k - B_k(C_j + D))/\det,\ (A_k(C_j + D) - A_j C_k)/\det\Big).$$

Pooh derives a (possibly empty) set of paths P_j^c within the interface region from a single path P_j by detecting each consecutive sequence of segments contained in the region, and optionally computing the sequence end points. Since Pooh ensures that all center-lines and points are within the polygon defined by Q_i, it does not need to detect intersection between a segment and a side. The following algorithm extracts this set of paths P_j^c from a path P_j within a distance D from the side Q_j:

```
define arc_intersection($s_1$, $L_2(x,y)$): point
    the point where the arc of $s_1$ intersects with $L_2(x,y)$
enddef
define all_inside($s$, $Q$, $D$):boolean
    all_inside $\leftarrow$ Interface_Point($bp(s)$, $Q$, $D$) &
            Interface_Point($ep(s)$,$Q$, $D$);
enddef
```

```
define outside_interface($p_k$, $Q_j$, $D$): boolean
    outside_interface ← $(A_j \times x_k + B_j \times y_k + (C_j + D)) \leq 0$;
enddef
define all_outside($s$, $Q$, $D$):boolean
    all_outside ← outside_interface($bp(s)$, $Q$, $D$) &
            outside_interface($ep(s)$,$Q$, $D$);
enddef
define cross_in($s$, $Q$, $D$):boolean
    cross_in ← outside_interface($bp(s)$, $Q$, $D$) &
            Interface_Point($ep(s)$,$Q$, $D$);
enddef
define cross_out($s$, $Q$, $D$):boolean
    cross_out ← Interface_Point($bp(s)$, $Q$, $D$) &
            outside_interface($ep(s)$,$Q$, $D$);
enddef
define Interface_Path($P_k$,$Q_j$,$D$): set of paths
    var $\mathcal{P}_Q$: set of paths;
        $P'$: path;
    $\mathcal{P}_Q \leftarrow \emptyset$; $P'$ ← nil;
    for $s_t \in S_k$ do
        if all_outside($s_t$, $Q_j$, $D$) then
            if $exists(P')$ then
                addarcsegment(arc_intersection($s_{t-1}$, $L_j(x,y)$))
                $\mathcal{P}_Q \leftarrow \mathcal{P}_Q \cup P'$;  $P'$ ← nil;
            fi
        else if all_inside($s_t$, $Q_j$, $D$) then
            if $\sim exists(P')$ then $P'$ ← newpath($P_k$) fi
            addsegment($s_t$);
        else if cross_in($s_t$, $Q_j$, $D$) then
            $P'$ ←newpath($P_k$);
            addnewseg($p_{int}(s_t, L'_j, D)$, $r_t$, $p_t$);
        else if cross_out($s_t$, $Q_j$, $D$) then
            if $\sim exists(P')$ then $P'$ ← newpath($P_k$) fi
            addnewsegment($bp(s_t)$, 0, $p_{int}(s_t, L'_j, D)$);
            $\mathcal{P}_Q \leftarrow \mathcal{P}_Q \cup P'$;  $P'$ ← nil;
        fi
    od
    if $exists(P')$ then
        $\mathcal{P}_Q \leftarrow \mathcal{P}_Q \cup P'$;  $P'$ ← nil;
    fi
    Interface_Path ← $\mathcal{P}_Q$;
enddef
```

A cell abstraction $E(C_i)$ of a leaf cell C_i is derived by finding all points $p_k \in \mathbf{p}_i$ and all paths $P_l \in \mathcal{P}_i$ within some distance D of the cell sides $\mathcal{Q}_i$. In fact the distance D does not have to be a single value representing the maximum GDR distance. Since both the `Interface_Point` and `Interface_Path` algorithms evaluate based on a distance D, the parameter may vary based on the point and path type. Therefore a path or a point remains in the interface region if and only if it is within the maximum GDR rule applicable to this type of element. Figure 6-8 illustrates the paths and points contained in the interface $E(C_i)$ for the multiplier cell shown in Figure 6-1.

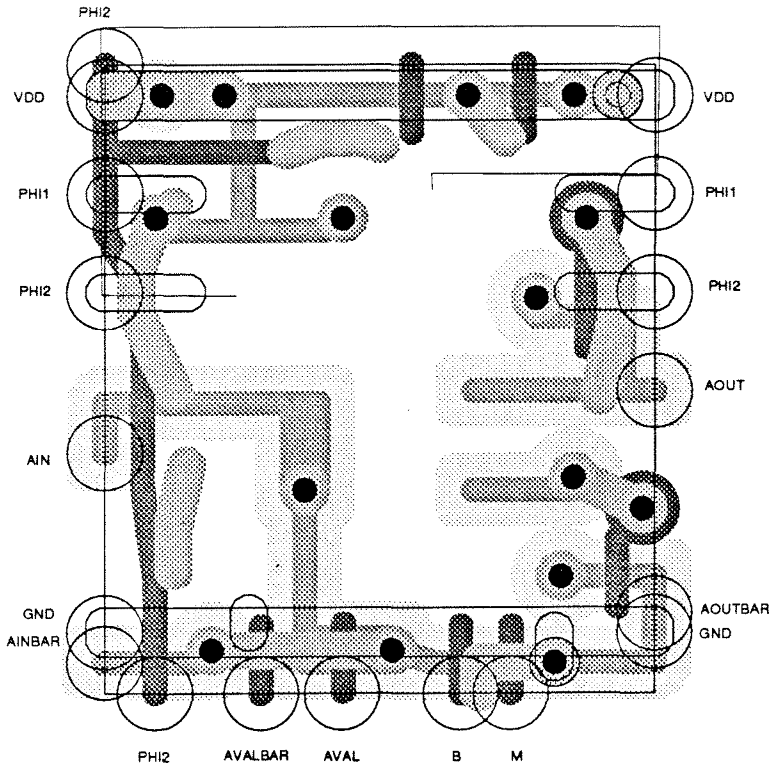


Figure 6-8: Actual GDR Interface

The GDR interface for a hierarchical cell is incrementally derived by incorporating the relevant GDR interface elements of each of the instances $I_j \in I_i$. Each time an instance is added to the cell C_i, the paths P_i and the $\mathbf{p}_i$ are updated to include transformed paths and points contained in the cell abstraction $E(cell(I_j))$.

The algorithm to take a cell C_i and create the cell interface $E(C_i)$ is:

```
define Cell_interface($C_i$) :   *cell interface*
    var $P_Q$, $P$ :   *set of paths;*
        $\mathbf{p}_Q$ :   *set of points;*
        in_interface:  boolean;
    $P_Q \leftarrow \emptyset$;  $\mathbf{p}_Q \leftarrow \emptyset$;
    for $P_j \in \mathcal{P}_i$ do
        in_interface ← false;
        for $Q \in \mathcal{Q}_i$ AND ~ in_interface do
            $\mathcal{P}$ ← Interface_Path($P_j$, $Q$, *maximum design rule distance for* $T_j$)
            if $\mathcal{P} \neq \emptyset$ then
                $\mathcal{P}_Q \leftarrow \mathcal{P}_Q \cup \mathcal{P}$;   in_interface ← true;
            fi
        od
    od
    for $p_j \in \mathbf{p}_i$ do
        in_interface ← false;
        for $Q \in \mathcal{Q}_i$ AND ~ in_interface do
            $\mathbf{p}$ ← Interface_Point($p_j$, $Q$, *maximum design rule distance for* $T_j$)
            if $\mathbf{p} \neq \emptyset$ then
                $\mathbf{p}_Q \leftarrow \mathbf{p}_Q \cup \mathbf{p}$;   in_interface ← true;
            fi
        od
    od
    Cell_interface ← $[\mathcal{P}_Q, \mathbf{p}_Q, \mathcal{Q}_i]$;
enddef
```

The Composition Algorithm

A Pooh cell definition contains internal topology information in the form of paths, points and instances, and external interface information in the ports and sides. Each time a composition occurs, Pooh checks that the composition does not introduce any GDR violations. Then, the next *state* of the cell currently being defined not only inherits a new side specification, but incorporates additional internal connectivity and updates its path and point sets to include new GDR interface information. This definition of compose guarantees that at all levels in the hierarchy each cell is well-formed. The updated `Compose` function is:

```
define Compose($C_i$, $E(C_q)$, $\mathcal{Q}_i^c$, $\mathcal{Q}_q^c$, $\mathcal{Q}_{omit_i}$, $\mathcal{Q}_{omit_q}$, orient:  transform):
                    boolean
    var composible:  boolean;
        $I$:  instance;
    Ports_Equivalence($E(C_q)$);
    composible ← true;
    $I$ ← *newinstance*($E(C_q)$);
    *orientCellSides*($I(E(C_q))$, $\mathcal{Q}_q^c$, orient);
    *make sure* $\mathcal{Q}_i$ *and* $\mathcal{Q}_q$ *are clockwise polygons*
    for $Q_j \in \mathcal{Q}_i^c, Q'_j \in \mathcal{Q}_q^c$ do
        Omit_Ports($Q_j$, $\mathcal{Q}_{omit_i}$);
        Omit_Ports($Q'_j$, $\mathcal{Q}_{omit_q}$);
        if $A_j \neq -A'_j$ OR $B_j \neq -B'_j$ OR $size(\mathcal{O}_j) \neq size(\mathcal{O}'_j)$ then
            composible $\leftarrow$ false fi
        if $j = 1$ then $\Delta x, \Delta y \leftarrow$ translation($\mathcal{O}_j$, $\mathcal{O}'_j$)
        else if $\Delta x, \Delta y \neq$ translation($\mathcal{O}_j$, $\mathcal{O}'_j$) then composible $\leftarrow$ false
        fi
        apply_translation($\mathcal{O}'_j, \Delta x, \Delta y$);
        Note_Port_Connections($\mathcal{O}_j$, $\mathcal{O}'_j$);
        composible $\leftarrow$ composible AND Connect_Ports($\mathcal{O}_j$, $\mathcal{O}'_j$) AND
                GDR_check($\mathbf{p}_i$, $\mathcal{P}_i$, $\mathbf{p}'_q$, $\mathcal{P}'_q$);
        $C'_j \leftarrow C'_j - (A'_j \times \Delta x + B'_j \times \Delta y)$;
    od
    for $Q \in (\mathcal{Q}_q - \mathcal{Q}_q^c)$ do
        Omit_Ports($Q$, $\mathcal{Q}_{omit_q}$);
        apply_translation($\mathcal{O}, \Delta x, \Delta y$);
    od
    Omit_All_Ports($\mathcal{Q}_i$, $\mathcal{Q}_{omit_i}$);
    if composible then
        $\mathcal{Q}_i \leftarrow ComposeSides(\mathcal{Q}_i, \mathcal{Q}_q, \mathcal{Q}_i^c, \mathcal{Q}_q^c)$;
        $\mathcal{I}_i \leftarrow \mathcal{I}_i \cup I$;    $\mathcal{P}_i \leftarrow \mathcal{P}_i \cup \mathcal{P}'_q$;    $\mathbf{p}_i \leftarrow \mathbf{p}_i \cup \mathbf{p}'_q$;
    fi
    Compose $\leftarrow$ composible;
enddef
```

The algorithm presented in this chapter places some restrictions on how composition occurs. The function Compose only detects GDRs — and invalidates the composition if a GDR violation is detected. A simple extension is to modify the translation factors Δx and Δy based on the minimum design rule distance of the limiting two elements: one in the current *state* of C_i, and the other in the instance of $E(C_q)$. This composition no longer requires strict abutment but rather places two instances some small distance apart. Such a composition function must also take into account the minimum connected spacing of the paths and points along the connecting sides.

Simple river routing is another method of composing two cell's together. Pooh does not support stretching [Rowson 80], since it assumes that if a designer wanted a cell to be larger, the cell would have been designed with a larger pitch. Occasionally the need might arise to connect two cells with mis-aligned ports. Martin Tompa developed an optimal river routing algorithm [Tompa 80] using Pooh-like wires for interconnection. Each wire consists of line segments and arcs. The Pooh composition methodology supports Tompa routing as one form of composition.

The Compose function can be accommodating when it comes to GDR spacing, but it cannot extend similar help to logical errors. If a logical error occurs between two connecting ports it is similar to a syntax error in a programming language, and similarly must be fixed before a valid composition may continue.

The composition algorithm presented in this chapter allows Pooh to represent large designs by supporting a hierarchical design methodology. The key to the methodology is to derive a cell abstraction $E(C_q)$ from the original cell C_q. This abstraction fully represents the external interface of a cell while removing the internal details. The Compose function presents a way of combining two or more cell abstractions to form a new abstraction.

6.5 Proof of Closure Under Composition

Cells with only geometry and not any instances, i.e., leaf cells, are generally small and therefore easy to prove correct, both logically and geometrically. The information content of these cells is small enough that a person can normally guarantee these cells correct by examination. Experience has shown that the overwhelming majority of problems occur at the interface between cells. Thus the key to the correctness of a large system is proving the composition of two or more cells correct.

Definition 1. Given the j^{th} side, Q_j, in the set of n sides $\mathcal{Q}$, the two end points of the side, based on equation (3-4) in Chapter 3, are:

$$sideBp(Q_j) = \begin{cases} p_{int}(L_j, L_{j-1}), & j > 1 \\ p_{int}(L_j, L_n), & j = 1 \end{cases}$$

$$sideEp(Q_j) = \begin{cases} p_{int}(L_j, L_{j+1}), & j < n \\ p_{int}(L_j, L_1), & j = n. \end{cases}$$

Definition 2. Given a path P_k and a set of sides $\mathcal{Q}$, the path is a *merge path* if and only if: (a) the path radius $W_k/2 > mr(T_k)$, (b) the path consists of a single segment,[†] (c) the path centerline lies on a line segment $L_j(x,y)$, where $Q_j \in \mathcal{Q}$, and (d) the path end points are between the end points $sideBp(Q_j)$ and $sideEp(Q_j)$, inclusively, and (d) the path does not extend more than $mr(T_k)$ outside any other side, $Q_j' \in \mathcal{Q}$, $Q_j' \neq Q_j$. [Section 6.4]

Definition 3. A pooh cell C is *well-formed* if and only if:

(1) There is a non empty set of sides $\mathcal{Q}$ that forms a clockwise simply-connected polygon and defines the cell boundary. The sides $\mathcal{Q}$ bound all path center-lines in the set $\mathcal{P}$ and points contained in the set $\mathbf{p}$. [Section 6.1]

(2) There is a (possibly empty) set of ports $\mathcal{O}$ distributed over the cell sides $\mathcal{Q}$, i.e., given m sides $\mathcal{O} = \mathcal{O}_1 \cup \mathcal{O}_2 \cup \ldots \cup \mathcal{O}_m$, that defines the cell's external connection points. The ports $\mathcal{O}_j$ associated with each side $Q_j \in \mathcal{Q}$ are sorted, and every port $o \in \mathcal{O}_j$ is both on the line that defines the side $L_j(x,y)$, and between the two end points $sideBp(Q_j)$ and $sideEp(Q_j)$, inclusively. [Section 6.1]

[†] Multiple segment paths may always be broken into multiple paths. This restriction simplifies the description without any loss of generality.

(3) There is a path $P_k \in \mathcal{P}$ or a point $p_l \in \mathbf{p}$ for every geometrical element in the fully instantiated geometry of C within the maximum applicable GDR distance from the sides $\mathcal{Q}$. [Section 6.4]

(4) For every path $P_k \in \mathcal{P}$, either: (a) the path geometry does not extend more than the minimum radius $mr(T_k)$ outside the cell boundary $\mathcal{Q}$, or (b) the path is a merge path.

(5) The fully instantiated geometry of C is GDR correct.

Definition 4. Let $InterfacePoint(p, \mathcal{Q}, D)$ denote a function that indicates whether or not a point is within a distance D of the polygon described by $\mathcal{Q}$. Given a set of points $\mathbf{p}$, the *external interface points* $\mathbf{p}'$ is the set of points, such that for every point $p \in \mathbf{p}'$, this point is also a member of the original set, or $p \in \mathbf{p}$, and for D equal to the maximum applicable GDR to p, the function $InterfacePoint(p, \mathcal{Q}, D)$ is true; and for every point not in the set $\mathbf{p}'$, or $p' \in (\mathbf{p} - \mathbf{p}')$, the function $InterfacePoint(p', \mathcal{Q}, D)$ is false.

Definition 5. Let $\mathcal{P}_{interface}(P, \mathcal{Q}, D)$ denote the (possibly empty) set of paths, derived from the set $\mathcal{P}$, within a distance D of the polygon described by $\mathcal{Q}$. Given a set of z paths $\mathcal{P}$, the *external interface paths* $\mathcal{P}' = \bigcup_i \mathcal{P}_{interface}(P_i, \mathcal{Q}, D), P_i \in \mathcal{P}$ where D is the maximum applicable GDR to P.

Definition 6. Given a well-formed cell C, characterized as $[\mathcal{P}, \mathbf{p}, \mathcal{Q}, \mathcal{I}, \mathbf{ic}]$, the *external interface* $E(C)$ of the cell is characterized as $[\mathcal{P}', \mathbf{p}', \mathcal{Q}]$. The external interface paths $\mathcal{P}'$ and the external interface points $\mathbf{p}'$ are those of GDR interest along the cell sides $\mathcal{Q}$, and are derived directly from the sets $\mathcal{P}$ and $\mathbf{p}$. [Section 6.4]

Definition 7. A *composible side sequence* $\mathcal{Q}^c$ of size w is a connected sequence of sides, or $\mathcal{Q}^c \equiv \{Q_1, Q_2, \ldots, Q_w\}$. [Section 6.2]

Definition 8. The *port size* of $\mathcal{O}_j$ on the side Q_j, is the number of ports $o \in \mathcal{O}_j$.

Definition 9. Let $PortType(o)$ denote the type of the port o, and $PortConnect(o)$ denote the set of valid port types to which a port o may connect. Then two ports o and o' are *connectable* if and only if $PortType(o) \in PortConnect(o')$ and $PortType(o') \in PortConnect(o)$. [Section 6.2]

Definition 10. A *translation factor* $\Delta x, \Delta y$ maps a side Q with m ports into a side Q' with m ports, as follows [Section 6.1]:

$$\begin{aligned} L'(x,y) &= [A, B, C - (A \times \Delta x + B \times \Delta y)] \\ T(o) &= [x(o)+\Delta x, y(o)+\Delta y] \\ \mathcal{O}' &= \{T(o_1), T(o_2), \ldots T(o_m)\}. \end{aligned}$$

Note: The orientation and mirroring of a cell are specified by the user, and incorporated before the composition occurs, since without this information the composition operation is ambiguous.

Definition 11. The *length* of the side Q_j is:

$$Length(Q_j) = |sideEp(Q_j) - sideBp(Q_j)|.$$

Definition 12. The *reverse side* of a side Q with a line equation $L(x,y)$ and a set of m ports $\mathcal{O}$ is:

$$\begin{aligned} Reverse(\mathcal{O}) &= \{o_m, o_{m-1}, \ldots, o_2, o_1\} \\ Reverse(L(x,y)) &= -L(x,y) \\ Reverse(Q) &= [Reverse(L(x,y)), Reverse(\mathcal{O})]. \end{aligned}$$

Definition 13. The set of *reverse sides* of a set of m sides $\mathcal{Q}$ is:

$$Reverse(Q) = \{Reverse(Q_m), Reverse(Q_{m-1}), \ldots, Reverse(Q_2), Reverse(Q_1)\}.$$

The following definition assumes that: (a) there is a function, denoted *Points-Equivalence*$(\mathcal{O}_1, \mathcal{O}_2)$, that takes two sets of ports and notes their connectivity, as described in Chapter 4 and Section 6.3, and (b) there is a a function, denoted *GDRCorrect*$(\mathcal{P}_1, \mathbf{p}_1, \mathcal{P}_2, \mathbf{p}_2)$, that determines whether or not a GDR violation occurs between the geometry defined in the sets of paths and points $\mathcal{P}_1$ and $\mathbf{p}_1$, and the geometry defined by the sets $\mathcal{P}_2$ and $\mathbf{p}_2$, using the approach described in Chapter 5 and Section 6.4.

Definition 14. Given two cells C_1 and C_2, and two composible side sequences $\mathcal{Q}_1^n$ of size a, and $\mathcal{Q}_2^n$ of size b, there is a *valid composition* of these two cells if:

(1) The size of the two composible side sequences is equal, or $a = b$,

(2) The port size of o_j is equal to the port size of o'_j, and the side lengths are equal, or $Length(Q_j) = Length(Q'_j)$, for $Q_j \in \mathcal{Q}_1^n$, $Q'_j \in \mathcal{Q}_2^n$.

(3) The ports o and o' are connectable, for $o \in \mathcal{O}_j$, $o' \in Reverse(\mathcal{O}'_j)$ on the sides $Q_j \in \mathcal{Q}_1^n$, $Q'_j \in \mathcal{Q}_2^n$.

(4) There is a translation factor $\Delta x, \Delta y$ that maps the composible sides $\mathcal{Q}^n$ into the composible sides $\mathcal{Q}^c$; and for $Q_j \in \mathcal{Q}_1^c$, $Q'_j \in Reverse(\mathcal{Q}_2^c)$, the line equations are equal, or $L_j(x,y) = Reverse(L'_j(x,y))$; and the ports occupy the same position, or for $o \in \mathcal{O}_j$, $o' \in Reverse(\mathcal{O}'_j), x(o) = x(o')$ and $y(o) = y(o')$.

(5) The intersection of the polygon described by $\mathcal{Q}_1$ and the polygon described by the translated set of sides $\Delta x, \Delta y(\mathcal{Q}_2)$, minus the composible side sequence is two points, or the intersection of $(\mathcal{Q}_1 - \mathcal{Q}_1^c) \cap (\Delta x, \Delta y(\mathcal{Q}_2) - \mathcal{Q}_2^c)$ is exactly two points. [Notice that if the polygons described by $\mathcal{Q}_1$ and $\mathcal{Q}_2$ overlap, the intersection between these two polygons is greater than two points.]

(6) There is a (possibly empty) set of merge paths $\mathcal{P}_1^c$ in C_1, and another set $\mathcal{P}_2^c$ in C_2, where the size of $\mathcal{P}_1^c$ is equal to the size of $\mathcal{P}_2^c$, and for every path $P_k \in \mathcal{P}_1^c$ on the side Q_j, there is a corresponding path $P'_k \in \mathcal{P}_2^c$ on the side Q'_j. Given these two paths, the path type, path width and two end points are equal, or $T_k = T'_k, W_k = W'_k, p_0 = p'_0$, and $p_1 = p'_1$.

(7) Given that the nodes of the ports on the two composible side sequences are set as equal, there is not a GDR violation in the geometry derived to represent the cell in the external interfaces $E(C_1)$ and $E(C_2)$; or given $PointsEquivalence(\mathcal{O}_j, \mathcal{O}'_j)$, for $\mathcal{Q}_j \in \mathcal{Q}_1^c$, $\mathcal{Q}'_j \in \mathcal{Q}_2^c$, then $GDRCorrect(\mathcal{P}'_1 - \mathcal{P}_1^c, \mathrm{p}'_1, \mathcal{P}'_2 - \mathcal{P}_2^c, \mathrm{p}'_2)$.

Notice that for a valid composition, a GDR violation cannot be introduced by the merge paths in the sets $\mathcal{P}_1^c$ and $\mathcal{P}_2^c$, since for a merge path in the set P either (a) there is an equivalent corresponding merge path in the other set that will be merged with P during composition, and therefore the path already exists in the cell geometry and is GDR correct from the definition of well-formed, or (b) there is not an equivalent path in the other set in which case this composition is not valid.

It is worth noting that given a clockwise simply connected polygon $\mathcal{Q}$ composed of a closed sequence of directed line segments, $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_n\}$, and a composible side sequence $\mathcal{Q}^c$ that is a subset of these polygonal sides, or $\mathcal{Q}^c \subset \mathcal{Q}$, there is always a partitioning of the polygon $\mathcal{Q}$ into $\mathcal{Q}^c$ and $\mathcal{Q} - \mathcal{Q}^c$. These two partitions intersect at exactly two points.

Definition 15. Given two sets of sides: $\mathcal{Q}$ of size n and $\mathcal{Q}'$ of size m, and two composible side sequences of size w, $Q_{j\ldots(j+w)}$ and $Q'_{k\ldots(k+w)}$, the *composition side set* $\mathcal{Q}_{compose}$ that results from superimposing the two composible side sequences is [Section 6.2]:

$$Merge(o_{1\ldots y}, o'_{1\ldots z}) = \{o_1, o_2, \ldots o_y, o'_1, o'_2, \ldots, o'_z\}$$

$$Merge(Q, Q') = \begin{cases} [L(x,y), Merge(\mathcal{O} \cup \mathcal{O}')], & A = A' \wedge B = B' \wedge C = C' \\ \{Q, Q'\}, & \text{otherwise.} \end{cases}$$

$$LastSide(Q_j, n) = \begin{cases} Q_{j-1}, & j > 1 \\ Q_n, & j = 1 \end{cases}$$

$$NextSide(Q_j, n) = \begin{cases} Q_{j+1}, & j < n \\ Q_n, & j = n \end{cases}$$

$$\begin{aligned} \mathcal{Q}_{compose} = {} & (\mathcal{Q} - \{LastSide(Q_j, n), Q_j, \ldots Q_{j+w}, NextSide(Q_{j+w})\}) \\ & \cup Merge(LastSide(Q_j, n), NextSide(Q'_{k+w}, m)) \\ & \cup Merge(LastSide(Q'_{k-1}, m), NextSide(Q_{j+w}, n)) \\ & \cup (\mathcal{Q}' - \{LastSide(Q'_k, m), Q'_k, \ldots Q'_{k+w}, NextSide(Q'_{k+w})\}). \end{aligned}$$

Lemma 1. Given two clockwise simply connected polygon sides $\mathcal{Q}_1$ and $\mathcal{Q}_2$, and two composible side sets $\mathcal{Q}_1^c$ and $\mathcal{Q}_2^c$ of size w, where for $Q_j \in \mathcal{Q}_1^c$ and $Q'_j \in Reverse(\mathcal{Q}_2^c)$, $Length(Q_j) = Length(Q'_j)$ and $L_j(x,y) = Reverse(L'_j(x,y))$, and if the intersection $(\mathcal{Q}_1 - \mathcal{Q}_1^c) \cap (\mathcal{Q}_2 - \mathcal{Q}_2^c)$ is exactly two points, then $\mathcal{Q}_{compose}$ is a clockwise simply connected polygon.

Proof: The two sets of line segments $\mathcal{Q}_1 - \mathcal{Q}_1^c$ and $\mathcal{Q}_1^c$ intersect at exactly two points, from the definition of simply connected polygons, as do the two sets $\mathcal{Q}_2 - \mathcal{Q}_2^c$ and $\mathcal{Q}_2^c$. Since $\mathcal{Q}_1^c$ and $Reverse(\mathcal{Q}_2^c)$ define the same sequence of line segments, the sets $Q_1 - \mathcal{Q}_1^c$ and $Q_2 - \mathcal{Q}_2^c$ intersect at two points. Since both of these sequences are clockwise, the set defined as $\mathcal{Q}_{compose}$ creates a simply connected polygon by concatenating the two sequences $Q_1 - \mathcal{Q}_1^c$ and $Q_2 - \mathcal{Q}_2^c$, and merging equivalent line segments. ■

There are several important properties of the boundary and the interior points of the polygons $\mathcal{Q}_1$, $\mathcal{Q}_2$ and $\mathcal{Q}_{compose}$ of Lemma 1:

(1) The intersection of the two polygons $\mathcal{Q}_1$ and $\mathcal{Q}_2$, or $\mathcal{Q}_1 \cap \mathcal{Q}_2$, is the sequence of line segments $\mathcal{Q}^c \equiv \mathcal{Q}_1^c \equiv \mathcal{Q}_2^c$, from the definition of the polygons.

(2) The interior points of the polygons $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are also interior points of the polygon $\mathcal{Q}_{compose}$, from the definition of the boundary of $\mathcal{Q}_{compose}$.

(3) The boundary points of $\mathcal{Q}_1$ occurring along the sequence $\mathcal{Q}_1 - \mathcal{Q}_1^c$ and the boundary points of $\mathcal{Q}_2$ occurring along the sequence $\mathcal{Q}_2 - \mathcal{Q}_2^c$ are also boundary points of $\mathcal{Q}_{compose}$, from the definition of the $\mathcal{Q}_{compose}$ boundary.

(4) The points on the line segment sequence $\mathcal{Q}^c$, minus the two end points of the sequence, are interior points of $\mathcal{Q}_{compose}$ from the definition of simple polygons.

Definition 16. Given two sets of paths, $\mathcal{P}'_1$ and $\mathcal{P}'_2$, the *composition path set* $\mathcal{P}_{compose} \equiv \mathcal{P}'_1 \cup \mathcal{P}'_2$.

Definition 17. Given two sets of points, $\mathbf{p}'_1$ and $\mathbf{p}'_2$, the *composition point set* $\mathbf{p}_{compose} \equiv \mathbf{p}'_1 \cup \mathbf{p}'_2$.

Definition 18. The polygon described by the set of sides $\mathcal{Q}$ *bounds* a point p if and only if either: (a) the point p is an interior point of the polygon $\mathcal{Q}$, or (b) the point p is a boundary point of the polygon $\mathcal{Q}$.

Lemma 2. Given the side sets $\mathcal{Q}_1$ and $\mathcal{Q}_2$, from Lemma 1, the path sets $\mathcal{P}_1$ and $\mathcal{P}_2$, and the point sets $\mathbf{p}_1$ and $\mathbf{p}_2$, if the polygon $\mathcal{Q}_1$ bounds all path center-lines $P \in \mathcal{P}_1$ and points $p \in \mathbf{p}_1$, and the polygon $\mathcal{Q}_2$ bounds all path center-lines $P' \in \mathcal{P}_2$ and points $p' \in \mathbf{p}_2$, then the polygon described by $\mathcal{Q}_{compose}$ bounds the path center-lines $P \in \mathcal{P}_{compose}$, and the points $p \in \mathbf{p}_{compose}$.

Proof: The area bounded by the polygon $\mathcal{Q}_{compose}$ is the area bounded by the union of the two polygons $\mathcal{Q}_1$ and $\mathcal{Q}_2$, from the definition of $\mathcal{Q}_{compose}$. Therefore it is not possible to find a point or a path that is bounded by $\mathcal{Q}_1$ or bounded by $\mathcal{Q}_2$ that is not bounded by $\mathcal{Q}_{compose}$. ■

Lemma 3. Given the two side sets $\mathcal{Q}_1$ and $\mathcal{Q}_2$, from Lemma 1, if the ports $o \in \mathcal{O}_j$ are in sorted order along each side $Q_j \in \mathcal{Q}_1$ and the ports $o \in \mathcal{O}'_j$ are in sorted order along each side $Q'_j \in \mathcal{Q}_2$, then the ports $o \in \mathcal{O}_i$ are in sorted order along each side $Q_i \in \mathcal{Q}_{compose}$.

Proof: There are two types of sides $Q_i \in \mathcal{Q}_{compose}$ included in the definition of $\mathcal{Q}_{compose}$. The first type of sides is mapped directly from the sets $\mathcal{Q}_1$ or $\mathcal{Q}_2$. The ports on this type of side are sorted by assumption. The second type of sides is a side defined by $Merge(Q, Q')$ and only occurs when one of the two points of intersection between the two side sets $Q_1 - \mathcal{Q}_1^c$ and $Q_2 - \mathcal{Q}_2^c$ is the end point of the side Q and the beginning point of Q', or $p \equiv sideEP(Q) \equiv sideBp(Q')$. If the equation of the lines are not equal, then these two sides are mapped into $\mathcal{Q}_{compose}$ unchanged, and are sorted by assumption. Otherwise the two sets $\mathcal{O}$ and $\mathcal{O}'$ are concatenated. Since both polygons are clockwise by assumption, and the ports $\mathcal{O}_j$ occur on the line before the point p and the ports $\mathcal{O}'_j$ occur on the same line after the point p, this concatenation is sorted. ■

Lemma 4. Given the side sets $\mathcal{Q}_1$ and $\mathcal{Q}_2$ from Lemma 1, where every port $o \in \mathcal{O}_j$, on the sides $Q_j \in \mathcal{Q}_1$ is both on the line $L_j(x, y)$ and between the two end points $sideBp(Q_j)$ and $sideEp(Q_j)$, and every port $o \in \mathcal{O}'_j$, on the sides $Q'_j \in \mathcal{Q}_2$, is both on the line $L'_j(x, y)$ and between the two end points $sideBp(Q'_j)$ and $sideEp(Q'_j)$, then every port, $o \in \mathcal{O}_i$ on the sides $Q_i \in \mathcal{Q}_{compose}$ is on the line $L_i(x, y)$ and between the two end points $sideBp(Q_i)$ and $sideEp(Q_i)$, inclusively.

Proof: There are two types of sides $Q_i \in \mathcal{Q}_{compose}$ included in the definition of $\mathcal{Q}_{compose}$. The first type of sides is mapped directly from the sets $\mathcal{Q}_1$ or $\mathcal{Q}_2$. The ports on this type of side are both on line and between the two side end points by assumption. The second type of sides is a side defined by $Merge(Q, Q')$ and only occurs when one of the two points of intersection between the two side sets $Q_1 - \mathcal{Q}_1^c$ and $Q_2 - \mathcal{Q}_2^c$ is the end point of the side Q and the beginning point of Q', or $p \equiv sideEP(Q) \equiv sideBp(Q')$. If the equation of the lines are not equal, then these two sides are mapped into $\mathcal{Q}_{compose}$ unchanged, and are on the line and between the end points by assumption. Otherwise the two sets $\mathcal{O}$ and $\mathcal{O}'$ are concatenated. Since the line equation in this case is the same, or $L_j(x, y) \equiv L'_j(x, y) \equiv L_{compose}(x, y)$, the ports are on the line. Since $sideBp(Q_j) \leq (o \in \mathcal{O}_j) \leq sideEp(Q_j)$ and $sideBp(Q'_j) \leq (o \in \mathcal{O}'_j) \leq sideEp(Q'_j)$ by assumption, and the ports are sorted in $Q_i \equiv Merge(Q_j, Q'_j)$ from Lemma 3, this implies $(sideBp(Q_j) \equiv sideBp(Q_i)) \leq (o \in \mathcal{O}_i) \leq (sideEp(Q'_j) \equiv sideEp(Q_i))$. ■

Lemma 5. Given the external interface points $\mathbf{p}'_1$ and $\mathbf{p}'_2$ that describe the points of geometrical interest within the maximum applicable GDR of the polygons described by $\mathcal{Q}_1$ and $\mathcal{Q}_2$, respectively, then the composition point set $\mathbf{p}_{compose}$ includes every point of interest within the maximum applicable GDR to the polygon described by $\mathcal{Q}_{compose}$.

Proof: The distance between a point p and a side Q_j, is the distance between the point p and the line $L_j(x, y)$, as described in equation (3-3) in Chapter 3. Since the line equations of $Q_i \in \mathcal{Q}_{compose}$ are the same as the line equations of a subset of $\mathcal{Q}_1$ and a subset of$\mathcal{Q}_2$, from the definition of $\mathcal{Q}_{compose}$, and the polygon $\mathcal{Q}_{compose}$ bounds all points $\mathbf{p}_1'$ and $\mathbf{p}_2'$, from lemma 2, every point within the maximum applicable GDR is present in $\mathbf{p}_{compose} \equiv \mathbf{p}_1' \cup \mathbf{p}_2'$. ■

Lemma 6. Given the external interface paths $\mathcal{P}_1'$ and $\mathcal{P}_2'$ that describe the paths of geometrical interest within the maximum applicable GDR of the polygons described by $\mathcal{Q}_1$ and $\mathcal{Q}_2$, respectively, then the composition path set $\mathcal{P}_{compose}$ includes every path of interest within the maximum applicable GDR to the polygon described by $\mathcal{Q}_{compose}$.

Proof: Since the placement of a path is described by a set of points from the definition of a path, $\mathcal{P}_{compose}$ includes all paths of GDR interest from Lemma 5. ■

Lemma 7. Given a path $P_k \in \mathcal{P}$ that does not extend more than the minimum radius $mr(T_k)$ outside the polygon described by $\mathcal{Q}$, and given a second set of sides $\mathcal{Q}'$, then the path P_k does not extend more than the $mr(T_k)$ outside the polygon described by $\mathcal{Q}_{compose}$.

Proof: Since the polygon described by $\mathcal{Q}_{compose}$ bounds all points and paths bounded by $\mathcal{Q}$, from Lemma 2, the path P_k cannot extend more than the original distance $mr(T_k)$ outside of $\mathcal{Q}_{compose}$. ■

Lemma 8. Given a merge path $P_k \in \mathcal{P}_1$ on the side set $\mathcal{Q}_1$, and given a valid composition with a second set of sides $\mathcal{Q}_2$, then the path P_k either: (a) is a merge path on the sides $\mathcal{Q}_{compose}$ or (b) does not extend more than $mr(T_k)$ outside the polygon described by $\mathcal{Q}_{compose}$.

Proof: The path center-line of P_k is bounded by $\mathcal{Q}_{compose}$ from Lemma 2. If there is a side Q_j such that the path P_k lies on this side, and $Q_j \in \mathcal{Q}_1$ and $Q_j \in \mathcal{Q}_{compose}$ then (a) is true, from the definition of $\mathcal{Q}_{compose}$. Otherwise there is a side $Q_j \in \mathcal{Q}_1^c$ on which the path P_k lies, from the definition of $\mathcal{Q}_1$ and $\mathcal{Q}_{compose}$. Since this composition is valid, by assumption, there is a corresponding path P_k' on the side $Q_j' \in \mathcal{Q}_2^c$. These paths do not extend more than $mr(T_k)$ outside any side $\mathcal{Q}_1 - \{Q_j\}$ and $\mathcal{Q}_2 - \{Q_j'\}$ from the definition of merge paths. Since $\mathcal{Q}_{compose}$ is defined from the sides $\mathcal{Q}_1 - \mathcal{Q}_1^c$ and $\mathcal{Q}_2 - \mathcal{Q}_2^c$, from the definition of $\mathcal{Q}_{compose}$, (b) is true. ■

Lemma 9. Given the external interface paths $\mathcal{P}_1'$ and $\mathcal{P}_2'$, and two side sets $\mathcal{Q}_1$ and $\mathcal{Q}_2$, where for a path $P_k \in \mathcal{P}_1'$ and $P_k' \in \mathcal{P}_2'$, either (a) the path geometry does not extend more than $mr(T_k)$ outside the polygon described by $\mathcal{Q}_1$ and $\mathcal{Q}_2$, respectively, or (b) the path is a merge path on $\mathcal{Q}_1$ and $\mathcal{Q}_2$, respectively. Then for each path $P_k \in \mathcal{P}_{compose}$ either (a′) the path geometry does not extend more than $mr(T_k)$ outside the polygon describes by $\mathcal{Q}_{compose}$ or (b′) the path is a merge path on $\mathcal{Q}_{compose}$.

Proof: Each path $P_j \in \mathcal{P}_1'$ and $P_j' \in \mathcal{P}_2'$ fall into either (a) or (b) by assumption. If (a) is true then (a′) is true from Lemma 7, otherwise (b) is true and either (a′) is true or (b′) is true from Lemma 8. ■

Definition 19. The *composition* of two cells C_1 and C_2 is characterized as:

$$[\mathcal{P}_{compose}, \mathbf{p}_{compose}, \mathcal{Q}_{compose}, \{I_1, I_2\}, ic]$$

where I_1 represents the cell C_1, and I_2 represents the cell C_2, and ic represents the connectivity between the two cells. [Section 6.1]

Theorem 1. Given the well-formed cells C_1 and C_2, and given a valid composition between the two cells, the fully instantiated geometry of the composition of the two cells is GDR correct.

Proof: Since each cell is GDR correct independently, from the definition of well-formed, a GDR violation may only be introduced along the interface, or where $\mathcal{Q}_1^c$ is superimposed on $\mathcal{Q}_2^c$. Every geometrical element from the fully instantiated geometry within the maximum applicable GDR is present in $\mathcal{P}_1$, $\mathbf{p}_1$, $\mathcal{P}_2$, and $\mathbf{p}_2$, from the definition of well-formed, and therefore in the external interface sets $\mathcal{P}_1'$, $\mathbf{p}_1'$, $\mathcal{P}_2'$, and $\mathbf{p}_2'$ by definition. There is a valid composition, by assumption, and therefore the composition of the geometry in $\mathcal{P}_1'$, $\mathbf{p}_1'$, $\mathcal{P}_2'$, and $\mathbf{p}_2'$ is GDR correct from the definition of a valid composition. ■

Theorem 2. Given two well-formed cells C_1 and C_2, with composible side sequences $\mathcal{Q}_1^n$ and $\mathcal{Q}_2^n$, a valid composition of the two cells produces a well-formed cell.

Proof: The two cells C_1 and C_2 may be characterized as

$$[\mathcal{P}_1, \mathbf{p}_1, \mathcal{Q}_1, I_1, \mathbf{ic}_1] \text{ and } [\mathcal{P}_2, \mathbf{p}_2, \mathcal{Q}_2, I_2, \mathbf{ic}_2].$$

The geometrical information of interest, from the definition of external interface, is $[\mathcal{P}_1', \mathbf{p}_1', \mathcal{Q}_1]$ and $[\mathcal{P}_2', \mathbf{p}_2', \mathcal{Q}_2]$.

(1) Given the sides $\mathcal{Q}_1$ and $\mathcal{Q}_2$ from well-formed cells, $\mathcal{Q}_{compose}$ is a clockwise simply connected polygon from Lemma 1, and bounds all path center-lines in $\mathcal{P}_{compose}$ and points in $\mathbf{p}_{compose}$ from Lemma 2.

(2) Given $\mathcal{O}_1$ and $\mathcal{O}_2$ from well-formed cells, the ports $\mathcal{O}_i$, where $Q_i \in \mathcal{Q}_{compose}$, are sorted from Lemma 3, and on the line $L_i(x, y)$ and between the two end points $sideBp(Q_i)$ and $sideEp(Q_i)$, inclusively, from Lemma 4.

(3) Given $\mathcal{P}_1'$, $\mathbf{p}_1'$, $\mathcal{P}_2'$, and $\mathbf{p}_2'$ derived from well-formed cells, there is a point $p \in \mathbf{p}_{compose}$ for every point within the maximum applicable GDR from the sides $\mathcal{Q}_{compose}$ from Lemma 5, and a path $P \in \mathcal{P}_{compose}$ for every path within the maximum applicable GDR from the sides $\mathcal{Q}_{compose}$ from Lemma 6.

(4) Given $\mathcal{P}_1'$ and $\mathcal{P}_2'$, from well-formed cells, for each path $P \in \mathcal{P}_{compose}$ either: (a) the path does not extend more than $mr(T_k)$ outside the polygon described by $\mathcal{Q}_{compose}$, or (b) the path is a merge path on $\mathcal{Q}_{compose}$, from Lemma 9.

(5) Given two well-formed cells C_1 and C_2, the fully instantiated geometry in the composition is GDR correct from Theorem 1. ■

Definition 20. Given a cell C_1, the *omit ports* are a list of ports ordered on the list of sides $\mathcal{Q}_{omit_1}$, where $\mathcal{Q}_{omit_1} \subseteq \mathcal{Q}_1$. For every port $o \in \mathcal{O}_j$, where $Q_j \in \mathcal{Q}_{omit_1}$, the port is also defined in the cell ports $\mathcal{O}_1$, or $o \in \mathcal{O}_1$. [Section 6.2]

Definition 21. The *omitted port side* that results from removing the omit ports from the side Q is:

$$OmitSide(Q, Q_{omit}) = \begin{cases} Q, & L(x,y) \neq L_{omit}(x,y) \\ [L(x,y), \mathcal{O} - \mathcal{O}_{omit}], & L(x,y) = L_{omit}(x,y). \end{cases}$$

Corollary 1. Given two well-formed cells C_1 and C_2, with composible side sequences $\mathcal{Q}_1^n$ and $\mathcal{Q}_2^n$, and two omit port sets $\mathcal{Q}_{omit_1}$ and $\mathcal{Q}_{omit_2}$, a valid composition of the two cells produces a well-formed cell.

Proof: The sides $\mathcal{Q}_1$ and $\mathcal{Q}_2$ may be mapped into two new side sets $\mathcal{Q}_1'$ and $\mathcal{Q}_2'$ by removing all omitted ports contained in $\mathcal{O}_j$, where $Q_j \in \mathcal{Q}_{omit_1}$ and $Q_j \in \mathcal{Q}_{omit_2}$ using *OmitSide*. The two new cells, characterized as

$$[\mathcal{P}_1, \mathbf{p}_1, \mathcal{Q}_1', \mathcal{I}_1, \mathbf{ic}_1] \text{ and } [\mathcal{P}_2, \mathbf{p}_2, \mathcal{Q}_2', \mathcal{I}_2, \mathbf{ic}_2]$$

are well-formed, from (2) in the definition of well-formed. A valid composition of these two cells produces a well-formed cell from Theorem 2. ■

Corollary 2. Given $m+1$ cells $\{C_1, C_2, \ldots C_{m+1}\}$ and m pairs of composible side sequences $\{\langle \mathcal{Q}_{2_1}^n, \mathcal{Q}_{2_2}^n \rangle, \ldots \langle \mathcal{Q}_{m_1}^n, \mathcal{Q}_{m_2}^n \rangle\}$, a sequence of m valid compositions produces a well-formed cell.

Proof: The proof is by induction on valid composition. The 1st valid composition between cell C_1 and C_2 produces a well-formed cell from Theorem 2. Assume that the $m-1$ valid composition produces a well-formed cell C_{m-1}'. Then the composition between this cell and the m^{th} cell C_m produces a valid composition from Theorem 2. ■

7. System Considerations

This chapter describes some of the major considerations of a system that supports the Pooh representation. Any design system supporting Pooh runs on a computer with finite resolution and speed. There are always issues such as computational efficiency versus accuracy faced by the implementor. Two approaches to a Pooh system have been implemented, and Chapter 8 describes both systems. The first approach is batch oriented, where a user executes a program that creates a Pooh circuit from an input specification. The second approach is interactive, where a designer receives immediate feedback from the system as the circuit is designed.

The primitive elements used in the synthesis and analysis algorithms described in the preceding chapters are lines, arcs and circles; and the algorithms based their validity on the existence of perfect circles and arcs. The first section presents a circle approximation, indicates how the approximation affects the algorithms, and then analyzes the potential errors introduced by the approximation.

Any Pooh system that detects GDR violations between elements must have quick and easy access to the paths and points surrounding a given path segment or Pooh point. The second section describes a data structure that allows a Pooh system to find all paths and points "near" a path.

Part of the motivation for developing the Pooh representation was the observation that designs have always used 45° degree angles, and that their addition introduces additional complexity over a strictly vertical and horizontal approach (a manhattan approach). We were interested in finding a general representation. Once the limits of a representation such as Pooh have been explored, it is often useful to look for simplifications. The third section presents an integer based Pooh that uses an octagon circle approximation and supports vertical, horizontal and 45° lines.

7.1 Circle Approximation

At the heart of a Pooh system lies its circle approximation. The original Pooh implementation used a twelve sided circle. In this section we will look at the approximation for an n-sided circle, where $8 \leq n \leq 100$. Pooh uses this approximation when generating polygons from its internal paths and points. The polygons for all of the segment ends and arcs are generated based on this circle as shown in Figure 7-1. The GDR calculations described in Chapters 3 and 5 perform calculations between center-lines, points, and arcs assuming that the geometry will be composed of perfect circles. Obviously actual GDR violations could be introduced on the chip if the circle approximation introduced edges that were closer together then the allowable distances, even though fabrication tends to smooth out the edges.

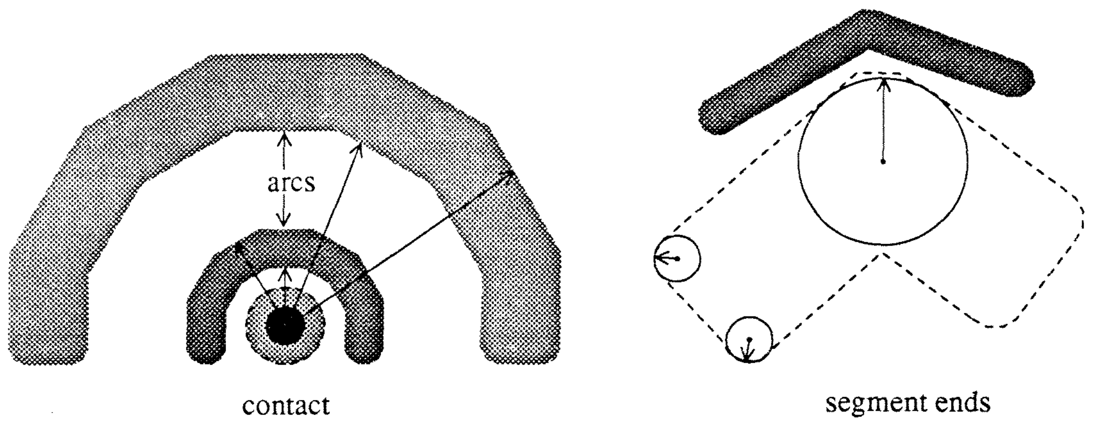


Figure 7-1: Geometry Generated From Circle

The Circle

Pooh uses an n-point unit polygon that circumscribes the unit circle. A twelve sided polygon is shown in Figure 7-2. This unit circle is Pooh's "grid", similar to a box in a manhattan system. One Pooh unit is calculated based on a circle of radius one. Therefore a unit circle approximation cannot be smaller than the unit circle at any point. The points $\mathbf{p}^c$ for an n-point unit circle approximation are:

$$d = 1/\cos(\frac{\pi}{n})$$
$$p_i^c = \left(d \times \cos(\frac{2\pi}{n}(i + .5)), d \times \sin(\frac{2\pi}{n}(i + .5))\right)$$
$$\mathbf{p}^c = \{p_0^c, p_1^c, \cdots p_{n-1}^c\}.$$

The factor d ensures that the line segments connecting any two consecutive points lie outside the unit circle. If the number of points $n = 3$, then $d = 2$. As n increases, the angle π/n becomes smaller and d approaches 1.

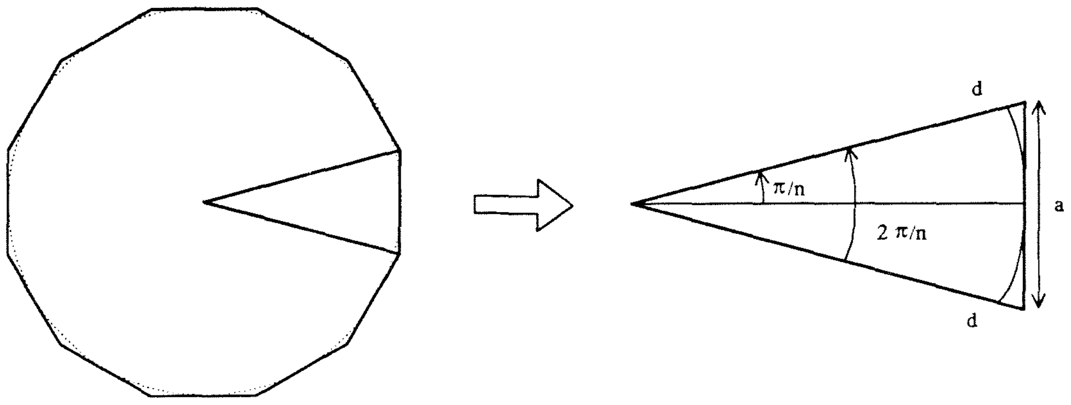


Figure 7-2: A twelve sided circle approximation

Each pie-shaped section approximates an arc by a straight line, as shown in the figure. The actual arc length is $(2\pi r)/n = (2\pi)/n$. The triangle side length $a = 2d\sin(\frac{\pi}{n}) = 2\sin(\frac{\pi}{n})/\cos(\frac{\pi}{n}) = 2\tan(\frac{\pi}{n})$. The triangle approximates each arc by a line, or $\frac{\pi}{n} \approx \tan(\frac{\pi}{n})$, and $d \approx 1$.

Error Analysis

In Pooh, the geometry for contacts, arcs, and segment ends is generated based on this circle approximation. GDR violations may be introduced if the circle approximation diverges significantly from the actual circle. Each circular polygonal point, as shown in Figure 7-3, is calculated as follows:

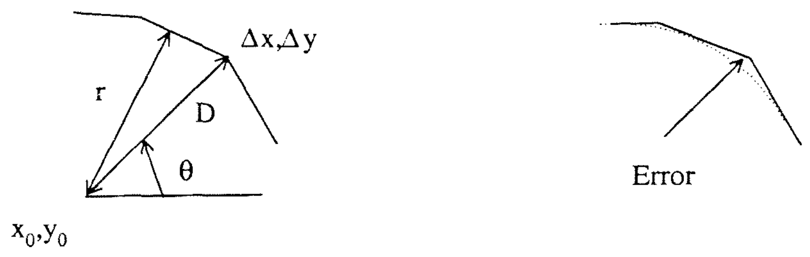


Figure 7-3: Polygon Error

$$\begin{aligned} \Delta x &= r \times d \times \cos\theta \\ \Delta y &= r \times d \times \sin\theta \\ D^2 &= r^2 d^2(\cos^2\theta + \sin^2\theta) \\ D &= r \times d. \end{aligned}$$

The maximum error is the actual distance D minus the expected distance r or:

$$Error = \mathcal{E} = r \times d - r = r(d-1).$$

The following table indicates what r is equal to before the error $\mathcal{E} \geq 0.5$ for different values of n.

n	8	12	16	20	50	100
d	1.0824	1.0353	1.0196	1.0125	1.0020	1.0005
$d-1$	0.0824	0.0353	0.0196	0.0125	0.0020	0.0005
r	6	14	25.5	40	253	1013

The Pooh GDR calculations effected by the circle approximation fall into four distinct categories: 1) point checks, 2) arc synthesis, 3) arc-segment checks, and 4) arc-arc checks. Each of these categories uses the circle approximation in different ways.

All Pooh points are constructed using the circle approximation. Pooh points include both g-points, used at the end of segments, and contacts. Transistor points occur at the intersection of two segments and are guaranteed to have overlap on all sides, thus the transistor point geometry is irrelevant to this discussion. Examples of polygonal edges generated based on the circle approximation are shown in Figure 7-4.

Segment ends are either path ends or intermediate points. A segment end that is also a path end is generated as two arcs connected by a line. The radius of these arcs is the minimum radius of the path, thus the maximum possible error for the two path ends is constant for each path type. The minimum radius for the path is half the minimum width, typical values are between 1 and 1.5, therefore $\mathcal{E} \approx d - 1$. Segment ends that are not path ends use the circle approximation to bend around corners. The arc radius is the path radius, or half the path width. Typical values for path radii are between 1 and 4, therefore the worst case error is normally $(d-1) \leq \mathcal{E} \leq 4(d-1)$, which is quite tolerable.

The geometry of a contact is generated from the circle approximation in a straightforward manner. Typical contact radii are between 1 and 2, and a very large contact would have a radius of 4. Therefore the worst case error associated with a contact is similar to the segment ends and is very tolerable.

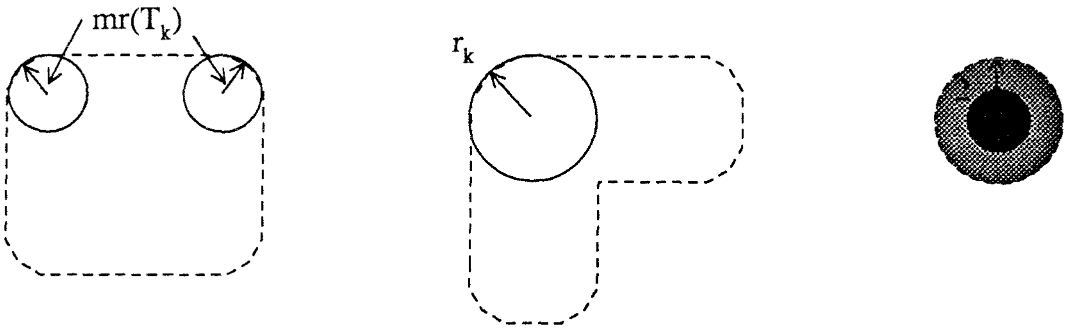


Figure 7-4: Point Geometry

Many arcs may "miss" the same point p_q, each arc having a unique arc radius r. Given two arcs, $\mathcal{A}_1$ and $\mathcal{A}_2$ with radius r_1 and r_2, respectively, where $r_1 \neq r_2$, the expected distance between these two arcs is $D_E = |r_1 - r_2|$. The minimum actual distance $D_A = |d \times r_1 - d \times r_2| = d|r_1 - r_2|$. Since $d > 1$ the actual distance $D_A > D_E$, therefore an error is not introduced.

Pooh performs the check between an arc $\mathcal{A}_2$ and the line segment portion of a segment s_1 by checking the distance between the point p_2 and the line $L_1(x, y)$ and comparing this distance minus the arc radius r_2 to the GDR spacing rule. The expected distance is $D_E = |LineDis(L_1, p_2)| - |r_2|$ and the actual distance at the circle corners is $D_A = |LineDis(L_1, p_2)| - d \times |r_2|$. The maximum error is $\mathcal{E} = D_E - D_A = d \times |r_2| - |r_2| = |r_2| \times (d-1)$. This error increases with the arc radius, as described earlier.

A simple restriction on the line segments would allow a Pooh system to ignore this type of error. If the slopes of the line equations $(-A_i/B_i)$ are chosen from a set of allowable slopes parallel to one of the circle approximation sides, then it is not possible to introduce the error of $r(d-1)$. A system that supports this type of approach would allow $n/2$ possible line segment slopes where n is the number of points in the circle approximation. A fixed number of line styles is reasonable for any real design and would ensure that the separation between the arc and the line would always occur along one of the straight edges of the arc, as illustrated in Figure 7-5.

The other possibility is to be conservative and always calculate the separation between the line segment and the arc based on the worst case error. This simply means changing the distance calculation in the function `line_arc_interfere`, in Chapter 5, to $|LineDis(L_1, p_2)| - d \times |r_2|$.

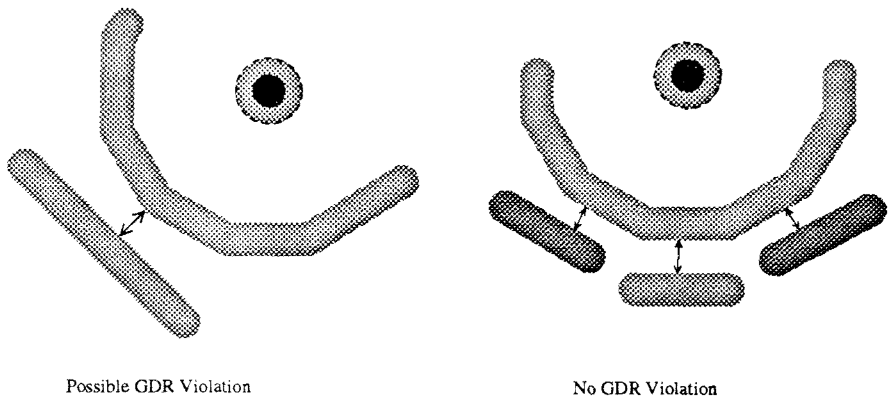


Figure 7-5: Arc Geometry

Pooh computes the expected distance between two arcs $\mathcal{A}_1$ and $\mathcal{A}_2$, with radii r_1 and r_2 as $D_E = PointDis(p_1, p_2) - |r_1| - |r_2|$. The actual distance is $D_A = PointDis(p_1, p_2) - |d \times r_1| - |d \times r_2| = PointDis(p_1, p_2) - d|r_1 + r_2|$, therefore the worst case error is $\mathcal{E} = D_E - D_A = d \times |r_1 + r_2| - |r_1 + r_2| = |r_1 + r_2|(d-1)$.

The worst case error increases with the sum of the two arc radii. This error cannot be ignored. In order to avoid a GDR violation, the arc-arc error detection function `arcs_interfere`, from Chapter 5, simply uses the D_A calculation rather than the D_E calculation.

The possible errors introduced by the circle approximation do not affect in any fundamental way the approach supported by the Pooh algorithms. Any Pooh system may easily compensate for the polygonal edges introduced by the circle approximation.

Arc Algorithms Using the Approximation

Arcs, unlike Pooh's other primitive structures, can be computationally expensive to handle properly. Fortunately, it is possible to exploit the circle approximation to minimize the arc calculations. The Pooh arc representation, as described in Section 3.1, is $\mathcal{A}_i = (r_i, p_i, p_i^c, \mathcal{A}_{i+1})$. From this information we may calculate the arc's beginning and ending points a_i^b and a_i^e using two unit circle points p_i^c and p_{i+1}^c. The circle approximation $\mathbf{p}^c$ provides n points on the unit circle, and the unit points p_i^c, from (3-5), may always be chosen from $p_0^c, p_1^c, \ldots p_{n-1}^c$. Pooh exploits this fact in all of its arc calculations.

In Chapters 3 and 5, several arc functions were left undefined. There was no reason to introduce a more complicated function definition, when by postponing their definition, the functions could be defined in terms of the circle approximation. In Chapter 3, the function $arclen(A_i)$ that calculates the length of an arc, and the function in(U_1, U_2) that detects overlap between two arcs were undefined. In Chapter 5, the function point_in(U_1, p_q) that detects overlap between a point and an arc was undefined. We may exploit the fact that the arc is defined in terms of the circle approximation to define these functions.

The *arclen* function computes the length of the arc, based on the number of line segments in the arc times the length of a single segment, or double the first y coordinate. The *arclen* function is:

```
$index(p^c) = position\ (0 \ldots n-1)\ of\ p^c\ in\ the\ circle\ approximation$
define $arclen(A_i)$:  length
    nseg $\leftarrow index(p^c_{i+1}) - index(p^c_i)$;
    if $r_i > 0$ & nseg<0 then nseg $\leftarrow$ nseg $+n$
    else if $r_i < 0$ & nseg>0 then nseg $\leftarrow$ nseg $-n$
    fi
    $arclen \leftarrow |nseg| \times |r_i| \times y^c_0 \times 2$;
enddef
```

The function in uses the unit point indices to determine overlap between two arcs "missing" the same point. The function is:

```
define in_Range($b, e, q$):  boolean
    in_Range $\leftarrow b \leq q \leq e$;
enddef

define arcs_in_Range($b_1, e_1, b_2, e_2$):  boolean
    if $b_2 < b_1$ then
        arcs_in_Range $\leftarrow$ in_Range($b_2, e_2, b_1$) OR in_Range($b_2, e_2, e_1$)
    else arcs_in_Range $\leftarrow$ in_Range($b_1, e_1, b_2$) OR in_Range($b_1, e_1, e_2$)
    fi
enddef

define in($U_i, U_j$):  boolean
    if $index(p^c_i) > index(p^c_{i+1})$ then
        in $\leftarrow$ if $index(p^c_j) > index(p^c_{j+1})$ then TRUE
                else arcs_in_Range($index(p^c_i), n-1, index(p^c_j), index(p^c_{j+1})$) OR
                    arcs_in_Range($0, index(p^c_{i+1}), index(p^c_j), index(p^c_{j+1})$)
                fi
    else if $index(p^c_j) > index(p^c_{j+1})$ then
        in $\leftarrow$ arcs_in_Range($index(p^c_j), n-1, index(p^c_i), index(p^c_{i+1})$) OR
                arcs_in_Range($0, index(p^c_{j+1}), index(p^c_i), index(p^c_{i+1})$);
    else in $\leftarrow$ arcs_in_Range($index(p^c_i), index(p^c_{i+1}), index(p^c_j), index(p^c_{j+1})$)
    fi
enddef
```

Finally the function point_in determines whether a point on the unit circle occurs between the two end points of a unit arc. This function is:

```
define point_in($U_i, p^c_q$) :  boolean
    point_in $\leftarrow$ if $index(p^c_i) > index(p^c_{i+1})$ then
                    in_Range($index(p^c_i), n-1, index(p^c_q)$) OR
                    in_Range($0, index(p^c_{i+1}), index(p^c_q)$)
                 else in_Range($index(p^c_i), index(p^c_{i+1}), index(p^c_q)$)
                 fi
enddef
```

With the introduction of the circle approximation, all functions associated with arc synthesis and analysis are defined.

7.2 Storing Paths and Points

The effectiveness of any system that creates and uses objects that span more than one dimension depends in large part on how easily the objects may be stored and accessed in time and space. Batch oriented design tools typically build up sorted lists while they are processing the data [Mosteller 81, Baker 80, Baird 77, Whitney 81]. Interactive systems face a different set of constraints. An intelligent approach to the data structure can be the deciding factor in the feasibility of an approach [Ousterhout 81, Ousterhout 84]. The Pooh representation presents a different set of issues than those of other approaches. A Pooh system must be able to order both points and path segments in a reasonable fashion.

The data structures that maintain the Pooh points and path segments depend on the idea of a design area. The design area is a dynamic rectangle that is always bounded by four edges — llx, lly, urx, ury. This area is usually either the bounding box of the current cell, or the screen area in an editor.

Pooh Points

The points provide the underlying structure for a Pooh circuit. A point provides both placement and connectivity information. Any two paths that are directly connected share at least one reference to the same point. Each time a path segment is created, the system must either find an existing point, if defined, or create a new point for each of the segment's points. Therefore a Pooh system must be able to quickly detect whether or not a particular point already exists. The following discussion presents a data structure that allows fast access to any point. First the static structure is presented, and then extended for dynamic use.

The point data structure is capable of representing all points in the design area, but occupies an amount of space proportional to the number of actual points and not the number of representable points. Given a cell C with r points $\mathbf{p} \equiv \{p_1, p_2, \cdots p_r\}$, we may distribute these r points on a set of buckets that span the y-axis. The number of buckets is $\sqrt{r}$ [Bentley 80], and each bucket spans an interval along the y-axis equal to $interval = (ury - lly)/\sqrt{r}$. Each bucket contains from 1 to r points. Assuming an even distribution of points in the design, the expected number of points per bucket is $\sqrt{r}$. The contents of each bucket is maintained sorted in increasing x order. Figure 7-6 illustrates the point data structure.

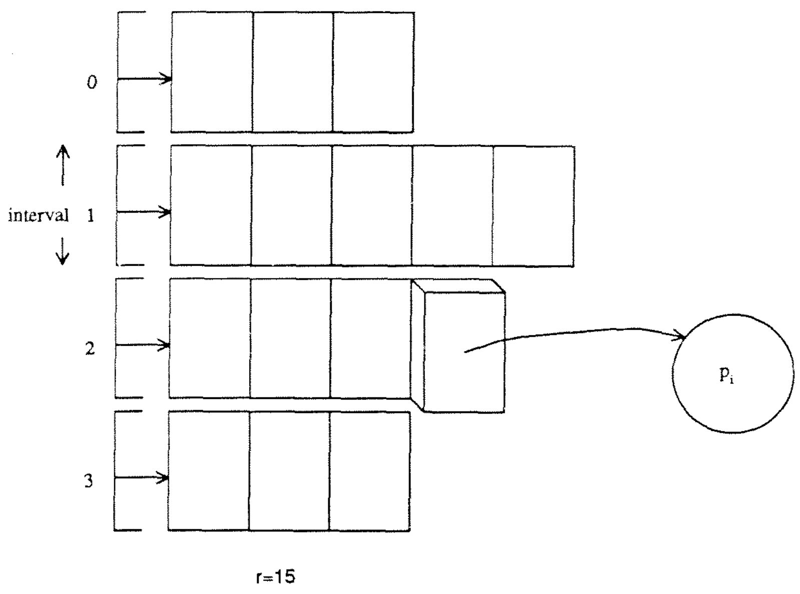


Figure 7-6: The Point Array

For each point lookup and/or insertion, the system computes the bucket as $\lfloor (y_i - lly)/interval \rfloor$, and then uses a binary search [Aho 74] to find the appropriate location. The expected number of comparisons for each point insertion is $1 + log(\sqrt{r})$ or $O\left(log(\sqrt{r})\right)$. The space complexity is $O(r)$ since all entries are used, and each entry contains a single point.

The dynamic behavior of this structure is important because an interactive editor is constantly adding points. Given an initial static configuration, it is easy to continuously update it to accommodate new points. If the design area enlarges, i.e., *lly* becomes smaller, or *ury* becomes larger, new buckets are added onto the two ends. Otherwise, the only action occurs when the number of points *r* becomes larger than the square of the number of buckets, or $nbuckets^2$. Then *nbuckets* is incremented by a small number, the *interval* is lowered, and the points are redistributed.

Path Segments

A trickier storage problem is that of storing Pooh path segments. A Pooh system must have easy access to the segments surrounding a particular point. But the number of comparisons needed to detect a GDR violation should be a function of the number of elements in an area rather than of the total number of path segments. However, unlike points, path segments occupy space. The data structure must reflect this fact, and allow the program to move easily through both dimensions.

Pooh divides the design area into a coarse grid, represented as a two-dimensional array. Each element in the array represents a rectangle in the design area space and lists every path segment whose bounding box overlaps this rectangle. This approach allows fast access to all path segments that cross a particular region of the design area, at the expense of representing the entire design area, including the unused portions.

The data structure for storing paths is a two-dimensional array representing the design area, where the z paths of cell C, $\mathcal{P} \equiv \{P_1, P_2, \ldots P_z\}$, are distributed over the array. If the average number of segments in each path is m, then the total number of path segments is approximately $m \times z$ or mz. The rows are similar to the buckets in the point array, in fact there are $2\sqrt{mz}$ rows, each spanning an interval $yinterval = (ury - lly)/(2\sqrt{mz})$. The number of columns is constant across the segment matrix, unlike the point structure. The length of each column is $2\sqrt{mz}$; each entry spans an interval $xinterval = (urx - llx)/(2\sqrt{mz})$.

Since all possible GDR violations must be detected, the single most important criteria is that path segments are represented in the data structure at every point the path geometry occupies in space. Path segments are stored based on their minimum bounding box (MBB). Often the MBB is larger than the segment itself, but as a rough first order approximation, it never allows geometry to go undetected.

The path segments' MBBs are mapped onto the two dimensional array, and segments are kept in a linked list at all entries that any part of the MBB overlaps. Figure 7-7 illustrates six segments mapped onto the array. In the figure, arrows indicate a linked list entry, and an x indicates an empty entry. For each path segment, there are four possible non-unique bounding entries, one at each of the MBB corners.

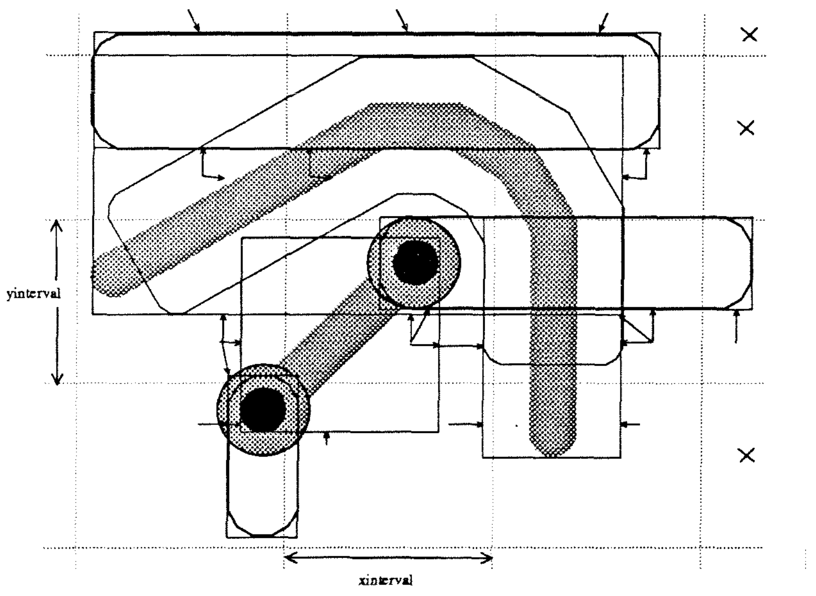


Figure 7-7: A Segment Array

For a cell with mz path segments, the total number of entries in the array is $2\sqrt{mz} \times 2\sqrt{mz} = 4 \times mz$ or $O(mz)$. Since the system may access an array entry directly from a point p_i as $\lfloor (y_i - lly)/yinterval \rfloor, \lfloor (x_i - llx)/xinterval \rfloor$, the number of comparisons is proportional to the number of segments listed in a single entry. The average number of entries in which each segment appears is the average MBB area, or A, divided by the intervals, or $nE = A/(xinterval \times yinterval)$. Assuming a uniform distribution of segments, the expected number of segments per entry is the average number of entries in which each segment appears times the number of segments, divided by the number of entries, or$(nE \times mz)/(4 \times mz)$= $nE/4$. Thus the expected time complexity for accessing elements at a particular point is a constant.

The dynamic behavior of the system is similar to that of the point structure. Given an initial array based on the number of paths, it is simple to add new path segments. If the design area bounds change, then the size of the array increases accordingly. At some point the number of path MBB edges, or four times the number of path segments, becomes larger than the number of rows squared, or row^2. At this time, the system increases both the number of rows and the number of columns by a small number, recalculates the intervals *yinterval* and *xinterval*, and redistributes the segments. The array then covers the entire design area, and keeps the number of entries proportional to the number of path segments.

The algorithm to look at interactions between a path segment and all paths "near" the segment is:

```
define row($y$):  integer
    row $\leftarrow$ $\lfloor (y - lly)/yinterval \rfloor$;
enddef
define column($y$):  integer
    column $\leftarrow$ $\lfloor (x - llx)/xinterval \rfloor$;
enddef
define path_in_error($P_k$):  boolean
    error_occured $\leftarrow$ false;
    for $s_j \in S_k$ do
        check all connected segments and mark as checked
        $lx, ly, ux, uy \leftarrow$ MBB($s_j$);
        for $r \leftarrow$ row($ly$) to row($uy$) do
            for $c \leftarrow$ column($lx$) to column($ux$) do
                for $s \in$ segment_array[$r$,$c$] do
                    if $s$ is not marked then
                        mark segment $s$
                        check GDR interactions
                    fi
                od
            od
        od
    od
enddef
```

7.3 Integer Pooh

A system that supports the complete Pooh representation is a useful tool for some applications, but it may introduce unnecessary computational inefficiencies under many circumstances. This section explores an integer Pooh. Designers have used 45° lines for a long time. The use of diagonal lines in layout is almost universally related to avoiding, or "missing", another element. The approach described in this section uses an octagon circle approximation to support vertical, horizontal and 45° lines. Using only these three line styles, Pooh arcs and lines resemble a series of connected line segments, as shown in Figure 7-8, but paths are still constructed by "missing" other structures, as shown in the figure. The points of the unit octagon may be represented as integers, and all of the basic Pooh calculations are both simplified and integer based, with obvious implications for efficiency.

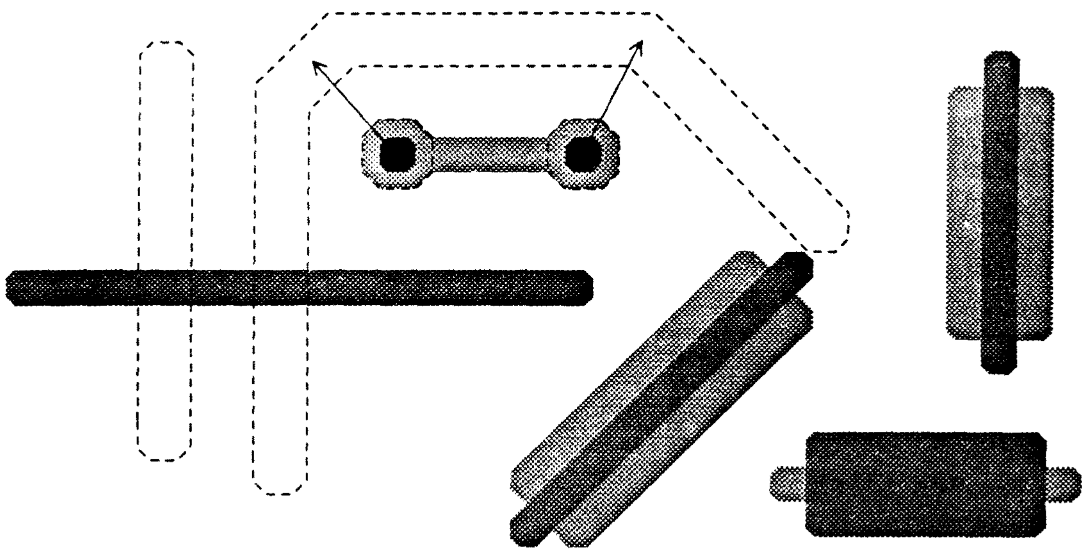


Figure 7-8: Transistors and Wires in the Integer Pooh

The Circle and the Grid

The integer Pooh uses an integer grid, where valid grid points are even integers. Figure 7-9 illustrates the integer grid. Valid grid points are designated by circles in the figure. One unit in Pooh is equivalent to four grid points. Half an internal unit is two grid points, and is the smallest increment between the coordinates of valid addressable points in the integer Pooh. Points with odd coordinates are forbidden under all circumstances.

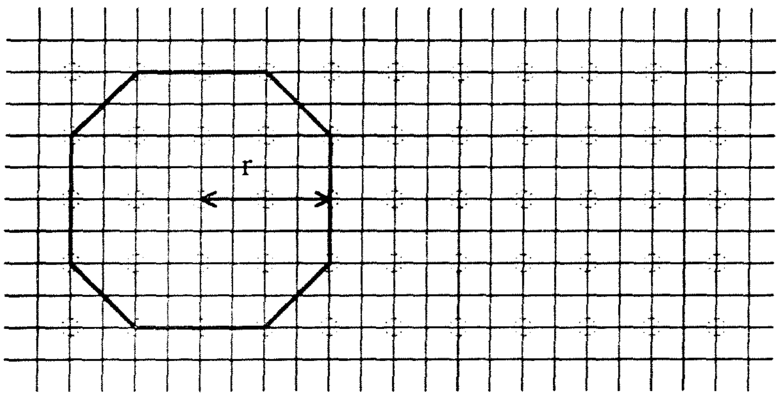


Figure 7-9: The integer grid and Unit Circle

The underlying unit circle approximation in the integer Pooh is an octagon of radius four, as shown in Figure 7-9. Using this circle as the basic unit of measure, the range of the integer system is sufficient both in lambda units [Mead 80] and in microns.

For example, on a 32 bit computer, with one internal Pooh unit equal to a hundredth of a micron, the largest representable design unit is 2^{30}. For signed numbers, the range of representable units is $\pm 2^{29} = \pm 536,870,912$. Typical chips today are a centimeter on a side, or 10^4 microns. Wafers are approximately 10 – 20 centimeters on a side, or $\sim 10^5$ microns, or 10^7 hundredths of a micron. Thus any design conceivable today is easily representable in this numbering scheme, with plenty of resolution remaining for devices of ultimately small dimension.

This particular octagon approximation was chosen primarily because its end points are on a simple grid, i.e., an even grid, and the lines and arcs generated based on this circle approximation are on the same grid. Although this octagon is not truly circular, according to the criteria described in Section 7.1, its deviation from an actual circle is very tolerable, as illustrated in Figure 7-10.

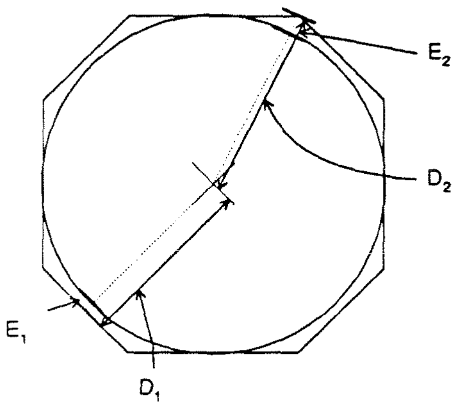


Figure 7-10: Octagon Error

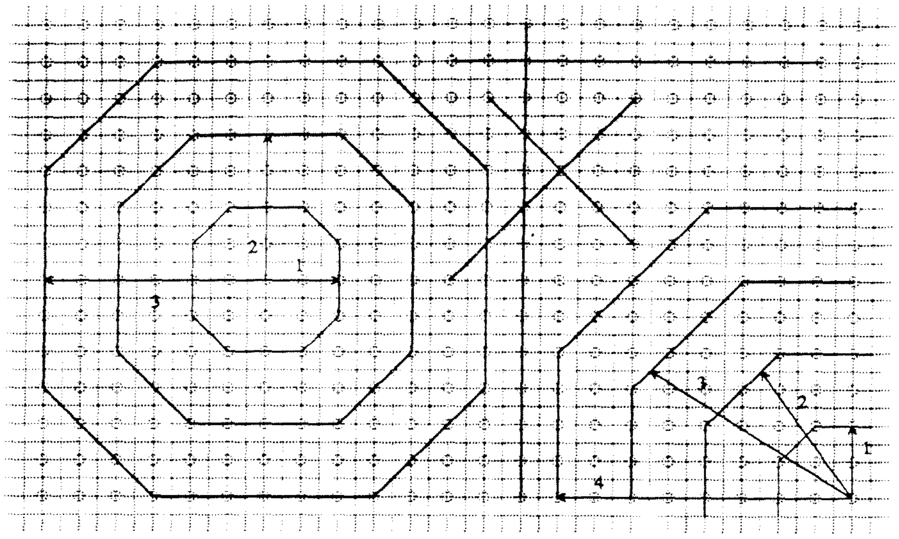
The distance from the oblique edges of the octagon to the octagon center is $D_1 = \sqrt{18} = 3\sqrt{2} \approx 4.24$. The deviation from the circle radius is $D_1 - 4$, or .24, with an error of $E_1 = .24/4 = .06$. The distance from the octagon corners to the center is $D_2 = \sqrt{20} = 4\sqrt{5} \approx 4.47$. The deviation from the circle radius is $D_2 - 4$, or .47, or an error of $E_2 = .47/4 = .118$. These deviations are comparable to the $d - 1$ factor described in Section 7.1.

Lines, Points and Arcs

The primitive elements of the Pooh representation are lines, points and arcs. All of the Pooh algorithms are based on calculations between these elements. The first step in exploring an integer based Pooh is to decide how to represent these elements.

Pooh points are circles, represented as octagons in the integer Pooh. An octagon with radius four represents a unit circle, and points with radii larger than one are scaled accordingly. Figure 7-11 illustrates example points of various radii.

Lines are restricted to be horizontal, vertical or 45° according to the original criteria of the integer Pooh. If the line end points are guaranteed to be on valid grid points, lines may be placed anywhere on the grid. The intersection of any two lines is always on a valid grid point, with one exception that is described later in this section. There are eight different directed lines available in the integer Pooh. The line equations for each of these lines are shown in the following table.



Points Lines Arcs

Figure 7-11: Circles, Lines and Arcs

	A	B	C
$\leftarrow$	0	-1	y
$\rightarrow$	0	1	$-y$
$\uparrow$	-1	0	x
$\downarrow$	1	0	$-x$
$\nearrow$	$-\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}(x-y)$
$\searrow$	$\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}(-x-y)$
$\swarrow$	$\frac{1}{\sqrt{2}}$	$-\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}(y-x)$
$\nwarrow$	$-\frac{1}{\sqrt{2}}$	$-\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}(x+y)$

The line equation $Ax+By+C=0$ for valid integer Pooh lines may be rewritten as $\frac{1}{m}(A'x+B'y+C')=0$, where the coefficients $A', B' \in \{0, \pm 1\}$, and C' is always an even integer. The values of both the original coefficient and the new coefficient for the three types of lines are shown in the following table.

	A	B	C	m	A'	B'	C'
Horizontal	0	± 1	$\mp y$	1	0	± 1	$\mp y$
Vertical	± 1	0	$\mp x$	1	± 1	0	$\mp x$
45°	$\pm\frac{1}{\sqrt{2}}$	$\pm\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}(\mp x \mp y)$	$\sqrt{2}$	± 1	± 1	$\mp x \mp y$

Pooh arcs are calculated based on a circle and a signed radius. In the integer Pooh, arc radii must be multiples of the basic unit of measure, i.e., an arc radius may not be on a half unit grid. The reason for this restriction is discussed later in this section. Figure 7-11 illustrates examples of legal arcs.

Line Calculations

There are two important line calculations that all the Pooh algorithms depend on. If these two calculations are integer based, all of the line calculations in Pooh become integer based. These two calculations are: (1) the distance between a point and a line, or equation (3-3), and (2) the line intersection calculation, or equation (3-4).

The distance calculation is:

$$D = Ax + By + C.$$

There are two important pieces of information in this calculation: the sign of the distance and the actual distance from a point to a line. The sign of $Ax + By + C$ is the same as the sign of $A'x + B'y + C'$. The square of the GDR distance may be stored by the integer Pooh system and compared to the square of the line equation. If a point is less than the GDR distance from a line, the square of the distance from the point to the line is less than the square of the GDR distance. These two equations are:

$$\begin{aligned} Sign(D) &= A'x + B'y + C' \\ D^2 &= \frac{1}{m^2}(A'x + B'y + C')^2. \end{aligned}$$

The number D^2 is guaranteed to be an integer. If the line is horizontal or vertical then $m = 1$, $1/m^2 = 1$, $A', B' \in \{0, \pm 1\}$, x and y are integer, therefore D^2 is an integer. If the line is 45°, then $m = \sqrt{2}$ and $1/m^2 = 1/2$, and the 45° distance equation may be rewritten as:

$$\begin{aligned} D^2 &= 1/2 \times \left[2 \times \left(\pm 1(x/2) \pm 1(y/2) + (\mp x \mp y)/2\right)\right]^2 \\ &= 2[\pm 1(x/2) \pm 1(y/2) + (\mp x \mp y)/2]^2. \end{aligned}$$

Since both x and y are even, the bracketed computation is an integer, making the entire computation an integer one.

Notice that the distance calculation is not susceptible to errors introduced from the octagonal approximation, since all valid line types are parallel to an octagon edge, as suggested in Section 7.1.

The line intersection calculation determines the point at which two lines intersect, and is used by Pooh extensively, often in a simplified form. The intersection equation is:

$$\begin{aligned} \det &= B_1 A_2 - A_1 B_2 \\ p_{int} &= \left((B_2 C_1 - B_1 C_2)/\det,\ (A_1 C_2 - A_2 C_1)/\det\right). \end{aligned}$$

If the determinant is zero, the two lines are parallel. The following discussion assumes that the two lines of interest are not parallel.

If the two lines in question are vertical and horizontal, then the line coefficients are 0 or ± 1, or $A_1, A_2, B_1, B_2 \in \{0, \pm 1\}$, thus $\det \in \{0, \pm 1\}$. If the determinant is 0, the lines are parallel, and therefore uninteresting. Otherwise the only information of interest is the sign of "det". Multiplying by ± 1 is equivalent to dividing, thus the equation may be rewritten as:

$$p_{int} = \det \times \left((B_2 C_1 - B_1 C_2),\ (A_1 C_2 - A_2 C_1)\right).$$

If both lines are 45°, the determinant is either 0 or ± 1, as is always true for parallel and perpendicular lines. If the determinant is non-zero, then we may multiply by "det", rather than dividing. The form of the intersection between these two lines is:

$$\begin{aligned} p_{int} &= \det \times \left[\pm\frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}(\pm x \pm y)\right) - \pm\frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}(\pm x \pm y)\right), \right. \\ &\quad \left. \pm\frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}(\pm x \pm y)\right) - \pm\frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}(\pm x \pm y)\right)\right] \\ p_{int} &= \det \times \frac{1}{2} \times (B'_2C'_1 - B'_1C'_2,\ A'_1C'_2 - A'_2C'_1)\,. \end{aligned}$$

The point p_{int} is an integer because: (a) the coefficients A' and B' are ± 1, and the C' coefficients are combinations of the even x and y coordinates, (b) the component det is ± 1, and (c) the factor $1/2$ applied to even integers produces integers.

If one line is 45°, and the other is vertical, then the calculation is of the form:

$$\begin{aligned} \det &= \pm\frac{1}{\sqrt{2}}(\pm 1) - \left(\pm\frac{1}{\sqrt{2}}\right) \times 0 = \pm\frac{1}{\sqrt{2}} \\ p_{int} &= \left[(B_2\left(\frac{\pm x_1 \pm y_1}{\sqrt{2}}\right) - \pm\frac{1}{\sqrt{2}}C_2)/\left(\pm\frac{1}{\sqrt{2}}\right), \right. \\ &\quad \left. (\pm\frac{1}{\sqrt{2}}C_2 - A_2\left(\frac{\pm x_1 \pm y_1}{\sqrt{2}}\right)/\left(\pm\frac{1}{\sqrt{2}}\right)\right] \\ p_{int} &= \sqrt{2}\left[\frac{1}{\sqrt{2}}(B_2C'_1 - B'_1C_2,\ A'_1C_2 - A_2C_1)\right] \\ p_{int} &= (B_2C'_1 - B'_1C_2, A'_1C_2 - A_2C_1). \end{aligned}$$

The intersection between 45° and horizontal lines works in an identical fashion. In general, the intersection between two lines may be calculated as:

$$\begin{aligned} \det' &= B'_1A'_2 - A'_1B'_2 \\ p_{int} &= \det' \times \left(\frac{1}{m_1^2} \max \frac{1}{m_2^2}\right) \times (B'_2C'_1 - B'_1C'_2, A'_1C'_2 - A'_2C'_1). \end{aligned}$$

The remaining algorithms for lines are angle-angle detection. Since there are only eight possible line types, the simplest solution is to calculate the angle between the eight types of lines once, and perform a table lookup based on the types of the two lines in question.

Point-Point Calculations

The final operation of importance is to convert to integers the calculation of the distance between two points. The calculation, equation (5-1), is:

$$PointDis(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

One possibility is to remove the square root by comparing the point distance to the GDR distance squared, the approach taken in the line to point calculation. This approach is valid, however it is susceptible to errors introduced by the octagonal approximation to the circle, especially when this calculation is used in conjunction with the arc-arc distance calculation, as described in Section 7.1.

An alternative approach is illustrated in Figure 7-12. Notice that given two points p_1 and p_2, we may construct four lines through p_2, parallel to the four unique sides of the octagon surrounding p_1. Each of these lines, denoted as L_1, L_2, L_3, and L_4, represents one side of an octagon centered at p_1, with a radius equal to the distance between p_1 and the line. The largest of these octagons includes the point p_2 on its boundary.

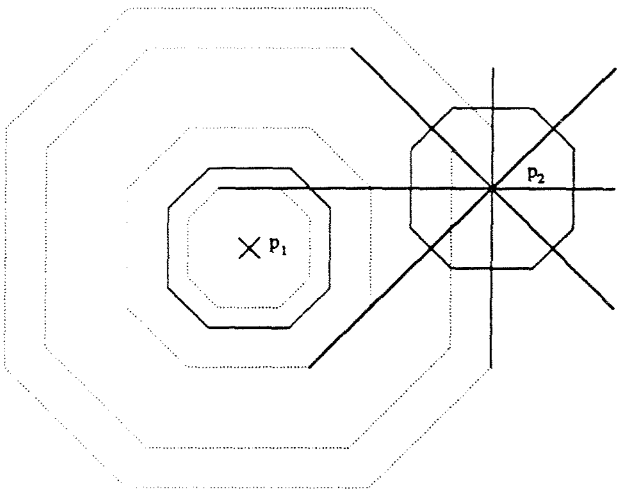


Figure 7-12: Distance Between Two Points

Given two points and the integer grid, it is always possible to construct an octagon surrounding p_1, with p_2 on its boundary. Since there are only four unique line types in an octagon, at least one of the four lines L_1, L_2, L_3, or L_4 is on this octagon. Since the line segments that comprise this octagon are evenly spaced around p_1, it is not possible to construct a line with a larger distance to p_1 that still intersects p_2. Thus we define the point distance between p_1 and p_2 to be the maximum of the line-point distances between the lines through p_2, L_1, L_2, L_3 and L_4, and the point p_1. This distance is the closest octagon radius to the actual Euclidean point distance between the two points, is always greater than or equal to the actual distance, and is not susceptible to circle approximation errors.

The sign of the line-point distance between these four lines and p_1 is not important, thus we choose four of the eight possible line equation, each with a unique slope. The four line equations of lines that intersect the point p_2 are shown in the following table.

	A	B	C	$D = Ax_1 + By_1 + C$
$\rightarrow$	0	1	$-y_2$	$y_1 - y_2$
$\downarrow$	1	0	$-x_2$	$x_1 - x_2$
$\searrow$	$\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}(-x_2 - y_2)$	$\frac{1}{\sqrt{2}}(x_1 + y_1 - x_2 - y_2)$
$\swarrow$	$\frac{1}{\sqrt{2}}$	$-\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}(y_2 - x_2)$	$\frac{1}{\sqrt{2}}(x_1 - y_1 + y_2 - x_2)$

The calculation is:

```
define points_too_close($p_1$, $p_2$, $D$, $D^2$):  boolean
    $I_1 \leftarrow x_1 - x_2$; $I_2 \leftarrow y_1 - y_2$;
    if $|I_1|$ max $|I_2| > D$ then points_too_close $\leftarrow$ false
    ef $\frac{1}{2}\left[|(I_1 + I_2)|\right.$ max $\left.|(I_1 - I_2)|\right]^2 > D^2$ then
    points_too_close$\leftarrow$ false
    else points_too_close $\leftarrow$ true fi
enddef
```

This function is integer based since both I_1 and I_2 are defined by the even coordinates x and y. This calculation performs fewer operations than the point distance calculation, and the distance is based on the octagon circle approximation, thus circle approximation errors are not present.

Problems

There are always problems in any system that attempts to restrict its domain. Usually these problems are not insurmountable, but they must be addressed. For example, in a strictly manhattan system that enforces GDRs, the system must decide what to do about the non-manhattan distances of corner to corner spacings. In the integer Pooh, there are two inherent problems with the interactions between valid line segments and valid points.

The first problem is with the intersection of two 45° lines. It is possible for two such lines to intersect at an odd grid point, as shown in Figure 7-13(a). The intersection is still on an integer grid, but not on the valid even grid. A system supporting an integer Pooh must detect and prevent this situation by moving one of the lines until the intersection is at a valid grid point, as shown in the figure.

The second problem is with half unit arcs. The end points of the line segments algorithmically generated for the half unit arc do not fall on valid grid points, shown as invalid arcs "missing" p_1 in Figure 7-13(b). This problem may be addressed by constructing the half unit arcs in terms of a related point, shown in the figure as p_2. This point produces valid arcs for the half unit arcs of p_1. It is illegal however to mix half unit arcs and full unit arcs "missing" the same point.

The Pooh Algorithms

The Pooh algorithms are mapped directly into the integer Pooh with the restrictions described, and using the integer based calculations. All of the calculations described in Chapter 3 depend on the line equations and rely heavily on the line intersection calculation. Most of the path construction calculations are simplifications of the line intersection. The angle between lines may be calculated once, since there are a small number of unique legal line slopes, and looked up for a particular interaction. The arc calculations already exploit the circle approximation, as described in Section 7.1.

The analysis algorithms described in Chapter 5 use the line-point distance and the point-point distance as the basis for all of its GDR violation detection techniques. The point-point distance is also used to detect violations between arcs. A simplification

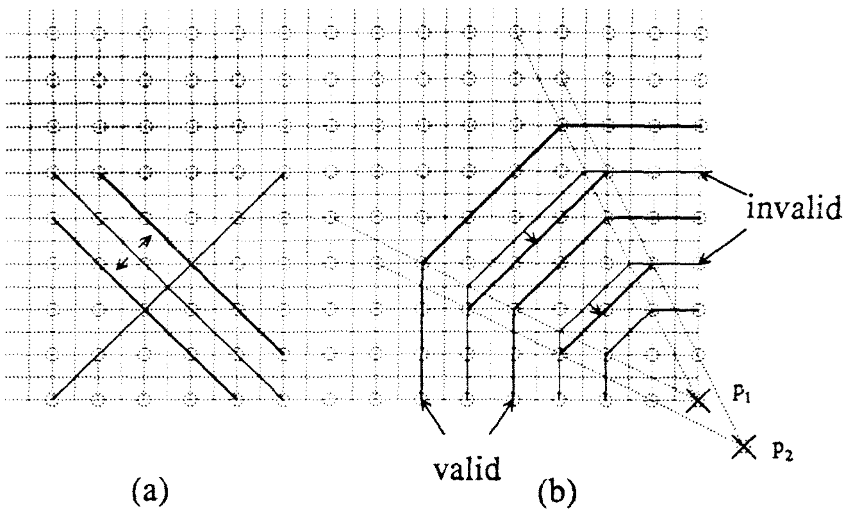


Figure 7-13: Possible Invalid Elements

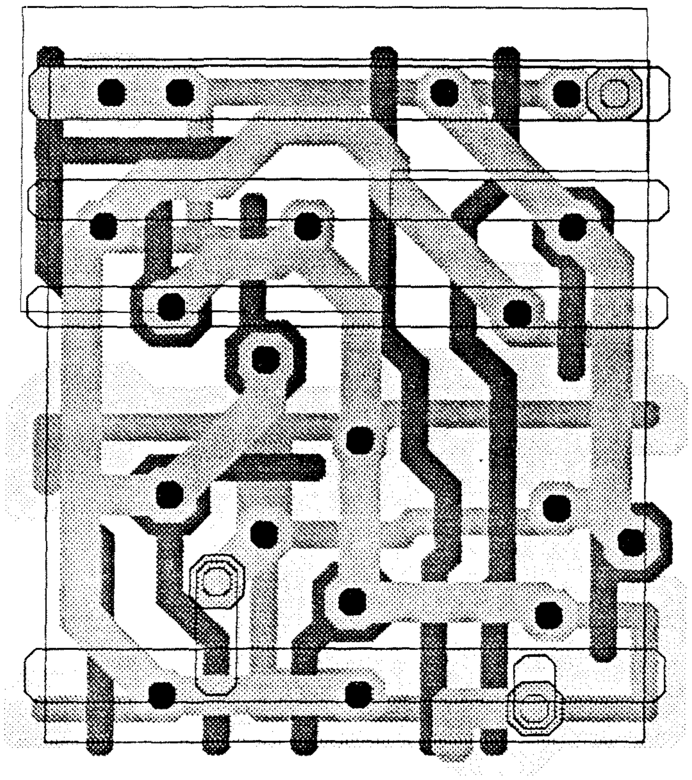


Figure 7-14: The Integer Pooh version of the Bit Serial Multiplier Cell

of the line intersection calculation allows Pooh to detect whether a point is between the two end points of a line segment. Thus the analysis algorithms may become integer based.

Pooh composition uses the line equations to represent the cell sides. The integer version of the composition restricts the valid sides to vertical, horizontal or 45°. In order to preserve the integer grid, cell rotation and mirroring is restricted to multiples of ninety degrees. The line intersection calculation is used during composition in the construction of the cell interface. Finally the analysis algorithms are used to detect GDR violations between composed cells. Thus the entire composition approach may also be integer based.

By imposing some restrictions on the angle of valid lines, the Pooh representation and its associated algorithms may be both simplified and integer based. One further observation about the integer calculations described in this section: most of the multiplications use the coefficients 0, ± 1, and 2. None of these numbers actually require a real multiplication, instead they may be implemented as shifts and sign changes to increase the numerical efficiency even further. Figure 7-14 illustrates the multiplier cell, used in Chapter 6, mapped into the integer Pooh. This integer version of the cell is the same size as the original cell.

8. A Pooh System

This chapter describes the implementation of a system that supports the Pooh representation. The first section describes a Pooh embedded language system developed as the underlying representation for the Silicon Structure Project (SSP) Siclops silicon compiler. The second section presents Tigger: an interactive circuit editor that supports the Pooh representation and interactively detects and prevents GDR violations. The third section compares the Pooh approach to existing alternatives.

8.1 An Embedded Pooh

The original Pooh was implemented as an embedded language [Whitney 82], embedded in the programming language Mainsail [Wilcox 79]. It served as the base representation for the Siclops silicon compiler [Hedges 82]. This Pooh system is a set of procedures in the programming language Mainsail whose purpose is to aid in the definition of cells. There are two types of cells: leaf cells and composition cells. A leaf cell contains the interconnection of transistors and no references to other cells. A composition cell is a cell that contains instances of other cells and their interconnection. The various siclops subsystems communicated through blocks described in Pooh.

Overview of Siclops

The Siclops silicon compiler was based on Dave Johannsen's Bristle Block silicon compiler [Johannsen 81]. The design of siclops included the following goals:

1. An automatic wire routing system, including power and ground,
2. Modularity of the silicon compiler,
2. A flexible floorplan,
4. portability,
5. technology independence, and
6. support of simulation.

Figure 8-1 is a diagram of the complete siclops compiler. The automatic wire routing was handled by the General Interconnect (G.I.). The modularity of the system was met by implementing each subsystem as a stand-alone tool that communicated by using a consistent base level representation — Pooh. The flexible floorplan was possible because of the availability of a G.I. The portability was provided by the implementation language — Mainsail. The technology independence was addressed both by keeping any technology dependent information outside the high level subsystems, and by using the Pooh system, that supports multiple technologies. The simulation support was provided by a translation from the Pooh representation to a switch level simulator.

Siclops was designed and implemented in only six months in the spring of 1982. Consequently, not all of the pieces were completed. Most of the participants left shortly

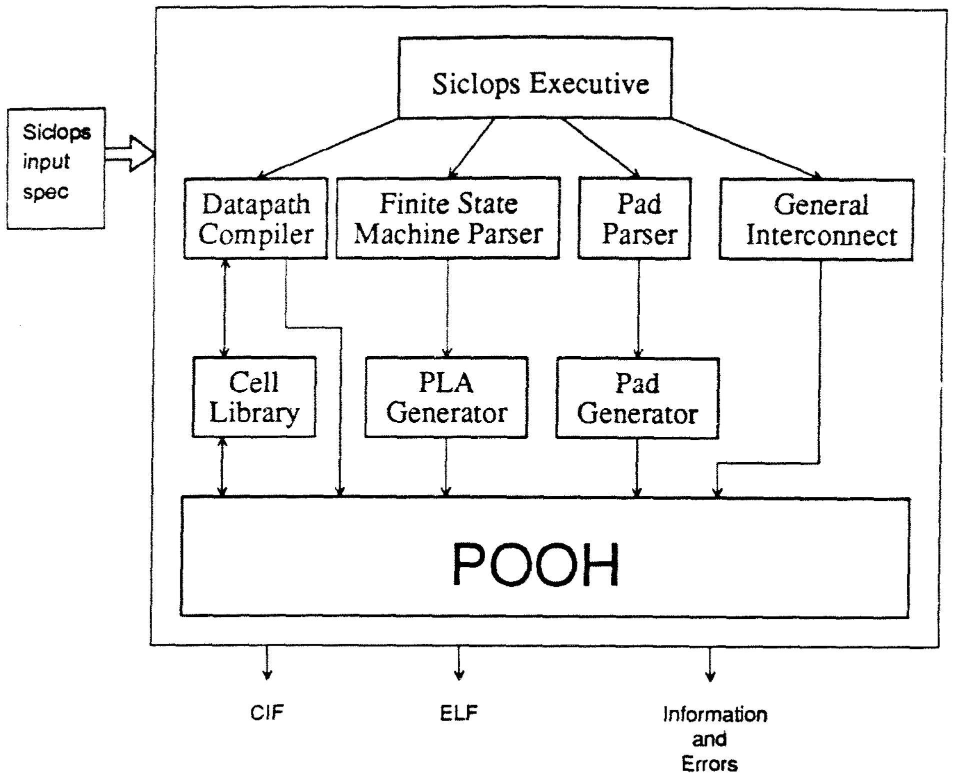


Figure 8-1: The Siclops Silicon Compiler

thereafter, thus a fully working version of Siclops was never completed. The following describes some of the subsystems that were completed and how they interfaced to Pooh.

The datapath compiler was designed and implemented by Ken Slater of Digital Equipment Corporation. A datapath generated by the compiler is composed of columns of functional elements. Each column consists of vertically stacked bit cells. The framework for the datapath is completely technology independent. The actual cells are described in a cell library. The Mainsail language supports run-time linking, thus a cell header is defined and used by the datapath compiler without any detailed knowledge of the individual cells. The datapath compiler queries each cell for its size and power requirements, and then adjusts the entire datapath based on the information supplied by the cells. Each cell informs the datapath compiler at runtime as to the parameters it understands, making each cell as flexible or inflexible as appropriate.

The original cell library was nMOS, supporting the Mead and Conway [Mead 80] design rules. Each cell is a Mainsail module, with a standard interface. The topology of the cell is described by a set of calls to the procedures which comprise the Pooh embedded language. The cells are formed into columns by invoking the Pooh composition system. Further calls into the composition system form the columns into a datapath. Pooh provides no stretching capability as the composition occurred. Instead, the datapath compiler controls the pitch of the cells. The compiler determines the placement of the seven buses that provide communication between the columns, based on the maximum size required by any of the cells used in a datapath. The topology of the cells in the cell library is defined in terms of these seven buses. Thus, rather than the Pooh system attempting to guess the optimal pitch of a cell, the datapath compiler performs stretching in a controlled manner appropriate to the application. Figure 8-2 is a plot of one of the datapaths generated by the Siclops silicon compiler.

The pad generator takes an input specification and generates a row of Pooh pads. The actual pads were designed by Tony Bell of Fairchild, and are parameterized both in the speed of the pads, and in the number of TTL loads the pad will drive. There are a set of high and low profile pads. Pads in Pooh are represented as connection points, with a very large radius and are round, as are all connection points. The GDR analysis of pads is no different than for any other contact: only the actual rules are different.

Figure 8-3 is a plot of seven low profile Pooh pads that were recently fabricated and successfully tested. These pads are VDD, ground, clock, input and output pads, are slow speed and drive a .5 TTL load. Other pads available in the Pooh pad library are a multiplexor and tristate pad.

The Embedded Language Pooh

The original Pooh [Whitney 82] used by Siclops was very limited in its scope. There were a set of functions available in Mainsail that allowed the definition of transistors paths, interconnections paths, and the ability to connect these elements together. There were a second set of functions that supported the interconnection of cells. The original Pooh supported both nMOS and cMOS/SOS.

The synthesis algorithms, described in Chapter 3 were the first set of functions to be supported. All primitive elements were constructed correct according to the current set of GDRs. If the embedded specification was inconsistent, the Pooh system reported an error and attempted to leave the circuit in a consistent state.

The original Pooh composition was implemented by Tom Hedges, from Calma Corporation, one of the Siclops participants. Pooh cells were defined with sides and ports, and composed by connecting two sides. The composition system supported logical verification of the connection between the two sides; and supported the syntactic composition described in Chapter 6.

The original Pooh supported two output formats: Caltech Intermediate Form (CIF) [Mead 80, Hon 80], and the Experimental Layout Format (ELF). CIF is a hierarchical geometrical description and is commonly used as a geometry interchange format. ELF is a file format developed at Caltech that includes all the pieces of information present in a Pooh circuit in an easy to parse but terse format.

The original simulation interface was written by Mike Schuster from Burroughs, another Siclops participant, and converted ELF to a Mossim specification [Bryant 82, Bryant 81]. ELF describes the interconnection of transistors and interconnection wires, but did not maintain the connectivity or propagate node information. The results of the original conversion attempt affected the way in which connectivity information was eventually maintained in the Pooh representation.

Later versions of the embedded Pooh supported the complete analysis algorithms described in Chapter 5. Since the embedded Pooh is simply an executable circuit description, the interface to the analysis routines is "batch" oriented. If Pooh detects a GDR violation, it reports an error, and continues processing the circuit. Since GDR errors may occur, Pooh cannot guarantee the GDR correctness of the circuit, rather it

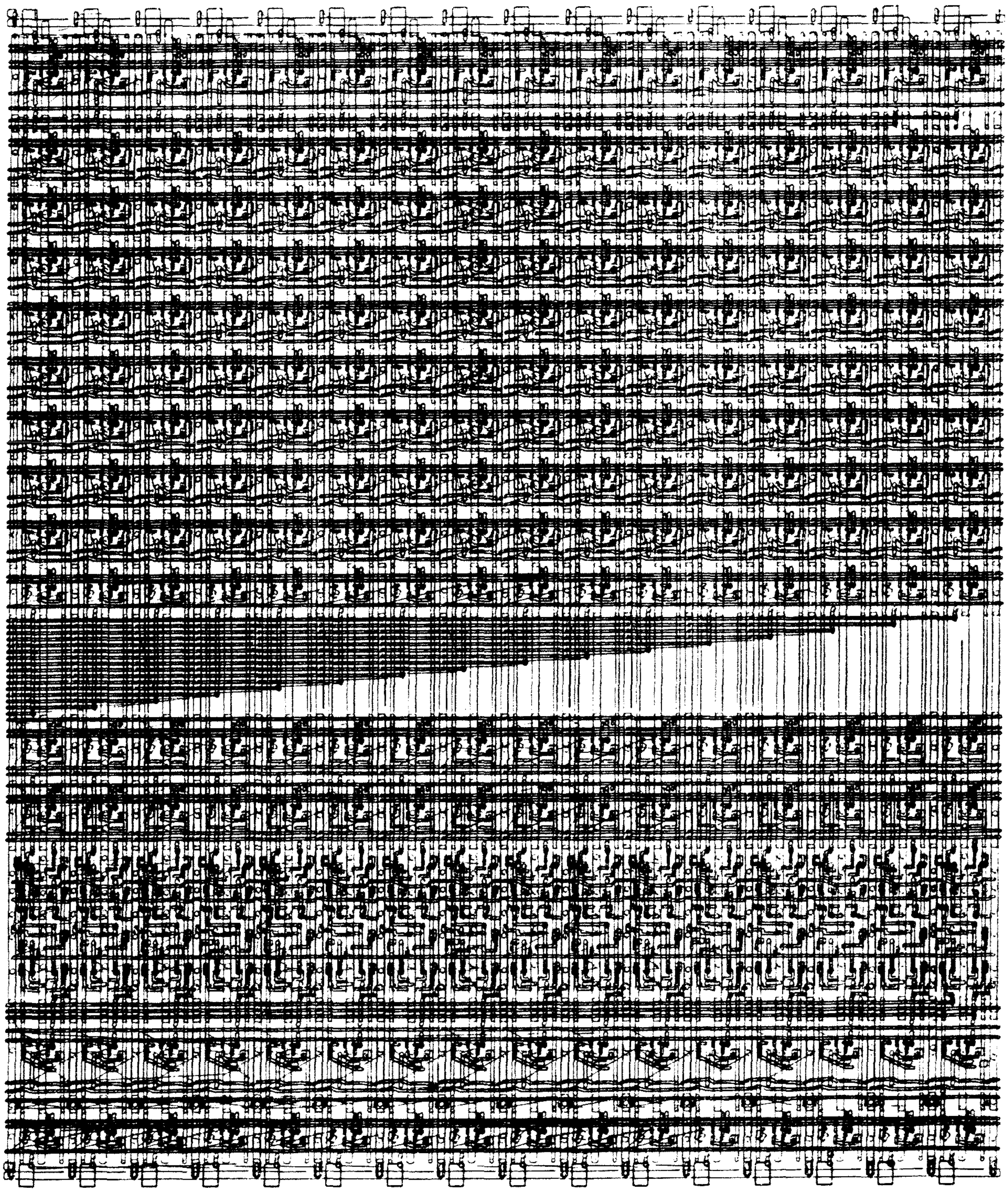


Figure 8-2: A Datapath Generated by the Datapath Compiler

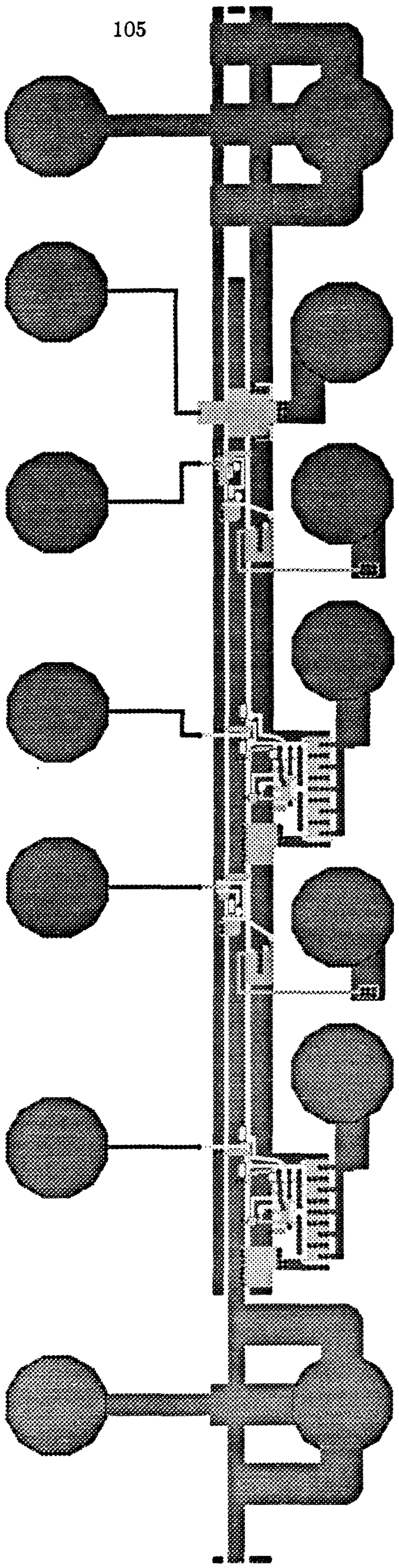


Figure 8-3: A Set of Pooh Pads

marks the circuit as correct or incorrect.

Shortly after the Siclops project, the embedded Pooh was extended to maintain the connectivity of a circuit, both within a cell and during composition, as described in Chapters 4 and 6. Pooh generates simulation input directly from the representation rather than going through a conversion program; since all the information is present, the simulation generation is straight-forward.

The embedded Pooh was a useful tool for developing the Pooh algorithms, and it provided a consistent interface to a larger system — Siclops. A language is, however, not a very convenient interface for describing the basic circuits. The system is slow to respond each time a change occurs, as is always true with a "batch" system.

8.2 Tigger: An Interactive Circuit Editor

Tigger is an interactive graphical circuit editor, written in Mainsail, that supports the Pooh representation. Individual cells are designed in Tigger, and cells are then composed with the embedded language composition system. Tigger provides immediate feedback to the designer as the transistors and the interconnection wires are positioned graphically, and guarantees the GDR correctness of the circuit by detecting the GDR violations as they occur, and not allowing the circuit to remain in an illegal state. Tigger allows the designer to edit the circuit by either removing elements or by moving individual points and segments. Tigger requires that the Pooh algorithms and data structures support fully dynamic behavior since the interactive program is constantly changing the circuit.

Tigger supports the description of transistor and interconnection paths and their connection, and uses constant graphical feedback to indicate the current *state* of the cell. Currently, Tigger is running on a Silicon Graphics, Inc. Iris workstation, although it has run on other workstations through the use of a different system-dependent graphics module. The graphical interface to the system is a three button mouse. The three buttons, during the creation of a path, map to: place a point, connect and make a transistor. Figure 8-4 illustrates several of the Tigger operations. The current path is "rubber banded", i.e., the path outline moves as the cursor moves, giving the user constant visual feedback as to the placement of the path. If a path moves "too close" to an existing element, the offending element is highlighted, indicating to the user the existence of a GDR violation. The designer may then move to a legal *state*, at which point the highlighted element returns to its normal display. The designer may also connect to or make a transistor with the offending element, using the two alternate mouse buttons, and if the situation is legal, the path continues on an optional new path segment. In fact the designer may indicate "connection" or "transistor" at any time during the creation of a path, including when the path starts, and the system will find the closest element, and attempt to make a connection or a transistor.

If the side of the current path "bumps into" an existing point, often it is possible for the path to "go around" the offending point. Figure 8-5 illustrates these Tigger interactions. Tigger highlights an element when a GDR exists between the current path and an existing element. If this element is a point, then Tigger attempts to "miss" the point with the appropriate GDR spacing rule. If the situation is legal, the path continues on a new path segment after creating a segment arc around the point. If the user moves to where the segment arc is no longer valid, then the arc is removed.

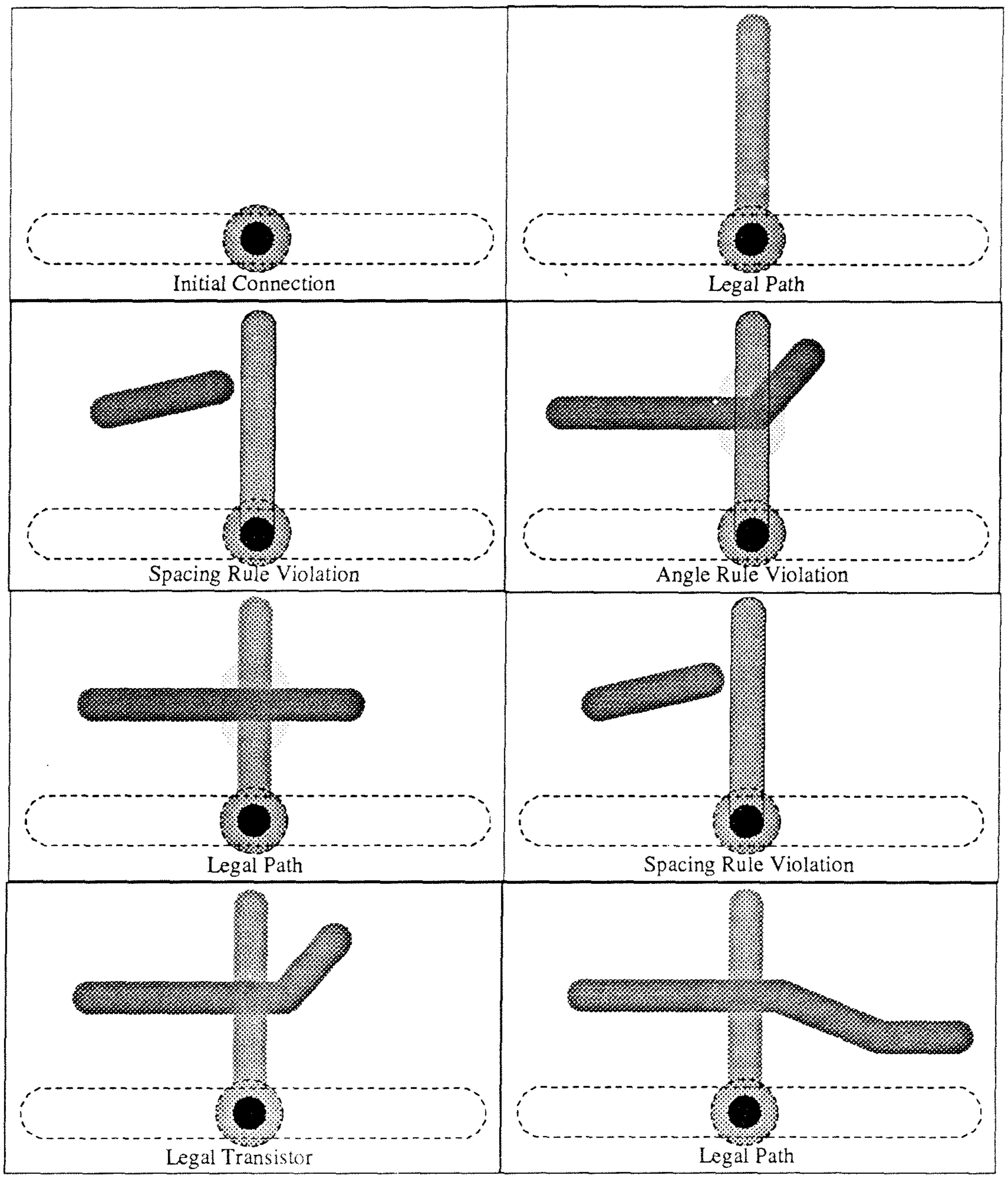


Figure 8-4: Tigger Interactions

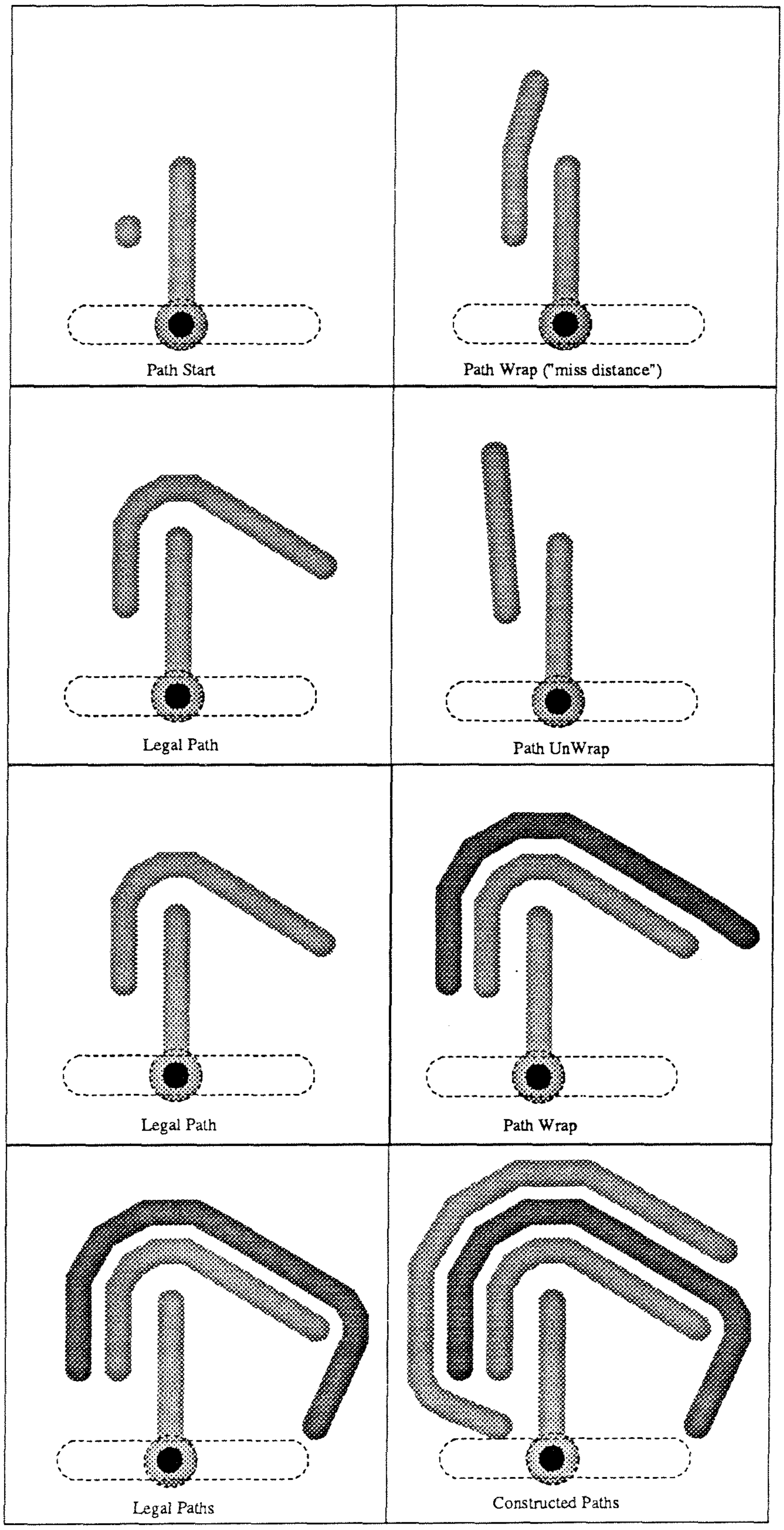


Figure 8-5: Tigger "missing" Interactions

Tigger and the Pooh Algorithms

The Pooh algorithms support Tigger, but the application is unique because Tigger is interactive. The circuit is dynamically evolving and the use of the algorithms reflect this constant change.

In Tigger, the synthesis and analysis algorithms interact in a unique way. Each time the current path segment moves a half unit, the analysis routines are invoked to determine if there is a GDR violation in this new configuration of the path. The synthesis algorithms are used to update the current path, often incorporating information provided by the analysis algorithms. If a GDR violation is detected, the user may connect to or make a transistor with the element in violation. Tigger connects the current path to the offending element, using the synthesis techniques, and then invokes the analysis routines to determine if this connection is legal. If it is legal, then the new segment is valid and the path optionally continues, otherwise the connecting segment is removed and the new offending element is highlighted. A similar interaction occurs if the analysis routines detect a GDR violation between the line of the current segment and a point. Tigger constructs a segment arc around the offending point, and then invokes the analysis routines to determine if the arc is legal. If the arc does not create a violation, the new segment remains, otherwise it is removed. Tigger calculates the dot product between the current path segment and the last segment and decides whether the last arc should be removed, using the algorithm described in Chapter 3.

The node synthesis routines allow Tigger to maintain the full circuit connectivity. Given an initial configuration for the cell, Tigger uses the node propagation algorithm, described in Chapter 4 to determine the connectivity. Tigger incrementally updates the connectivity each time a path is added to the circuit. If an element is deleted, or a transistor is placed on an existing path, the circuit connectivity is modified, and the node propagation algorithm is invoked.

Editing the Circuit

There are two types of editing functions currently supported in Tigger: delete and move. Delete allows the user to remove a path or a point from the circuit. Move allows a path segment or a point to be moved without changing the circuit connectivity. More sophisticated editing capabilities are desirable, but these two operations represent the basic functionality upon which more sophisticated editing functions depend. For example, currently one point or one path segment may be moved; eventually a user will be able to define a group of objects and move the entire group. Currently a single segment may be stretched, eventually the user will be able to stretch an entire cell.

Delete removes an interconnection path segment, a surround path, a transistor path or an unwanted point from the circuit. Individual path segments may be removed, but a transistor path and a surround path function as a single entity, thus the entire path is removed. Points may be removed if they are not critical to any path segments. For example, in nMOS, a buried contact cannot be removed if there is both a polysilicon path and a diffusion path connecting to the contact. But the same contact may be removed once the polysilicon path is removed.

The delete operation removes the designated element, synthesizes the connectivity, and then invoke the analysis routines on the elements directly connected to the deleted element. A GDR violation may occur between two elements that were connected through the deleted element, and are no longer connected. If there is a GDR violation, the circuit must be further modified by the user.

Move allows either a point or a path segment to move, while still maintaining the circuit connectivity. For a particular point, Tigger determines the segments that constrain the point. If there is only a single segment, then the point may move freely, similar to the current point in a new path. If there is more than one segment connecting to the point, but the segment lines are unconstrained, then the point moves freely, and the segments follow. If a segment line is constrained, then the point moves along the line. If more than one segment line is constrained, then the point is constrained and unable to move. For a particular segment, Tigger determines the points that constrain the segment, and the corresponding segments that constrain these points. The segment moves by moving its points along other connecting segments. If other segments are not present, then the point moves freely. In all editing functions, the movement is bound by the nearest constrained points in each direction. Figure 8-6 illustrates examples of editing functions, where a cross indicates the moving point in (a) and (b), and a line indicates the moving line in (c) and (d); the arrows indicate possible directions of movement.

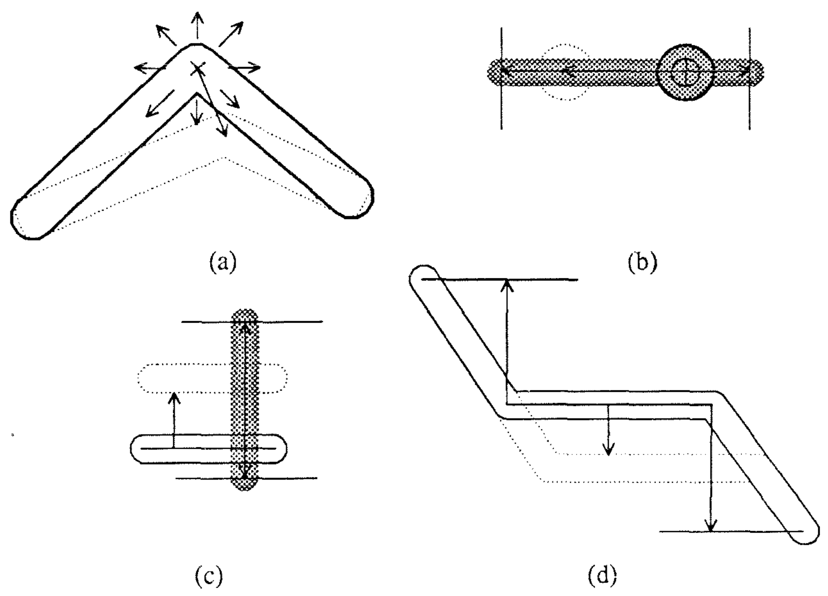


Figure 8-6: Possible Edit Functions

Design Examples

Describing circuits in Tigger is very different than interacting with the graphical layout editors currently available. The system is constantly calculating distances between elements and providing feedback to the designer as to the relative positions of wires and points, even when they are not on a typical grid. In order to use the embedded Pooh, designers had to first sketch the circuit on graph paper, thus the designs often followed traditional layout techniques. Unlike the embedded Pooh, Tigger removes the grid and allows the designer to see the constraining elements of the current circuit.

Tigger has been used to design several projects. The bulk cMOS bit serial multiplier presented in Chapter 6 was designed in Tigger. This multiplier was previously designed with a simple box layout tool called Wol [Mead 84] that supports strictly Manhattan geometry. A comparison of the area usage of these two designs for the same set of GDRs is shown in the following table. The units are lambda [Mead 80]. The actual set of cMOS design rules are listed in appendix II.

	Wol	Tigger
Top Row Height	50	51
Bottom Row Height	47	46
Column Width	92	81.5
Column Height	97	97
Area	8924	7905.5
Area Difference	1018.5	
Percentage Difference	11%	

The purpose of Pooh and Tigger is certainly not to try and squeeze the last micron out of a design, even though a designer will typical gain at least ten percent over a strictly Manhattan layout. To the contrary, Pooh is a general hierarchical circuit representation that allowed us to face into the issues of representing true circuits.

Researchers at Caltech are examining novel chip architectures that model biological systems; the current interests include modeling retinas [Mead 84] and exploring motion detection strategies [Tanner 84, Tanner]. A hexagon has many appealing characteristics as the form of the basic computational elements for systems that work in two dimensions. Figure 8-7 is the wiring diagram for the two-dimensional version of the retina. Currently the circuit design for the computational elements has not been completed, since the one-dimensional version recently returned from fabrication. Motion detection is another application of the hexagonal array. The number of interconnection wires between elements is reduced from four to three if a hexagon is used.

Tigger and the embedded Pooh share the same computational engine. The system is capable of supporting both batch and interactive design. Current unimplemented features are the integer Pooh and the GDR analysis during composition. The following table lists the components of the Pooh system, and the number of lines of source code in Mainsail.

Component	Source Lines
Data Structures	584
Analysis	4238
Pad Generator	2745
Technology	1184
Utilities	3924
Output Routines	2985
Tigger	7302
Node Synthesis	2543
Composition	1806
Total	27311

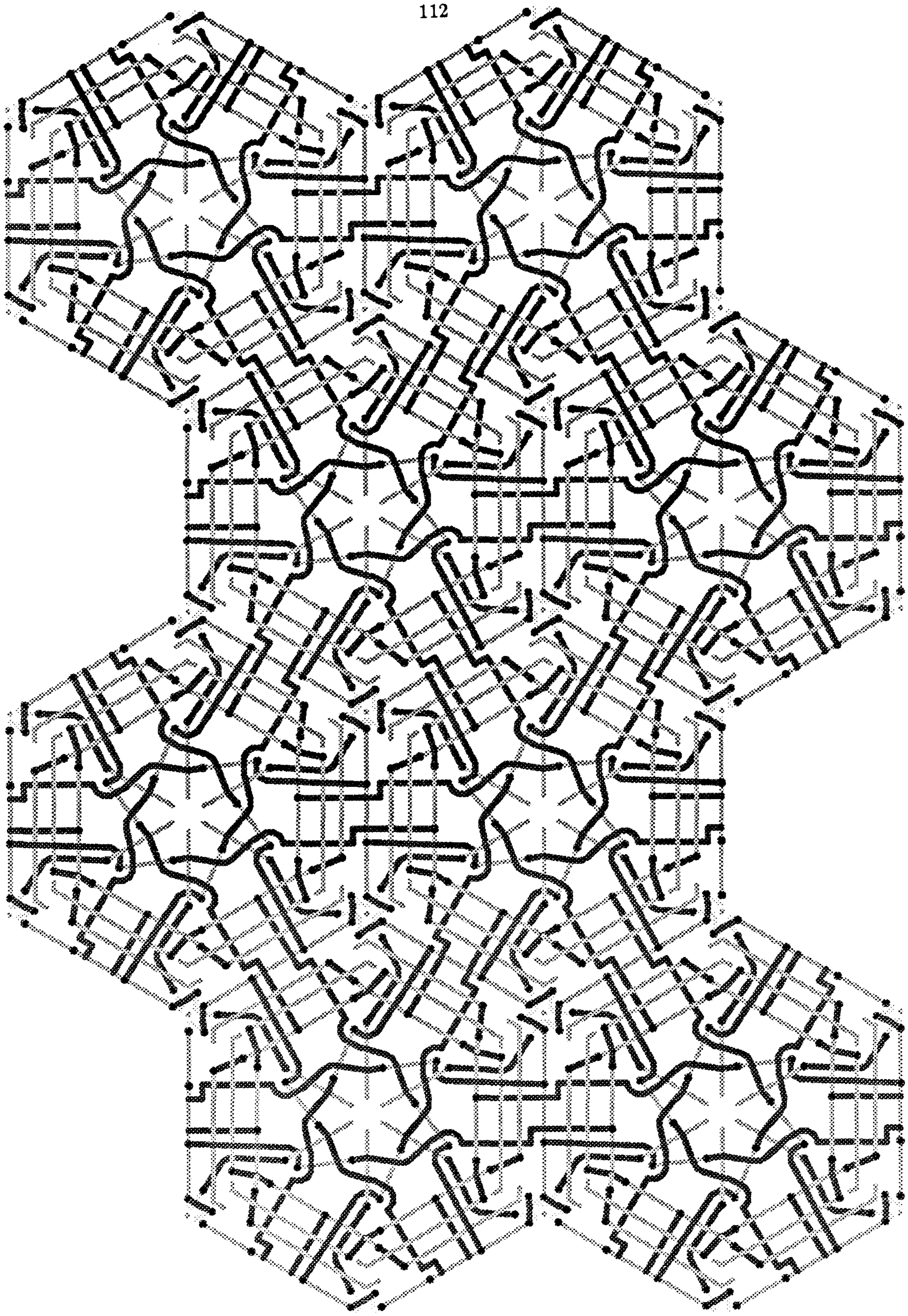


Figure 8-7: The wiring diagram for the Two Dimensional Retina

8.3 Comparison with Other Approaches

There are two complementary ideas in Pooh: the representation itself and the algorithms that perform operations on the representation. The algorithms are simple because care was taken in the design of the representation. Often these two ideas become entangled and inseparable. For example "sticks" [Williams 77] is a representation, where as compaction [Mosteller 81] is a set of algorithms that perform calculations on a Manhattan sticks representation. The two ideas are distinct; and thus treated separately in the following discussion.

Pooh is a "stick" representation, in that the devices of the fabrication technology are represented explicitly, and wires are connections between these devices. This semantic-based representation differs radically from the traditional approach of representing the mask layers. Many systems today still use layout as the representation of a design. These systems include Wol [Mead 84], Caesar [Ousterhout 81], and Earl [Kingsley 82]. More sophisticated tools such as Magic [Ousterhout 84] recognize where a transistor occurs, but represent the transistor area as another layout layer, and do not keep the transistor as an integral structure.

Other systems use a topological "sticks" representation. These systems commonly support a strictly Manhattan style of design, though there are exceptions. The most common model of a transistor is as a fixed symbol with three connection points for source, drain and gate. This transistor expands in two dimensions, based on two parameters length and width. Examples of this approach include the original sticks work [Williams 77] by John Williams, the ICSYS system [Buchanan 80], the Mulga system [Weste 81], the Rest systems [Mosteller 81] and the Cabbage system [Hsueh 80]. There are simple extensions to this transistor model that allow the definition of actual devices. For example, rather than using three single connection points, the transistor connections are lines along the three edges. One of the more sophisticated systems that supports this approach is Electric [Rubin 83]. A system that supports the definition of unique transistors is the inMos system [Barton 84]. Another extension to the transistor model is to push the complexity of the transistor layout into a fixed number of transistor symbols, similar to very small cells, where the transistor symbols still have three definitive connection points. This particular way of defining transistors is rather *ad hoc*, but circumvents the rigid approach supported by typical stick systems. Pooh, on the other hand, models transistors as algorithmic structures. The Pooh transistor points resemble the standard stick transistor. The Pooh transistor paths allow the description of legal devices in a general way. Each device is designed for the particular context. The path parameters are not limited to the length and width, but rather use the length, width and path center-line to algorithmically generate a unique transistor.

The geometrical algorithms used in conjunction with preventing GDR violations fall into three categories: (1) layout geometrical design rule checking, (2) compaction, and (3) sticks GDR detection. The first type is standard design rule checking. Rule checks are performed between edges of the layout. This approach is error prone, since the DRC must attempt to reconstruct part of the circuit in order to check many of the GDR rules; since the program is guessing the function of the layout, errors occur. DRC programs today often restrict the layout to Manhattan [Ousterhout 84], because in general, performing operations between the complex edges of polygons is a hard problem, subject to numerical inaccuracies.

Compaction is a batch oriented approach that does not detect GDR violation, but rather uses the GDR rules to determine the spacing between topologically ordered devices and interconnect. The representation for compaction is usually "sticks", and the algorithms are either graph based [Mosteller 81, Hsueh 80] or rely on a virtual grid [Weste 81]. All of these systems restrict the sticks to Manhattan, and separate the inherently two dimensional GDR representation into two one dimensional compactions. The advantage to compaction is that the technology rules may completely change, and designs are simply compacted with the new rules. The disadvantage is that the user ends up fighting the compactor. A designer has a model of the cell, and its interactions with other cells. Compactors inevitably end up moving cell elements around in an unsuitable manner. This problem is a common one in batch systems, and may or may not be acceptable for a particular problem.

The third approach, and the one supported by Pooh, is to detect GDR violations on a "sticks" representation. Electric also performs DRC on sticks, but although Electric does not restrict the cell elements to Manhattan wires, the DRC only works on Manhattan edges. The Electric DRC actually calculates distances between the edges of the layout contained in the sticks description rather than truly exploiting the representation. Pooh, by choosing circles and lines as its primitive structures, provides the luxury of allowing arbitrary angle geometry while still defining the GDR violation detection algorithms in terms of the transistors and wires, rather than mask layers. Tigger provides a unique interactive environment, where designers may interactively "compact" their cells in the manner appropriate to the application. Currently, work is in progress on full two-dimensional compaction of Pooh-like circuits using an annealing approach, with very interesting results [Mosteller].

All designs must eventually be described as a circuit in order to check the validity of the layout. Systems that maintain a layout representation must extract the circuits, whereas systems, such as Pooh, that maintain the connectivity may generate a circuit description trivially. Extractors tend to be error prone, since they must reconstruct the circuit from the layout, and often there is ambiguity.

The composition of cells is the single most important component in any system. The composition must be able to either detect or prevent GDR violations at the composition level. Layout editors, such as Magic, perform DRC during composition by flattening out the geometrical hierarchy at the cell interface; This approach to hierarchical GDR detection is well understood [Whitney 81, Scheffer 81]. There are many different forms of sticks composition. One approach is to ensure that each cell includes an interval of half the maximum spacing rule around the entire cell [Mosteller 82], thus not allowing any possible GDR violation, and also making the area of each cell subject to the worst case design rule. Another approach is to put the entire design on a virtual grid [Weste 81] and compact the entire chip. Earl [Kingsley 82], although it is a layout tool, supports a hierarchical constraint graph [Rowson 80]. The constraints do not include any GDR information, although hierarchical compaction using constraints on cells connectors (or ports) is implemented [Kingsley 84]. Electric does not do any GDR checking at the composition level. Pooh maintains the GDR relevant information at each level in the hierarchy, thus circumventing the need to instantiate the geometry during composition.

Although simple heuristics may eventually be applied when Pooh connects two cells, general stretching is usually unnecessary and inappropriate in many real designs.

Composition not only places two pieces of geometry together, but often assembles cells that are semantic units. Ports represent the interface to the cells and describe the signals the cells expect; ports are not truly represented by geometry alone. "Layout only" systems do not include any consistent mechanism for representing the cell. Systems such as Electric and Mulga do maintain a hierarchical description, rather than regenerating the description each time. Pooh is intrinsically linked to a hierarchical representation, and maintains the information necessary at the composition level in the cell external interface. This information is sufficient for composition, and creates a true abstraction. Pooh is a representation for hierarchical cells and provides a mechanism for topological design abstraction. This topological abstraction coupled with a semantic and timing abstraction of cells provides a complete and homogeneous design methodology.

Appendix I. Experimental Layout Format

The Experimental Layout Format (ELF)[†] is the file format used by the Pooh system to maintain the necessary information between design sessions. The file format expresses the notions inherent in the Pooh representation. ELF is considered object code generated by a higher level design system. In ELF, human readability was sacrificed for parsing efficiency, although the file format is ASCII for machine independence. An ELF file should never be created by a text editor.

An ELF file is based on the notion of lists. In fact the ELF file uses a LISP-like syntax. List indices are used to reference elements, rather than names. An ELF file consists of a list of cell definitions. Each cell definition is a list of: 1) Points, 2) Paths, 3) Instances of other cells, 4) Connectors, 5) Internal Connections, and 6) Constraints. Each element in a list may have a property list associated with it. A property provides additional information about the element not present in the formal syntax.

1 ELF Syntax

This section presents a formal definition of the ELF syntax. The following conventions are used: production rules use equals = to relate identifiers to expressions, non-terminal identifiers are surrounded by brackets, e.g., ⟨point⟩, terminal identifiers are always characters and are in a distinct font, e.g., `W`; double parenthesis (()) are used to group alternatives together; a vertical bar | indicates an or between alternatives; curly brackets {} indicate repetition any number of times including zero; square brackets [] indicate optional elements — i.e., zero or one repetitions; rules are terminated by a period. First the parsing syntax is presented, and then the full syntax including some redundant production rules present to indicate the semantics.

Parsing Syntax:

```
⟨Digit⟩          = (( 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 )).
⟨BlankC⟩         = ((   | linefeed | carriage return | tab ))
                   {⟨BlankC⟩}.
⟨NonBlankC⟩      = (( Any Character but a ⟨BlankC⟩, ' )).
⟨IdentChar⟩      = (( Any Character but a ⟨BlankC⟩, ⟨Digit⟩,
                   ', (, ), - )).
```

† The ELF format was originally a joint effort between John Tanner, Chris Kingsley, Richard Mosteller, and Telle Whitney, all graduate students at Caltech, and was intended as a way of communicating between various high level design tools. Once the Pooh system adopted it as its base format, the file format became more specific to the needs of representing Pooh, but many of the ideas expressed by the members of the original team remain in ELF.

```
⟨PositiveInteger⟩          = ⟨Digit⟩ {⟨Digit⟩} ⟨BlankC⟩.
⟨Integer⟩                  = [−] ⟨PositiveInteger⟩.
⟨CharacterSequence⟩        = ' {⟨NonBlankC⟩} ⟨BlankC⟩.
⟨Identifier⟩               = ⟨IdentChar⟩ ⟨BlankC⟩.
⟨Token⟩                    = (( ⟨CharacterSequence⟩ | ⟨Integer⟩ |
                             ⟨Identifier⟩ )).
⟨Element⟩                  = (( ⟨Token⟩ | ⟨List⟩ )).
⟨List⟩                     = ( {⟨Element⟩} ).
```

Full Syntax:

```
⟨Point⟩                    = ⟨Integer⟩ ⟨Integer⟩.
⟨Name⟩                     = ⟨CharacterSequence⟩.
⟨Index⟩                    = ⟨PositiveInteger⟩.
⟨Property⟩                 = ( ⟨Identifier⟩ {⟨Element⟩} ).
⟨Properties⟩               = ( {⟨Property⟩} ).
File                       = ( ⟨Version⟩ ⟨Date⟩ ⟨Time⟩ ⟨Technology⟩
                             ⟨Properties⟩ ⟨Cells⟩ ).
   ⟨Version⟩, ⟨Date⟩,
   ⟨Time⟩, ⟨Technology⟩    = ⟨CharacterSequence⟩.
⟨Cells⟩                    = ( {⟨CellDefinition⟩} ).
⟨CellDefinition⟩           = ( ⟨Name⟩ ⟨A⟩ ⟨B⟩ ⟨PtEntries⟩ ⟨Instances⟩
                             ⟨Paths⟩ ⟨Connectors⟩ ⟨IConnects⟩
                             ⟨Constraints⟩ [⟨Properties⟩] ).
   ⟨A⟩, ⟨B⟩                = ⟨PositiveInteger⟩.
⟨PtEntries⟩                = ( {⟨PtEntry⟩} ).
⟨PtEntry⟩                  = ( ⟨Point⟩ [⟨Properties⟩] ).
⟨Instances⟩                = ( {⟨Instance⟩} ).
⟨Instance⟩                 = ( ⟨Index⟩ ⟨Transform⟩ [⟨Properties⟩] ).
   ⟨Transform⟩             = ( [MirrorCommand] [RotateCommand]
                             [TranslateCommand] ).
   ⟨MirrorCommand⟩         = ( M ⟨BlankC⟩ (( X | Y )) ).
   ⟨RotateCommand⟩         = ( R ⟨BlankC⟩ ⟨Point⟩ ).
   ⟨TranslateCommand⟩      = ( T ⟨BlankC⟩ ⟨Point⟩ ).
⟨Paths⟩                    = ( {⟨Path⟩} ).
⟨Path⟩                     = ( ⟨PathType⟩ ⟨Properties⟩ ⟨PathPoints⟩ ).
   ⟨PathType⟩              = ⟨Identifier⟩.
⟨PathPoints⟩               = ( ⟨PathPoint⟩ {⟨PathPoint⟩} ⟨PathPoint⟩ ).
⟨PathPoint⟩                = ( ⟨TypePoint⟩ [Radius] [⟨Properties⟩] ).
   ⟨Radius⟩                = ⟨Integer⟩.
⟨Connectors⟩               = ( {⟨Connector⟩} ).
⟨Connector⟩                = ( ⟨Side⟩ ⟨TypePoint⟩ [⟨Properties⟩] ).
   ⟨Side⟩                  = (( W | E | S | N | I ⟨PositiveInteger⟩ ))
                             ⟨BlankC⟩.
⟨IConnects⟩                = ( {⟨IConnect⟩} ).
```

⟨IConnect⟩	=	(⟨TypePoint⟩ ⟨TypePoint⟩).
⟨TypePoint⟩	=	((P ⟨BlankC⟩ ⟨Index⟩ | I ⟨BlankC⟩ ⟨InstanceIndex⟩ ⟨ConnectIndex⟩)).
⟨InstanceIndex⟩, ⟨ConnectIndex⟩	=	⟨Index⟩.
⟨Constraints⟩	=	({⟨Constraints⟩}).
⟨Constraint⟩	=	(⟨TypePoint⟩ ((X | Y | Z)) ((= | >)) ⟨Integer⟩ ⟨TypePoint⟩).

2 Property Syntax and Semantics

The ELF file format provides a framework in which designs may be described. The notions of points, paths, connectors and instances form a design skeleton. In order to actually describe designs, information about the nature of the skeleton must somehow be present in ELF. Property lists are the mechanism through which ELF assigns interesting information to the various components in the file.

This section presents both the syntax and the semantics of the properties currently defined in ELF. The target for this particular set of properties is to describe pooh designs. Therefore the properties provide sufficient information for both the geometrical and topological elements of the design.

Each ELF property is surrounded by a set of parentheses, and consists of a single character identifier, and zero or more elements. Each element may either be a character sequence, an integer, an identifier, or a list.

⟨File⟩ Properties

The ELF format provides the capability of many different design tools communicating through a file or set of files. ELF provides information— it does not in any way guarantee the designs it describes. Often it is convenient to tag a file. This tag, for example could be interpreted as an indication of Geometrical Design Rule Integrity. The ⟨Generation⟩ property allows a design system to tag the file.

⟨Generation⟩	=	(G BlankC ⟨CharacterSequence⟩).
⟨FileProperty⟩	=	[⟨Generation⟩].

⟨CellDefinition⟩ Properties

Often a cell definition is simply represented by an "abutment box" and a set of connectors. An abutment box is a box that abuts to other cell instances during the composition process. An abutment polygon is a generalization of the abutment box—where a polygonal edge rather than a box edge abuts to other cell instances. The ⟨AbutBox⟩ and ⟨AbutPoly⟩ properties supply this information.

⟨AbutBox⟩	=	(B ⟨BlankC⟩ ⟨Box⟩).
⟨Box⟩	=	⟨Left⟩ ⟨Bottom⟩ ⟨Right⟩ ⟨Top⟩.
⟨Left⟩,⟨Bottom⟩, ⟨Right⟩,⟨Top⟩	=	⟨Integer⟩.
⟨AbutPoly⟩	=	(P ⟨BlankC⟩ ⟨PolyPoints⟩).
⟨PolyPoints⟩	=	({⟨Side⟩ ⟨Point⟩}).
⟨CellProperty⟩	=	((⟨AbutBox⟩ | ⟨AbutPoly⟩)).

⟨PtEntry⟩ **Properties**

ELF represents each important point in the design as a ⟨PtEntry⟩. A PtEntry always has a position, but PtEntries need type information to fully describe the point they represent. There are three possible types for a PtEntry: 1) a contact, 2) a transistor, and 3) a geometrical point. A contact may include more than one physical point, in which case the ⟨Index⟩ is used to differentiate between the points. If the contact is asymmetrical, then the ⟨Orientation⟩ property describes the orientation of the contact. A transistor point may have all the properties described in ⟨TranProperty⟩. Any point may have a ⟨PointName⟩ and a ⟨PointNodeNumber⟩.

```
⟨Contact⟩              = ( C ⟨BlankC⟩ ⟨ConDesc⟩ ).
   ⟨ConDesc⟩           = ⟨Contype⟩ [⟨Index⟩].
   ⟨ConType⟩           = ⟨Identifier⟩.
⟨Transistor⟩           = ( T ⟨BlankC⟩ ⟨TranDesc⟩ ).
   ⟨TranDesc⟩          = ⟨Trantype⟩ [⟨TranProperties⟩].
   ⟨Trantype⟩          = ⟨Identifier⟩.
⟨GeometricPoint⟩       = ( P ⟨BlankC⟩ ⟨Identifier⟩ ).
⟨PointName⟩            = ( N ⟨BlankC⟩ ⟨Name⟩ ).
⟨PointNodeNumber⟩      = (# ⟨PositiveInteger⟩ ).
⟨Orientation⟩          = ( O ⟨Integer⟩ ).
⟨PtEntryProperty⟩      = ((⟨Contact⟩ | ⟨Transistor⟩ | ⟨GeometricPoint⟩ |
                          ⟨PointName⟩ | ⟨PointNodeNumber⟩ |
                          ⟨Orientation⟩ )).
```

⟨Instance⟩ **Properties**

Instances of cells often have a name, stored as the ⟨InstName⟩ property.

```
⟨InstName⟩             = ( N ⟨BlankC⟩ ⟨Name⟩ ).
⟨InstanceProperty⟩     = [⟨InstName⟩].
```

⟨Path⟩ **Properties**

Paths may either be transistors or interconnection wires. An interconnection path may have a size, node number or width. The size indicates the relative capacitance of the path. A transistor path may have all the transistor properties ⟨TranProperty⟩; the gate node number of a transistor path is the ⟨NodeNumber⟩ property.

```
⟨Size⟩                 = ( K ⟨BlankC⟩ ⟨PositiveInteger⟩ ).
⟨NodeNumber⟩           = (# ⟨BlankC⟩ ⟨PositiveInteger⟩ ).
⟨PathWidth⟩            = ( W ⟨BlankC⟩ ⟨PositiveInteger⟩ ).
⟨PathProperty⟩         = ((⟨Size⟩ | ⟨NodeNumber⟩ | ⟨PathWidth⟩ |
                          ⟨TranProperty⟩ )).
```

Transistor Properties

Both PtEntries and Paths may be transistors. Any transistor may have a width, a length, a strength, a source and drain node number, and an initial and final extension property. The ⟨TranConn⟩ property contains the source and the drain node numbers. The ⟨TExt⟩ property is the initial and final extensions from the beginning and ending of the path to the start and the end of the gate regions, respectively.

```
⟨TranWidth⟩            = ( W ⟨BlankC⟩ ⟨PositiveInteger⟩ ).
```

```
⟨Length⟩             = ( L ⟨BlankC⟩ ⟨PositiveInteger⟩ ).
⟨Strength⟩           = ( S ⟨BlankC⟩ ⟨PositiveInteger⟩ ).
⟨TranConn⟩           = ( D ⟨BlankC⟩ ⟨PositiveInteger⟩ ⟨PositiveInteger⟩ ).
⟨TExt⟩               = ( E ⟨Integer⟩ ⟨Integer⟩ ).
⟨TranProperty⟩       = ((⟨TranWidth⟩ | ⟨Length⟩ | ⟨Strength⟩ |
                       ⟨TExt⟩ )).
⟨TranProperties⟩     = ( {⟨TranProperty⟩} ).
```

⟨PathPoint⟩ **Properties**

Each path consists of a sequence of path points. The ⟨ConnectionType⟩ property indicates the type of connection between the path and the point. If neither the path nor the point are transistors then the path may either connect to the point (`ConnectCenterLine`) or not connect to the point (`NotConnected`). If the point is a transistor point then the path may connect to the source (`ConnectSource`), drain (`ConnectDrain`) or gate (`ConnectCenterLine`) of the point. If the path is a transistor then the path may be unconnected (`NotConnected`) or the point is connected to the source (`ConnectToSource`), drain (`ConnectToDrain`) or gate (`ConnectCenterLine`). The ⟨PathType⟩ property indicates the point's position in the path. The point may: 1) start the path (`F`), 2) end the path (`L`), 3) end the segment (`S`), 4) occur along the segment (`I`), or 5) end the transistor region (`E`). The ⟨CircleIndex⟩ property indicate where on a discrete circle this segment ends.

```
⟨PointConnect⟩          = ( C ⟨BlankC⟩ ⟨ConnectionType⟩ ).
   ⟨ConnectionType⟩     = (( NotConnected | ConnectCenterLine |
                          ConnectDrain |ConnectSource |
                          ConnectToDrain | ConnectToSource )).
                        = (( 0 | 1 | 2 | 3 | 4 | 5 )).
⟨PathPointType⟩         = ( P ⟨BlankC⟩ ⟨PointType⟩ ).
   ⟨PointType⟩          = (( F | L | I | E | S )).
⟨CircleIndex⟩           = ( I ⟨BlankC⟩ ⟨Index⟩ ).
⟨PathPointProperties⟩   = (( ⟨PointConnect⟩ | ⟨PathPointType⟩ | ⟨CircleIndex⟩ )).
```

⟨Connectors⟩ **Properties**

Connectors may have properties designating their names and types. A ⟨Signal⟩ property is a global name. An ⟨InternalName⟩ is the name of the connector inside the cell. A ⟨ConnectName⟩ is the internal name of the connector to which this connector may connect. The ⟨Position⟩ is an index into a list of connectors with the same name. The ⟨ConnectorType⟩ indicates the type of this connector.

```
⟨Signal⟩             = ( S ⟨BlankC⟩ ⟨SignalTypes⟩ ).
   ⟨SignalTypes⟩     = (( 'vdd | 'gnd | 'phi1 | 'phi2 |
                       ⟨CharacterSequence⟩ ))⟨BlankC⟩.
⟨InternalName⟩       = ( N ⟨BlankC⟩ ⟨CharacterSequence⟩ ).
⟨ConnectName⟩        = ( C ⟨BlankC⟩ ⟨CharacterSequence⟩ ).
⟨Position⟩           = ( P ⟨BlankC⟩ ⟨PositiveInteger⟩ ).
⟨ConnectorType⟩      = ( T ⟨BlankC⟩ ⟨CType⟩ ).
   ⟨CType⟩           = (( vddType | gndType | Phi1Type | Phi2Type |
                       InputType | ControlInputType | OutputType |
                        TristateIOType | TristateOutType | BusType |
                        PreChargeBusType | UserType ))⟨BlankC⟩.
                      = (( 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 20 )).
⟨ConnectorProperty⟩   = (( ⟨Signal⟩ | ⟨InternalName⟩ | ⟨ConnectName⟩ |
                        ⟨Position⟩ | ⟨ConnectorType⟩ )).
```

3 Technology Dependent Information

This section describes the technology dependent identifiers. There are three different identifiers:

1) ⟨InterConnectType⟩,
2) ⟨TranType⟩, and
3) ⟨ConType⟩.

Each identifier is followed by a phrase in quotes that indicates the type, for example — D “Diffusion” indicates that D is the identifier for the diffusion layer.

nMOS Definitions

```
⟨InterConnectType⟩    = (( D “Diffusion” |
                        P “Polysilicon” |
                        M “Metal” )).
⟨TranType⟩            = (( A “Depletion Mode Transistor, diffusion centerline” |
                        B “Depletion Mode Transistor, poly centerline” |
                        E “Enhancement Mode Transistor, poly centerline” |
                        F “Enhancement Mode Transistor, diffusion centerline” )).
⟨PathType⟩            = (( ⟨InterConnectType⟩ | ⟨TranType⟩ )).

⟨ConType⟩             = (( B “Buried Poly To Diffusion” |
                        T “Butting Poly To Diffusion” |
                        P “Poly To Metal” |
                        D “Diffusion To Metal” |
                        X “Depletion Mode Buried Contact” )).
```

cMOS SOS Definitions

```
⟨InterconnectType⟩    = (( N “n-island” |
                        I “p-island” |
                        M “metal” |
                        P “polysilicon” )).
⟨TranType⟩            = (( G “p type of transistor, p-island centerline” |
                        H “p type of transistor, poly centerline” |
                        K “n type of transistor, n-island centerline” |
                        L “n type of transistor, poly centerline” )).
⟨PathType⟩            = (( ⟨InterConnectType⟩ | ⟨TranType⟩ )).
```

```
⟨Contype⟩          = (( N "n-island to poly" |
                        D "n-island to p-island" |
                        P "poly to p-island" |
                        T "n-island to poly to p-island" |
                        R "poly to n-island to p-island" |
                        L "n-island to metal" |
                        M "poly to metal" |
                        O "metal to p-island" )).
```

cMOS bulk Definitions

```
⟨InterconnectType⟩ = (( N "n-active" |
                        I "p-active" |
                        M "metal" |
                        P "polysilicon" |
                        Z "metal2" |
                        W "well" )).
⟨TranType⟩         = (( G "p type of transistor, p-active centerline" |
                        H "p type of transistor, poly centerline" |
                        K "n type of transistor, n-active centerline" |
                        L "n type of transistor, poly centerline" )).
⟨PathType⟩         = (( ⟨InterConnectType⟩ | ⟨TranType⟩ )).

⟨Contype⟩          = (( L "n-active to metal" |
                        M "poly to metal" |
                        O "metal to p-active" |
                        U "ohmic contact metal to well" |
                        S "ohmic contact metal to substrate" |
                        X "metal1 to metal2"|
                        V "poly to metal to metal2"|
                        W "n-active to metal to metal2"|
                        Z "p-active to metal to metal2")).
```

The following is the ELF for the synthesis example, figure 2-5, of Chapter 2. The example is fairly short, as ELF file go, but it illustrates most of the structures, and how they are used.

```
('3 'May-2-85 '16:38 'nmos ((G 'POOH))
(('synthesis 200 100 ((1000 1800((# 1)(P D)))(1900 1800((# 2)
(C P)))(2611 2112 ((# 3)(T A ((L 1200)(W 200)(D 1 4)))))
(3350 2150((# 5)(C D)))(3850 2150((# 4)(P D)))(1450 3300((# 6)
(P M)))(2200 3300((# 7)(C P)))(2950 3300((# 6)(C D)))(3550 3300
((# 6)(P M)))(3900 3400((# 8)(P P)))(4100 3400((# 8)(T A
((L 200)(W 200)(D 9 10)))))(4100 3200((# 9)(P D)))
(4100 3600((# 10)(P D)))(4300 3400((# 8)(P P))))
()
((M ((W 300)(# 6))((P 6 ((P F)(C 1)))(P 7 -650 ((P S)(I 11)))
(P 8 ((P S)(C 1)(I 6)))(P 9 ((P L)(C 1)(I 9)))))
(D ((W 200)(# 9))((P 12 ((P F)(C 1)))(P 11 ((P L)(C 3)(I 11)))))
(D ((W 200)(# 10))((P 11 ((P F)(C 2)))(P 13 ((P L)(C 1)(I 11)))))
(P ((W 200)(# 8))((P 10 ((P F)(C 1)))(P 11 ((P I)(C 1)))
(P 14 ((P L)(C 1)(I 8)))))
(A ((W 200)(# 3)(L 1200)(D 1 4)(E -100 -100))
((P 1 ((P F)(C 5)))(P 2 -700 ((P S)(I 10)))(P 3 ((P E)(C 2)))
(P 4 500 ((P S)(I 7)))(P 5 ((P L)(C 1)(I 11)))))
)
()
()
()
()
)))
```

Appendix II. Geometrical Design Rules

The pooh design rule file[†] describes technology specific information. This file is always coupled with a technology specific Mainsail module [Wilcox 79] that reads in this technology file, builds a data structure describing how to graphically display the contacts of the technology, and maintains any technology specific procedures. The technology information in the file is:

1) the valid interconnect paths,
2) the contacts,
3) the transistors,
4) path to path spacing rules,
5) path to contact spacing rules,
6) contact to contact spacing rules, and
7) angle rules between paths.

The spacing rules may be four different types:

d only applies if the paths are different electrical nodes, otherwise the paths may cross freely,

x applies regardless of the electrical node,

s one spacing rule applies if the paths are the same node, another if the paths are different nodes,

q one spacing rule for paths inside (distance<0), and another rule for paths outside (distance>0).

The format of the file is easy to parse, but not easy to characterize in BNF. The file is a series of lists and upper diagonal matrices. The following conventions are used: production rules use equals = to related identifiers to expressions, non-terminal identifiers are surround by brackets, e.g., ⟨item⟩, terminal identifiers are always in a distinct font, e.g., x; double parenthesis (()) are used to group alternatives together; a vertical bar | indicates an or between alternatives; curly brackets {} indicate repetition either any number of times including zero, or a specific number of times in a list; square brackets [] indicate optional elements — i.e., zero or one repetitions; rules are terminated by a period. Lists start with a number ⟨n⟩, and are followed by a set of ⟨n⟩ items. The syntax for a list is:

```
⟨List⟩                 =  ⟨n⟩ {⟨item⟩}.
```

[†] The format of the design rule file was a joint effort between Chris Kingsley and Telle Whitney, both graduate students at Caltech, and was intended to span two distinct design tools. Chris left shortly thereafter, and never used the design rule format, but the format remained intact.

Upper diagonal matrices are used to store interaction rules between elements. If the rules are between a list of ⟨n⟩ elements and a list of ⟨m⟩ elements, the rules are stored as a list of ⟨m⟩ elements, followed by a list of ⟨m-1⟩ elements, ... followed by a list of ⟨1⟩ element. The syntax for these matrices is:

⟨Matrix⟩	= ⟦⟨n⟩ ⟨m⟩ ⟨item⟩⟧.

The numbers ⟨n⟩ and ⟨m⟩ do not appear explicitly in the file, but rather are implied. Comments may appear anywhere, and are delimited by double quotes (").

The parsing syntax is:

⟨Digit⟩	= ((`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`)).
⟨BlankC⟩	= ((| `linefeed` | `carriage return` | `tab`)) {⟨BlankC⟩}.
⟨NonBlankC⟩	= ((`Any Character but a` ⟨BlankC⟩,)).
⟨IdentChar⟩	= ((`Any Character but a` ⟨BlankC⟩, ⟨Digit⟩, -)).
⟨Identifier⟩	= ⟨IdentChar⟩ ⟨BlankC⟩.
⟨PositiveInteger⟩	= ⟨Digit⟩ {⟨Digit⟩} ⟨BlankC⟩.
⟨Integer⟩	= [-] ⟨PositiveInteger⟩.
⟨Boolean⟩	= ((`T` | `F`)).
⟨CharacterSequence⟩	= NonBlankC {⟨NonBlankC⟩} ⟨BlankC⟩.
⟨Comment⟩	= "{((⟨BlankC⟩ | ⟨NonBlankC⟩))} ".
⟨SpacingRule⟩	= ((⟨PositiveInteger⟩ `d` | ⟨PositiveInteger⟩ `x` | ⟨PositiveInteger⟩ `s` ⟨PositiveInteger⟩ | ⟨Positiveinteger⟩ `q` ⟨Integer⟩)).
⟨ConnectAngle⟩	= ⟨Integer⟩ ~⟨Integer⟩.
⟨Token⟩	= ((⟨CharacterSequence⟩ | ⟨Identifier⟩ | ⟨Integer⟩ | ⟨PositiveInteger⟩ | ⟨Comment⟩ | ⟨Boolean⟩ | ⟨SpacingRule⟩ | ⟨ConnectAngle⟩)).
⟨List⟩	= ⟨n⟩ {⟨item⟩}.
⟨Matrix⟩	= ⟦⟨n⟩ ⟨m⟩ ⟨item⟩ ⟧.

Full Syntax:

⟨Index⟩	= ⟨PositiveInteger⟩.
⟨n⟩	= ⟨PositiveInteger⟩.
⟨Name⟩	= ⟨CharacterSequence⟩.
File	= ⟨A⟩ ⟨B⟩ ⟨CifLayers⟩ ⟨Intercons⟩ ⟨Transistors⟩ ⟨Contacts⟩ ⟨Spacing⟩ ⟨Angles⟩.
⟨A⟩, ⟨B⟩	= ⟨PositiveInteger⟩.
⟨CifLayers⟩	= ⟨n⟩ {⟨CifLayer⟩}. "Cif Name, minimum radius"
⟨CifLayer⟩	= ⟨CharacterSequence⟩ ⟨PositiveInteger⟩.
⟨Intercons⟩	= ⟨n⟩ {⟨Intercon⟩}. "Interconnect name, minimum radius, elf layer, path type, cif list"
⟨Intercon⟩	= ⟨CharacterSequence⟩ ⟨PositiveInteger⟩ ⟨Identifier⟩ ⟨PathType⟩ ⟨CifList⟩.

```
                        "center-line path, or surround path"
    ⟨PathType⟩        = (( C | S )) ⟨NonBlankC⟩.
    ⟨CifList⟩         = ⟨n⟩ ⟨CifEntry⟩.
                        "Cif layer index, layer radius offset,
                        layer end extension"
    ⟨CifEntry⟩        = ⟨Index⟩ ⟨PositiveInteger⟩ ⟨PositiveInteger⟩.
  ⟨Transistors⟩       = ⟨n⟩ {⟨Transistor⟩}.
                        "Transistor name, gate layer index, elf gate
                        layer, gate width, minimum gate extension,
                        channel layer index, elf channel layer,
                        minimum channel width,
                        minimum channel extension, cif list"
  ⟨Transistor⟩        = ⟨CharacterSequence⟩ ⟨PositiveInteger⟩ ⟨Identifier⟩
                        ⟨PositiveInteger⟩ ⟨PositiveInteger⟩ ⟨PositiveInteger⟩
                        ⟨Identifier⟩ ⟨PositiveInteger⟩ ⟨PositiveInteger⟩
                        ⟨CifList⟩.
                        "number of contacts, contact cif file,
                        list of contacts "
  ⟨Contacts⟩          = ⟨m⟩ ⟨Charactersequence⟩ {⟨Contact⟩}.
                        "contact name, elf name, number of connection
                        points, legal connections, legal angles for
                        legal connections, cif contact number"
  ⟨Contact⟩           = ⟨CharacterSequence⟩ ⟨Identifier⟩ ⟨PositiveInteger⟩
                        ⟨LegalC⟩ ⟨LegalA⟩ ⟨PositiveInteger⟩.
                        "for each interconnect layer,
                        if there is a valid connection"
         ⟨LegalC⟩     = {⟨Boolean⟩}.
                        "the legal angle of connection for each valid
                        connection"
         ⟨LegalA⟩     = {⟨ConnectAngle⟩}.
  ⟨Spacing⟩           = ⟨PathPathSpace⟩ ⟨PathConSpace⟩ ⟨ConConSpace⟩.
                        "The spacing rules between the ⟨n⟩ paths"
  ⟨PathPathSpace⟩     = ⟦⟨n⟩ ⟨n⟩ ⟨SpacingRule⟩ ⟧.
                        "The spacing rules between the ⟨n⟩ paths and
                        ⟨m⟩ contacts"
  ⟨PathConSpace⟩      = {⟨SpacingRule⟩}.
                        "The spacing rules between the
                        ⟨m⟩ contacts"
  ⟨ConConSpace⟩       = ⟦⟨m⟩ ⟨m⟩ ⟨SpacingRule⟩ ⟧.
                        "The minimum connection angle between paths"
  ⟨Angles⟩            = ⟦⟨n⟩ ⟨n⟩ ⟨Integer⟩⟧.
```

The following is an example GDR file for cMOS Bulk:

```
150 10
"cif layers #layers, layer name, min radius"
```

```
9 CM 15 CP 10 CD 10 CM2 15 CC 10 CW 10 CS 15 CV 10 CG 10
"interconnect types #layers, layer name, min radius,
elf layer, C(centerline path)/S(surround path)
#cif layers, –cif layer, min cif radius""
6

metal   15 M C 1 1 0 15
metal2  15 Z  C 1 4 0 15
poly    10 P  C 1 2 0 10
ndiff   10 N  C 1 3 0 10
pdiff   10 I   C 2 3 0 10 7 20 30
pwell   0   W S 1 6 0 10

"transistor types #trans, transistor type,
gate layer, elf gate name, min gate width, min ext gate,
chan layer, elf chan name, min chan width, min chan ext,
#cif layers, –cif layer distance beyond gate length,
distance beyond gate width" extralayer"
2
ptype 3 H 20 25 5 G 20 20 3 2 0 25 3 20 0 7 40 20 3
ntype 3 L 20 25 4 K 20 20 2 2 0 25 3 20 0 0


"contact types #contacts, cif file of contact description,
contact name, elf contact name, #connection points,
array of legal connections for each connection type,
legal angle of connection for each legal connection,
cif symbol number"
9 scmoscon.cif

pm       M  1 TFTFFF 0~0 0~0       1
pdm      O  1 TFFFTF 0~0 0~0       2
ndm      L  1 TFFTFF 0~0 0~0       3
ohmicp U  1 TFFFFT 0~0 0~0       4
ohmicn S  1 TFFFFF 0~0             5
mm2     X  1 TTFFFF 0~0 0~0       6
pmm2   V  1 TTTFFF 0~0 0~0 0~0 7
ndmm2 W 1 TTFTFF 0~0 0~0 0~0 8
pdmm2 Z  1 TTFFTF 0~0 0~0 0~0 9

"spacing rules codes
d(only applies if different electrical nodes, otherwise
they may cross freely), x(applies regardless of
electrical nodes), s(one spacing rule if same node and another
for different nodes) and q(surround distance - one
distance¿0 - outside, one distance¡0 - inside)"
```

```
"metal metal2 poly ndiff pdiff  pwell   ptype  ntype"
 30d    0x     0x   0x    0x   0x      0x     0x      "metal"
        40d    0x   0x    0x   0x      0x     0x     "metal2"
              20d   10x   10x  0x      20x    20x      "poly"
                    40d  120x  40q-40  120x   40x      "ndiff"
                          40d  80q0    40x   120x      "pdiff"
                               80d     80q0   0q-40   "pwell"
                                       20x   120x     "ptype"
                                              20x     "ntype"
"metal metal2 poly  ndiff   pdiff  pwell    ptype   ntype"
 50d    0x    50d   40x     40x    0x       40x     40x         "pm"
 50d    0x    30x   140x    60d    100q0    60s30   140x       "pdm"
 50d    0x    30x   60d     140x   60q-60   140x    60s30      "ndm"
 50d    0x    30x   50s30   120d   40q-40   120x    50x    "ohmicp"
 50d    0x    30x   90d     60x    50q0     60x     90x    "ohmicn"
 50d    60d   0x    0x      0x     0x       0x      0x         "mm2"
 55d    65d   50d   40x     40x    0x       40x     40x       "pmm2"
 55d    65d   35x   65d     145x   65q-65   145x    65s30   "ndmm2"
 55d    65d   35x   145x    65d    105q0    65s30   145x    "pdmm2"

 "pm    pdm    ndm    ohmicp ohmicn  mm2    pmm2    ndmm2  pdmm2"
 80s40  70s60  70s60  70s60  70s60   70s30  80s40   75s55  75s55        "pm"
        80s40  160x   140x   80s60   70s30  75s60   165x   85s40       "pdm"
               80s40  80s50  110x    70s30  75s60   85s40  165x        "ndm"
                      80x    90x     70s30  75s60   75x    145x     "ohmicp"
                             80x     70s30  75s60   115x   85x      "ohmicn"
                                     80s50  85s55   85s55  85s55       "mm2"
                                            80s60   90s65  90s65      "pmm2"
                                                    90s60  170x      "ndmm2"
                                                           90s60     "pdmm2"

"angles for path to path"
 "metal metal2 poly ndiff pdiff pwell ptype ntype"
     90     0     0    0     0     0     0     0 "metal"
           90     0    0     0     0     0     0 "metal2"
                 80   85    85     0    90    90  "poly"
                      70     0     0     0    90  "ndiff"
                            70     0    90     0  "pdiff"
                                  90     0     0  "pwell"
                                        90    90  "ptype"
                                              90  "ntype"
```

Glossary

arc radius	a signed radius that defines a directed arc. A negative radius indicates a clockwise arc and a positive radius indicates a counter-clockwise arc. The absolute value of the radius *indicates the circle radius that defines the arc.*
cell	either a leaf cell or a composition cell.
cell side	defined by a line equation, the cell side is a directed line that forms one side of the polygon that defines the cell boundary.
circuit	a set of interconnected transistors.
composition cell	recursively defined as a set of interconnected transistors and instances of other cells.
connection point	a point where a connection occurs between one or more path segments. If there is only one segment, or if the segments are of the same path type, then the point is a geometrical point. If the segments are of different path types, then the point is a contact.
contact	a technology specific connection point between two or more paths with different path types.
current *state*	the current topology of a cell in the process of being updated. A cell is updated by incrementally adding instances of other cells.
external interface	The information along the cell perimeter necessary during composition. This information is the cell sides and ports, and the paths and points occurring within the applicable design rule spacing of the cell boundary.
instance	a placed version of a cell, where the placement information is a two-dimensional graphics transformation matrix. The matrix incorporates rotation, mirroring and translation information.
internal connection	a connection between two ports on two distinct instances in the current cell.
GDR	Geometrical Design Rules are the rules dictated by the fabrication technology that indicate how to construct transistors, wires, contacts, and the valid spacing and angle rules between these elements.
leaf cell	a set of interconnected transistors, with ports.

merge path	a single segment path whose centerline occurs along one of the sides of the cell boundary, and whose width is greater than minimum. During composition, if the two merge paths along the composible sides are equivalent, then they are merged into a single path, otherwise the composition is not valid.
"miss" distance	the arc radius derived by Pooh, dictated by the GDRs, that indicates the distance a segment goes around a point.
node number	an integer which indicates the electrical node an element is connected to. Two elements are connected if and only if the element node numbers are equivalent.
path	a sequence of path segments.
path segment	is defined by a line segment and an arc. If the arc radius is zero, then the segment is equivalent to the line segment. If the arc radius is non-zero, then it defines the end point of the line segment as the tangent point to a circle of radius r, centered at the segment end point. The beginning point of the line segment is given by the arc of the previous segment in the path definition.
path type	a technology specific tag that indicates which of the allowable types a particular path is an instance of.
port	an external connection point that indicates both the position and the type of signal a cell expects. During composition, cell ports on two or more cells are connected.
river router	A program that accepts two parallel linear lists of ports and connects the two port sets using a single interconnection layer, i.e., there are no wires that cross.
silicon compiler	a software system that accepts as input a high level structural or functional specification, and automatically produces both the geometry and a simulation model of a chip that implements the specification.
pad	a connection point between a chip and the outside world.
transformation matrix	a two dimensional matrix (or three, but not for integrated circuit design) that indicates the positioning information for an element. Pooh uses these matrices to place cells. The matrix incorporates rotation, mirroring and translation information in six numbers.
transistor	a technology specific device with three distinct nodes: source, drain and gate, where if the gate is "on", the source is connected to the drain.

Bibliography

[Aho 74] Aho, A.V., Hopcroft, J.E., Ullman, J.D.,
The Design and Analysis of Computer Algorithms,
Addison-Wesley, Reading, Massachusetts, 1974.

[Baird 77] Baird, H.S.,
Fast algorithms for LSI Artwork Analysis,
Proceedings of 14th D.A. Conference, pages 303-311, June, 1977.

[Baker 80] Baker, C.M., and Terman, C.,
Tools for Verifying Integrated Circuit Designs,
Lambda Magazine 4th Quarter:22-30, 1980.

[Barton 80] Barton, E.E., Buchanan, I.,
The Polygon Package,
Computer Aided Design 12(3):3-11, January, 1980.

[Barton 84] Barton, E.E.,
private communication, July, 1984.

[Bentley 80] Bentley, J.L., Haken, D., and Hon, R.W.,
Statistics on VLSI Designs,
Technical Report # CS-80-111, Carnegie-Mellon University, April, 1980.

[Bryant 81] Bryant, R.E,
A Switch-Level Simulation Model for Integrated Logic Circuits,
PhD Thesis, Massachusetts Institute of Technology, 1981.

[Bryant 82] Bryant, R., Schuster, M., and Whiting, D.,
Mossim II: A Switch-Level Simulator for MOS LSI User's Manual,
Technical Report # 5033, California Institute of Technology, August, 1982.

[Buchanan 80] Buchanan, I.,
Modeling and Verification in Structured Integrated Circuit Design,
PhD Thesis, Computer Science University of Edinburgh, 1980.

[Chen 83] Chen, M.C., and Mead, C.A.,
A Hierarchical Simulator Based on Formal Semantics,
Proceedings of The Third Caltech Conference on VLSI, March, 1983.

[Dijkstra 76]	Dijkstra, E.W., **A Discipline of Programming,** Prentice-Hall, Inc., Englewood Cliffs, N.J., 1976.
[Hedges 82]	Hedges, T.S., Slater, K.H., Clow, G.W., Whitney, T.E., **The Siclops Silicon Compiler,** Proceedings of IEEE ICCC, pages 277-280, September, 1982.
[Hon 80]	Hon, R.W., and Sequin, C.H., **A Guide to LSI Implementation,** Technical Report # SSL-79-7, Palo Alto Research Centers, January, 1980.
[Hsueh 80]	Hsueh, M., **Symbolic Layout and Compaction of Integrated Circuits,** PhD Thesis, University of California, Berkeley, 1980.
[Jensen 74]	Jensen, K., Wirth, N., **PASCAL User Manual and Report, second edition,** Springer-Verlag, New York, 1974.
[Johannsen 81]	Johannsen, D.L., **Silicon Compilation,** PhD Thesis, California Institute of Technology, 1981.
[Kingsley 82]	Kingsley, C., **Earl: An Integrated Circuit Design Language,** Master's Thesis, California Institute of Technology, June, 1982.
[Kingsley 84]	Kingsley, C., **A Hierarchical Error-Tolerant Compactor,** Proceedings of 21st D.A. Conference, pages 126-132, June, 1984.
[Lin 84]	Lin, T-M., **A Hierarchical Timing Simulation Model for Digital Integrated Circuits and Systems,** PhD Thesis, Computer Science Dept., California Institute of Technology, 1984.
[McGrath 80]	McGrath, E.J., and Whitney, T.E., **Design Integrity and Immunity Checking: A New Look at Layout Verification and Design Rule Checking,** Proceedings of 17th D.A. Conference, pages 263-268, June, 1980.
[Mead 80]	Mead, C., and Conway, L., **Introduction to VLSI Systems,** Addison-Wesley, Reading, Massachusetts, 1980.
[Mead 83]	Mead, C.A., **Structural and Behavioral Composition of VLSI,**

Proceedings of International Conference on VLSI, Trondheim, Norway, pages 3-7, August, 1983.

[Mead 84] Mead, C.,
The Wolery,
Technical Report # 5113:TR:84, California Institute of Technology, January, 1984.

[Mead 84] Mead, C., Mohowald, M.,
An Electronic Model of the Y-System of Mamalian Retina,
Technical Report # 5144:DF:84, California Institute of Technology, 1984.

[Mead 85] Mead, C.A, Wawrzynek, J.,
A Discipline for cMOS Design: An Architecture for Sound Synthesis,
Proceedings of the Chapel Hill Conference on VLSI, 1985.

[Moore 79] Moore, G.,
Are We Really Ready for VLSI?,
Proceedings of Caltech Conference on VLSI, pages 3-14, January, 1979.

[Mosteller 81] Mosteller, R.C.,
Rest: A Leaf Cell Design System,
Master's Thesis, California Institute of Technology, December, 1981.

[Mosteller 82] Mosteller, R.C.,
An Experimental Compostion Tool,
Proceedings of Conference on Microelectronics,
The Institution of Engineers, Australia, 1982.

[Mosteller] Mosteller, R.C.,
PhD Thesis, in preperation.

[Nagel 75] Nagel, L.W.,
SPICE: A Computer Program to Simulate Semiconductor Circuits,
Technical Report # ERL-M520, Electronics Research Laboratory, University of California, Berkeley, 1975.

[Newman 79] Newman, W.M.,Sproull, R.F.,
Principles of Interactive Computer Graphics, second edition,
McGraw-Hill Book Company, New York, 1979.

[Ng 84] Ng, T-K.,
A Graph Model and the Embedding of MOS Circuits,
Master's Thesis, California Institute of Technology, 1984.

[Ousterhout 81] Ousterhout, J.K.,
VLSI Design,
Fourth Quarter:34-38, 1981.

[Ousterhout 84] Ousterhout, J.K., Hamachi, G.T., Mayo, R.N., Scott, W.S., Taylor, G.S.,
Magic: A VLSI Layout System,
Proceedings of 21st D.A. Conference, pages 152-159, June, 1984.

[Rowson 80] Rowson, J.,
Understanding Hierarchical Design,
PhD Thesis, California Institute of Technology, 1980.

[Rubin 83] Rubin, S. M.,
An Integrated Aid for Top-Down Electrical Design,
Proceedings of International Conference on VLSI, Trondheim, Norway, pages 63-72, August, 1983.

[Scheffer 81] Scheffer, L.K.,
A Methodology for Improved Verification of VLSI Designs without Loss of Area,
Proceedings of the 2nd Caltech Conference on VLSI, January, 1981.

[Sutherland 78] Sutherland, I.E.,
The Polygon Package,
Technical Report # 1438, California Institute of Technology, February, 1978.

[Tanner 84] Tanner, J.E., Mead, C.,
A Correlating Optical Motion Detector,
Proceedings of the MIT Conference on VLSI, January, 1984.

[Tanner] Tanner, J.E.,
PhD Thesis, in preperation.

[Tompa 80] Tompa, M.,
An Optimal Solution to a Wire-Routing Problem,
Proceedings of the Twelfth annual ACM Symposium on Theory of Computing, April, 1980.

[Weste 81] Weste, N., Ackland, B.,
A Pragmatic Approach to Topological Symbolic IC Design,
Proceedings of Very Large Scale Integration, University of Edinburgh, Edinburgh, Scotland, pages 117-129, August, 1981.

[Whitney 81] Whitney,T.E.,
A Hierarchical Design Rule Checker,
Master's Thesis, California Institute of Technology, May, 1981.

[Whitney 82] Whitney, T., and Hedges, T.,
Pooh User's Manual,
Technical Report # 5029, California Institute of Technology, July, 1982.

[Whitney 83] Whitney, T., and Mead, C.A.,
Pooh: A Uniform Representation for Circuit Level Designs,
Proceedings of International Conference on VLSI, Trondheim, Norway, pages 401-411, August, 1983.

[Wilcox 79] Wilcox, C.R., Dageforde, M.L., and Jirak, G.A.,
Mainsail(TM) Language Manual Version 4.0,
Xidak, 1979.

[Williams 77] Williams, J.,
Sticks—A New Approach to LSI Design,
Master's Thesis, Massachusetts Institute of Technology, June, 1977.

[Wolf 83] Wolf, W., Newkirk, J., Mathews, R., and Dutton, R.,
Dumbo, A Schematic-to-Layout Compiler,
Proceedings of Third Caltech Conference on VLSI, pages 379-393, 1983.