A Comparison of Tile-Based Wire Representations for Interactive IC Layout Tools

David Wallace

ABSTRACT

Several wire representations based on "corner stitching," a data-structuring technique developed by John Ousterhout [Oust84], are compared. Three classes of wire representation are defined: *skeletal* representations, which model a wire as a chain of connected line segments; *purely physical* representations, which model the space occupied by the material of a wire, and *directed box* representations, which attempt to combine the best features of the other two classes. WICRD, an experimental prototype interactive wire manipulation system based on a directed box representation, is described. Each class of wire representations has its pros and cons, but the directed box representations turned out to be more limited than expected in their ability to express the ways in which wires connect to other objects. Because of these limitations, the use of directed box representations for tile-based interactive IC layout tools is not recommended.

A Comparison of Tile-Based Wire Representations for Interactive IC Layout Tools

David Wallace

1. Introduction

In the custom IC design environment of today, wire routing can be a difficult and often time-consuming task. Just doing the initial routing by hand accounts for much of the total design time. Changes to this routing, due to engineering changes or errors in the initial layout, require additional time. Even though only a few wires may need to be added or moved, creating additional space in a layout that has already been routed is a slow and painful process. Existing layout editors, such as KIC [Kell82a] and Caesar [Oust81], provide minimal support for this critical function.

Another approach would be to design in a very loose environment with plenty of free space for additional wiring, and then compact the layout after the design is completed, using a compactor such as CABBAGE [Hsue79] or PYTHON [Bale82]. These programs are global compactors that operate on the entire circuit at a time. To modify an already compacted layout, the entire layout must be expanded, modified, and then recompacted.

A better solution to this problem is a program that enables a designer to make small incremental changes to a layout quickly. Such a program can include two strategies to speed up such changes. One is to restrict the area affected by a change as much as possible, unlike global compactors which always operate on the entire circuit. The other is to keep the layout information in a form that allows a program to quickly obtain a description of the local area surrounding the change.

How should the layout information be organized? In [Oust84], John Ousterhout proposes a data-structuring technique called "corner stitching" for Manhattan layouts, which, he argues, is

superior to traditional data structures based on linked lists, bins, or neighbor pointers. In this technique, the layout space is modeled as one or more planes partitioned into rectangular tiles, which are stitched together by pointers at their corners (see Figure 1). This figure shows eight pointers per tile; Ousterhout shows how this can be reduced to four pointers per tile without loss of generality. Corner stitching enables a program to search and update the tile structure quickly, particularly if these operations occur near the object of the previous operation, as is often the case. The corner stitching paper [Oust84] describes how the operations of point finding, neighbor finding, area searches, creation and deletion of tiles, enumeration of tiles in an area, and a form of compaction can be performed efficiently on a corner-stitched structure.

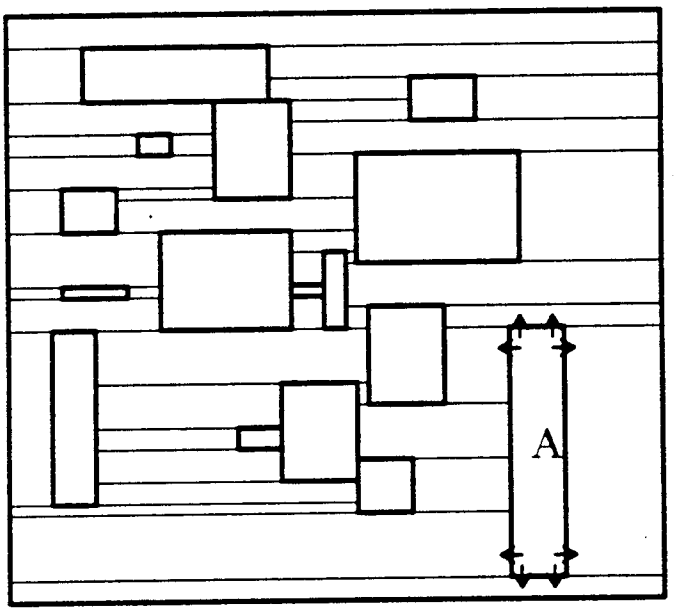

Figure 1: Corner Stitched Tiles, Showing Pointers from Tile A

The space between objects, as shown in Figure 1, and as described in the corner stitching report, is organized into maximally horizontal strips. There is nothing particularly magical about horizontal strips: corner stitching will work with any dissection of the plane into non-overlapping rectangles. However this dissection has the advantages that it is unique and easy to calculate, and it is possible to organize the algorithms that operate on the data structure so they can exploit the maximally horizontal strips for greater efficiency. Although there are some configurations of

objects for which maximally horizontal strips do not give a dissection into the minimal number of tiles, for a given number of objects (n), the worst case number of tiles created by this dissection is the same as the worst case of the optimal dissection ($3n+1$) [Oust84, Lips79].

To date, there have been two research projects based on a corner stitched data representation. One is my own work on an experimental prototype layout editor called WICRD (which stands for Wire Incremental Compaction, Routing, and Displacement), which is described in the latter part of this report. The other, more ambitious effort, is the Magic layout system developed by a group under John Ousterhout [Oust83]. Both WICRD and Magic emphasize the ability to make incremental updates to a layout: moving, routing, and compacting blocks connected by wires. Unlike WICRD, which is basically a toy research prototype, Magic is a working layout editor which has already been used to do real design work.

In designing any IC layout editor based on corner stitching, a key question to be resolved is how the system will represent wires internally. The choice of wire representation may affect both the system's functionality and its performance. WICRD and Magic use two very different wire representations, but these by no means exhaust the possibilities. The first part of this report is devoted to a comparison of six different wire representations based on corner stitching, including the two used in WICRD and Magic. These six representations were chosen to be representative of the many possible wire representations based on corner stitching. Five of these can be implemented using corner stitches; the sixth (called Overlapped Segments) cannot use the corner stitched approach directly, but is closely related to two of the other representations.

These representations are described in section 2 of this report, and compared and criticized in section 3. The issues discussed include the representations' descriptive power, ability to capture the semantics of wire movement, and efficiency implications. Section 4 describes the implementation of WICRD, including the movement, compaction, and routing algorithms used, and then discusses how the WICRD model might be extended to handle more general cases. Section 5 discusses the conclusions drawn from the comparison of the different wire representations and from the WICRD experimental implementation.

2. Description of Wire Representations

The general model of layout used here describes the higher levels of the layout process. At these levels, circuit elements have already been designed and incorporated into lower level blocks that are included as subcells in the current level of design. These subcells, or circuits, are placed in the layout space and are connected by wires that attach to terminals on the circuits. The primary objects on a design plane are circuits with terminals, wires, and the space between them. These objects are to be represented with tiles, although terminals need not be part of the tile structure. Tiles that represent part of circuits are called *circuit tiles,* or *circuit boxes.* Tiles that represent part of wires are called *wire tiles,* or *wire boxes.* Tiles that represent part of the space between objects are called *space tiles,* or *space boxes.*

The same example is used to illustrate each of six different wire representations. This example consists of a single circuit box with one terminal, and a wire with three segments connected to that terminal.

2.1. Space Bins

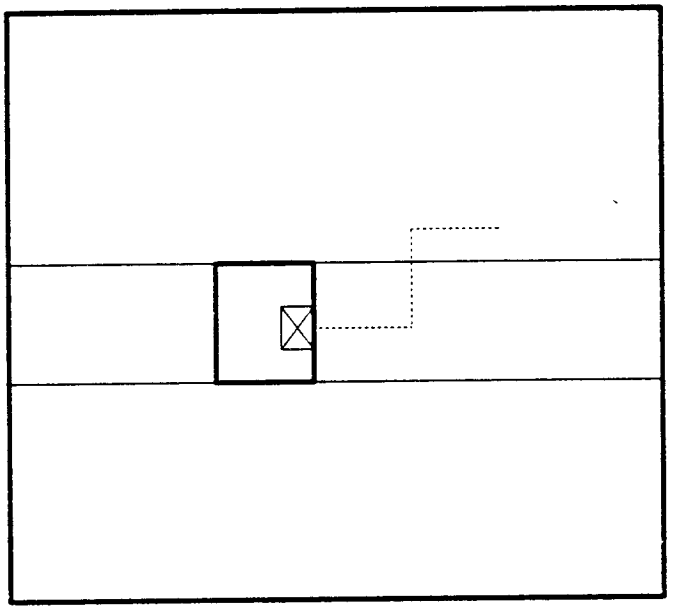

Figure 2: Space Bins with Wires. The heavy tile in the center is a circuit box with one terminal (marked with an X). The light solid lines show the partition of the remaining space into space tiles. The dotted line emerging from the terminal is the centerline of a wire with two horizontal pieces and one vertical piece that is broken at the space tile boundary to form two segments.

One possible wire representation using tiles is to let space tiles act as "bins" to divide up the area between circuit tiles. In this representation, the wires are not part of the tile structure, but are represented by an auxiliary data structure pointed at by the space tiles the wire passes through. In addition, any attached circuit boxes need to point to the wire so that electrical connectivity can be traced. Each space tile contains two sorted linked lists of wire segments passing through it: a horizontal list and a vertical list. By keeping the wire segments in each direction in a sorted doubly-linked list, we can easily find the first few wire segments in a given tile from any direction. The space tiles act as a "rough sorting" mechanism that allows us to eliminate all but a few wire segments in searching for the next adjacent wire segment.

Figure 2 illustrates the Space Bins representation. In this example, there are two space tiles containing wire segments, each with one horizontal and one vertical segment linked in the corresponding lists for that space tile. When a wire segment crosses a space tile boundary, it is

broken into two segments at that point (this will always be a vertical segment, since all space tiles are maximally horizontal). A (horizontal) segment lying on a space tile boundary can be unambiguously assigned to one of the two tiles by some standard convention.

2.2. Tagged Edges

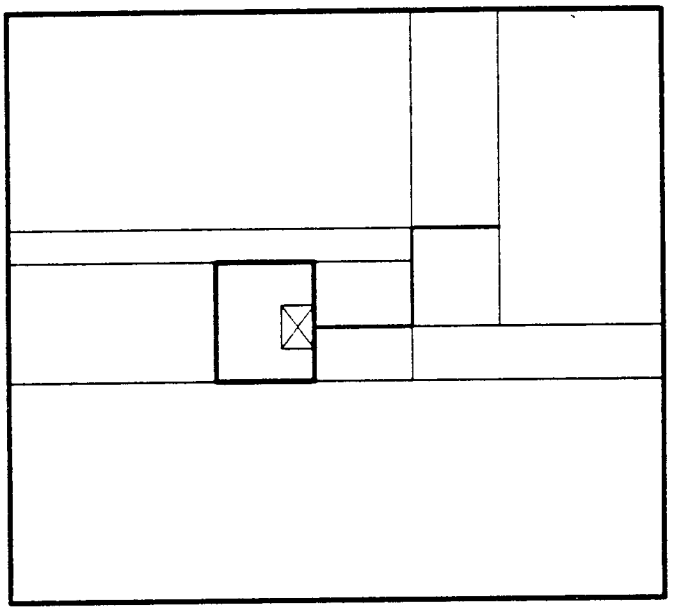

Figure 3: Space Boxes with Tagged Edges. The tagged edges are shown slightly heavier than the untagged edges.

The second representation uses space tiles to define wires implicitly by breaking the space tiles at the centerline of a wire, and tagging the edges of the adjacent space tiles to indicate the presence of the wire and its attributes. Figure 3 illustrates this representation. We still attempt to make the space tiles maximally horizontal, but we are subject to the constraint that each space tile edge must be either totally occupied by a wire segment or totally vacant. This constraint could be partially relaxed: if the space tiles on one side of a wire were not tagged, they would not need to be so constrained. For example, wire segments could tag only the space tiles above or to the right of them.

2.3. Joint Patches

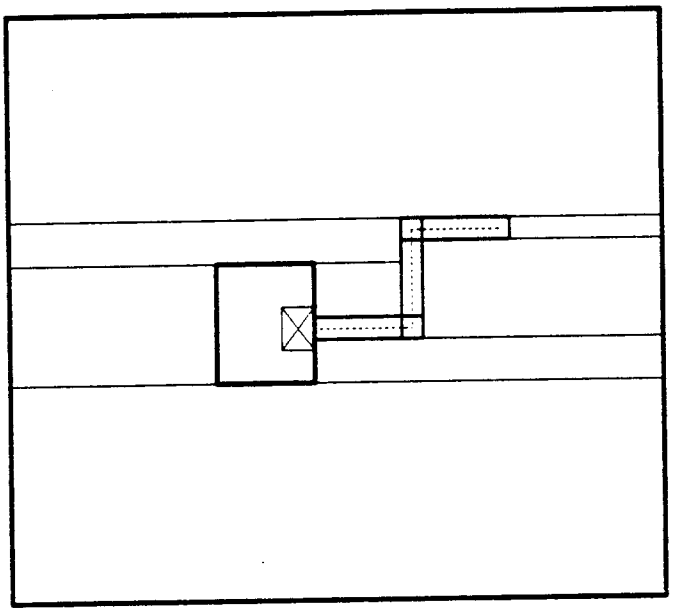

Figure 4: Wires with Joint Patches. Three straight segment tiles alternate with two square corner tiles.

We now consider some representations for wires that represent the space occupied by the wire. The first of these is called Joint Patches. We split the wire into two kinds of pieces: corners and straight segments. Each of these is represented by a separate tile. This is illustrated in Figure 4. The wire consists of three straight segment tiles, and two corner tiles to connect them. These tiles are connected by special pointers to show the electrical connectivity, in addition to the regular tile pointers.

2.4. Overlapped Segments

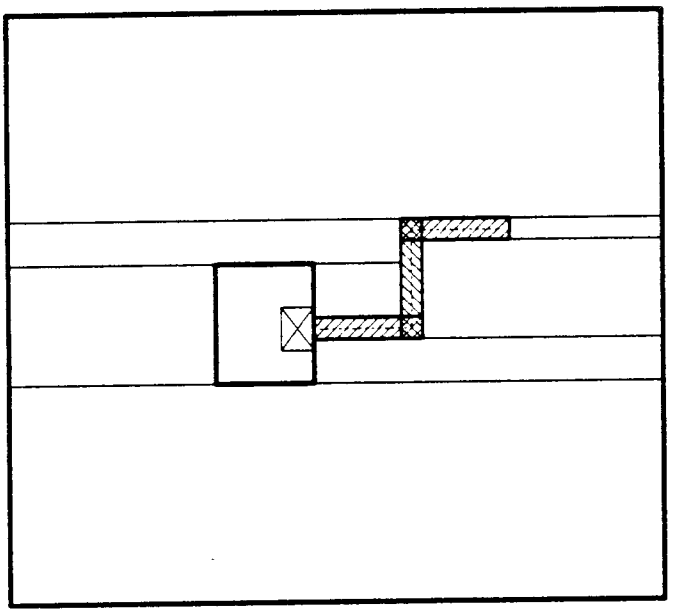

Figure 5: Overlapped Segments. Horizontal segments are shown with rising diagonal hatching left to right; the vertical segment is shown with falling diagonal hatching.

The next representation, Overlapped Segments, deviates from the strict requirement of no overlap among the tiles. It represents each segment of the wire as a rectangle that extends a wire radius beyond the ends of the segment centerline. Consecutive segments thus overlap each other. This representation is illustrated in Figure 5. The three segments are shown cross-hatched to allow each different segment to be identified.

Because of the overlapped tiles, this representation cannot use corner-stitching to connect all the tiles. It will be discussed as if some alternate pointer structure were possible (possibly a modification of corner-stitching), but the reader should be aware that this is an open implementation issue, so this representation may not be realizable in an effective manner.

2.5. Wire Extension

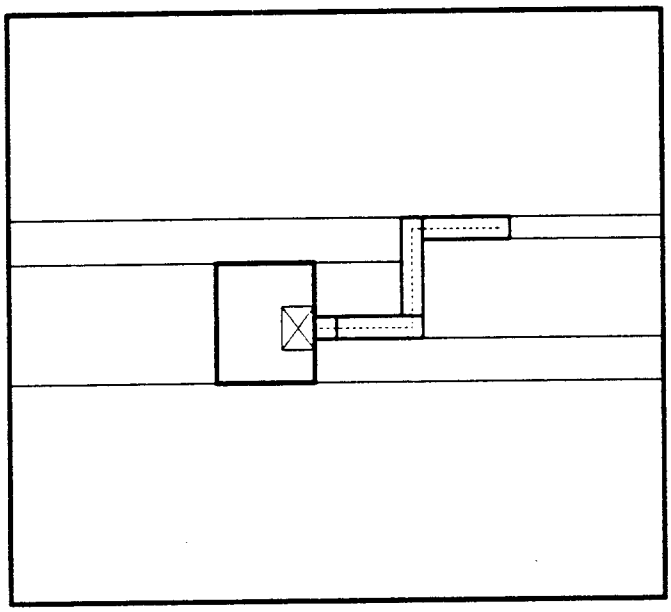

Figure 6: Wire Extension. The wire base is the square tile adjacent to the terminal.

The next representation, Wire Extension, starts each wire with a square tile one wire width on a side, called the wire base. The wire is built up as a chain of tiles, one per segment, starting from the wire base. Each tile in the chain has a width equal to the wire width, and a length equal to the length of the centerline segment; where the length of the first segment is measured from the center of the wire base.

This representation is shown in Figure 6. Notice that each tile in the chain is offset by a wire radius from the start of the corresponding segment. Each tile also has two electrical pointers pointing to the next and previous tiles in the chain to show electrical connectivity.

This representation is used in the WICRD prototype system, which will be presented in Section 4.

2.6. Paint

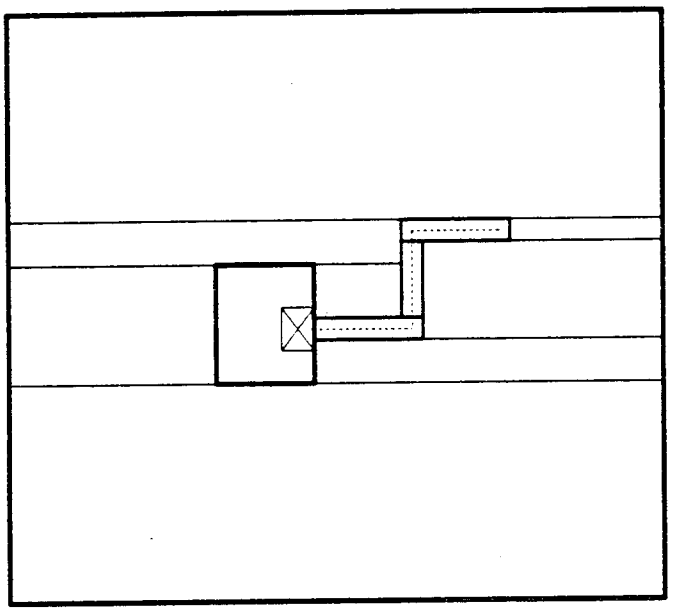

Figure 7: Paint. Wires are represented as maximally horizontal strips of material.

The final representation we will examine here is called Paint, in the tradition of the graphics editor Caesar [Oust81]. Although Caesar was not based on corner stitching, it broke all material into maximally horizontal strips. Paint is the representation used in the Magic system [Oust83]. This representation regards wires as unstructured material: the space occupied by the wire is dissected into maximally horizontal tiles just as the space is dissected into space tiles. Since the representation provides information only about the presence or absence of wire material, it is up to the design tool to deduce the underlying structure of the wire. The resulting tile structure is shown in Figure 7.

3. Analysis of Wire Representations

In this analysis section, we will compare and criticize the preceding six wire representations. We begin by classifying the representations into different categories, followed by a discussion of physical issues (the geometry of wires), semantic issues (how wires respond to manipulation), and efficiency issues (space and time complexity) for the different representations.

3.1. Classifying Representations

The preceding representations for wires can be divided into three broad classes. First, there are the *skeletal* representations. Space Bins and Tagged Edges are in this class. These representations directly express the connectivity of a wire: a wire is a connected chain of line segments connecting two or more points. The physical geometry implementing the wire must be deduced by fleshing out this skeleton to form boxes. Second, there are the *purely physical* representations. Paint is the primary representative of this class. These representations directly express the physical geometry occupied by a wire. Connectivity must be deduced by examining adjacent or overlapping tiles. Finally, there are the *directed box* representations. This class includes Joint Patches, Overlapped Segments, and Wire Extension. These representations can directly express both the physical geometry and the connectivity of a wire. Because they include discrete tiles that can be identified with consecutive segments of a wire, connectivity information can be captured by special pointers that trace consecutive tiles of a wire.

The distinction between the purely physical and the directed box classes is a matter of degree. Directed box representations always maintain a direct mapping between the tiles that represent a wire and the segments of its skeleton; purely physical representations need not do so. It is often possible to establish such a mapping for a purely physical representation such as Paint: Figure 7 was such an example. But Figure 8 shows a case where no such mapping is possible. The four tiles of the wire shown cannot be cleanly identified with the five segments of the wire skeleton.

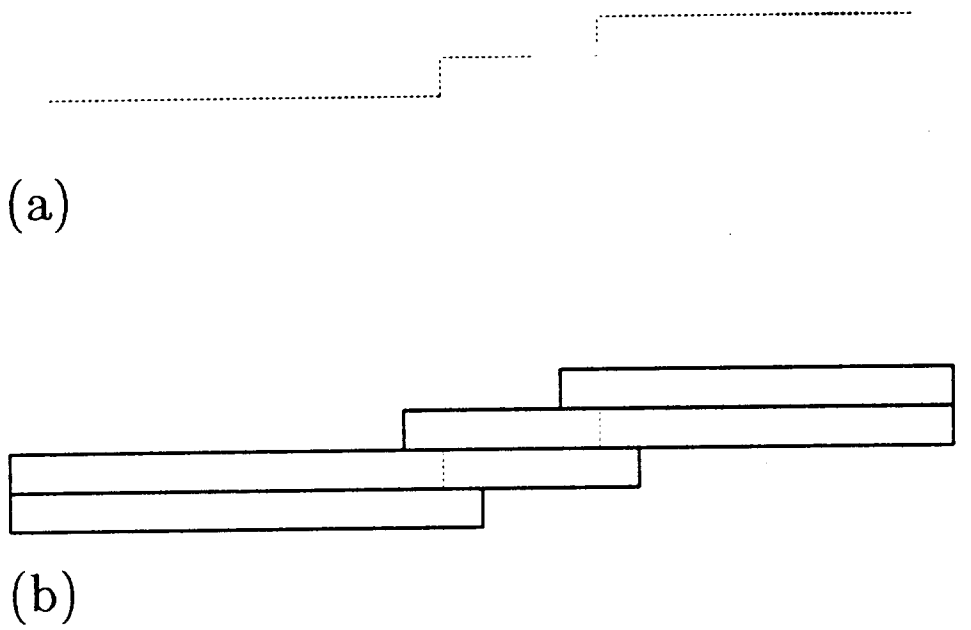

Figure 8: (a): Skeleton of a Wire. (b): The Corresponding Paint Representation.

Because directed box representations can represent connectivity information with pointers, another technique becomes possible. The tiles of a wire can include space beyond the edges of the wire's physical geometry. For example, wires can be expanded by half the minimum separation between objects on their plane. This means that parallel wires at minimum spacing would abut, eliminating the need for a space box between them.

3.2. Representational Power -- Physical Issues

The basic physical issue concerning any wire representation is how well it can express whatever physical geometry the user may want to use to electrically connect two or more terminals. For manhattan geometries, Paint has the greatest representational flexibility. Any desired manhattan mask geometry can be dissected into maximally wide horizontal strips to yield a Paint tile structure. Thus the Paint representation will be used as the basis for comparison with all other representations.

Some of the more structured wire representations encounter problems with some special cases. First, consider two-point connections on a single plane made by a wire of constant width.

There are at least two possible problems here, both involving short wire segments: *short wires,* and *short jogs* (See Figure 9). A short wire is a straight connection between two terminals less than one wire width apart. The only representation likely to have a problem with this is Wire Extension, because of the requirement that each wire start with a wire base one wire width on a side. For this representation, wires less than a wire width long would have to be treated as special cases.

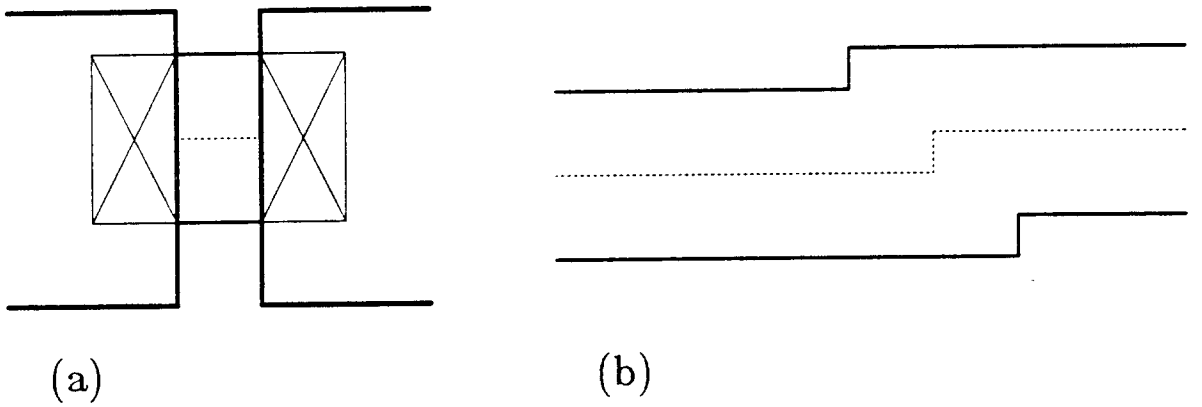

Figure 9: Short Wire Segments. (a): Short Wire. (b): Short Jog.

One possible variant of Wire Extension that would avoid this problem is a representation where the wire base is merged with the first segment of the wire. If this segment is allowed to become less than a wire width in length, it can be used to represent short wires. This variant was a candidate for the wire representation in WICRD, but was rejected because of the complexity it would have added to the program. Virtually every piece of code that had anything to do with wires would have to treat the first segment of a wire as a special case. The complexity of such treatment was judged to be greater than the complexity of having a special tile for the wire base.

A short jog is a segment in the middle of a wire less than a wire width in length, as measured along the centerline. In Figure 8, both vertical segments were short jogs. Short jogs are a problem for Joint Patches, and a potential problem for Overlapped Segments. In the Joint Patches representation, a short jog would require the corner tiles at the ends of the jog segment to overlap, which is not allowed. Attempting to fix this problem by introducing a special "jog"

tile to replace the two corner tiles leads to further problems if a short vertical jog is immediately followed by a short horizontal jog. Attempting to extend this approach beyond one or two successive jogs appears to be futile. It seems likely that any reasonable implementation of Joint Patches will have to prohibit or limit the existence of short jogs. In the Overlapped Segments representation, short jogs cause more than two segments to overlap at a corner. No method for linking overlapping tiles has been proposed, but this problem could make it even more difficult to link them.

A related issue concerns wires connecting to terminals on the corners of boxes, as shown in Figure 10. All the directed box representations have trouble representing a wire that connects to the terminal on the corner, rather than squarely on either side. In essence, a corner of the wire overlaps a corner of the terminal. Overlapped Segments might be able to handle this through the overlap mechanism, but it is an additional complication. Joint Patches and Wire Extension cannot represent this situation without special case treatment, either by extending the terminal beyond the edge of the box so the wire can connect with the side of the terminal, or by adding a special kind of "filler" tile that would serve the same purpose.

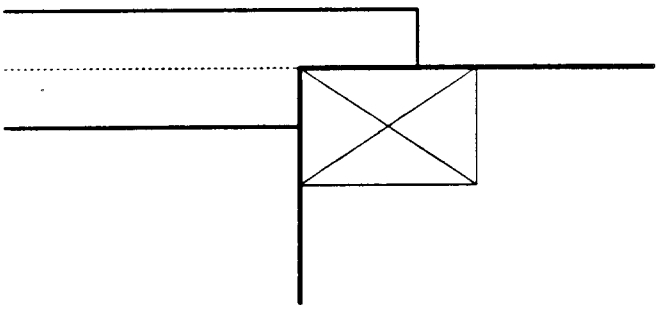

Figure 10: Wire Connecting to the Corner of a Terminal. The centerline of the wire continues for half a wire width along the upper edge of the terminal.

The next set of problems is created by two-point connections on a single plane that do not maintain a constant width. Potentially, this category could include any sort of odd geometric figure used to connect two points. Such figures may be theoretically interesting to prove that Paint has greater expressive power than the other representations, but have little to do with an intuitive

notion of the concept of wire, and do not play an important role except at the very lowest level of cell design, such as is encountered in RAM cells. A subcategory comprises wires that have a consistent centerline, but which change width at one or more points along their length. This might occur if a wire of more than minimum width had to neck down to pass through a heavily constrained region. Such a wire can be handled by breaking it up into separate wires wherever it changes width.

To handle multi-point connections on a single plane, we need to be able to represent junctions of three or four wires at a point. First, assume that all wires at the junction have the same width. Space Bins can represent junctions by breaking all wires at the junction point, and having each segment point to the segment clockwise (or counterclockwise) from it. Tagged Edges can represent junctions without any special modifications. For directed box representations, we have two choices: we can create a special junction box and have all the wires connect to it, or we can form a junction by allowing wires to connect directly to the side of an existing wire. Using a junction box (one wire width on a side), there is a problem representing the case where a wire jogs immediately after leaving the junction. This is similar to the problem of connecting to a terminal by a corner, discussed above. But a similar problem can arise if we construct junctions by joining wires to the sides of existing wires. This problem is illustrated in Figure 11, the pinwheel junction. No matter which two wires we choose out of the four wires meeting at this junction to form the initial wire, at least one of the other two wires will have to connect to this wire on a corner.

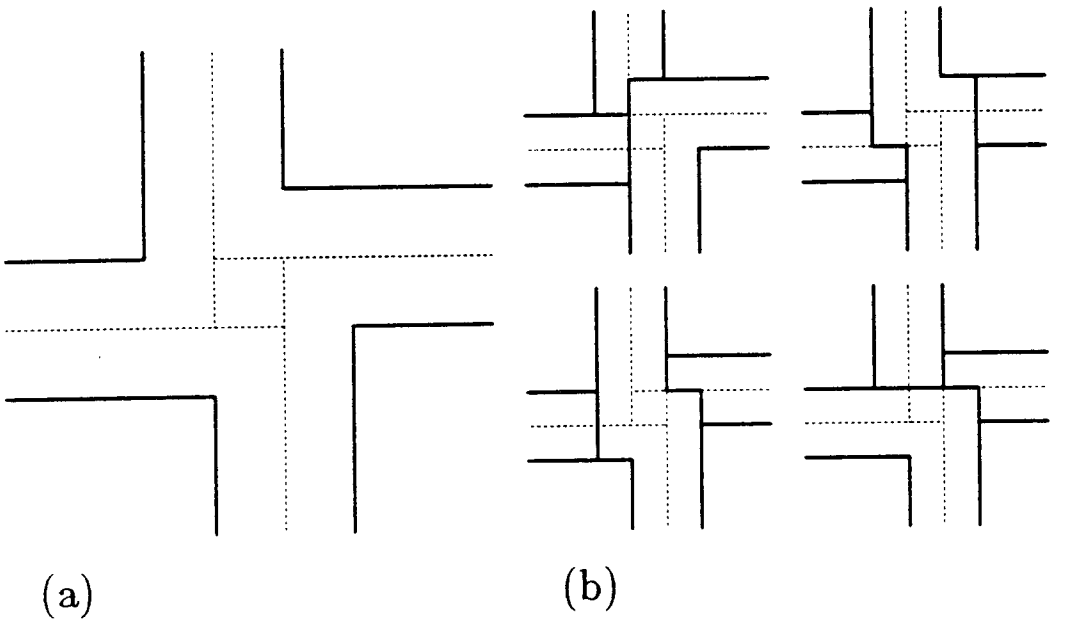

Figure 11: (a): The Pinwheel Junction. This figure cannot be decomposed into a wire with two branches without forcing at least one branch to meet this base wire on a corner. (b): Possible decompositions that include the bottom segment as part of the base wire. The remaining decompositions, which form the base wire from two of the other segments, may be obtained from these by rotation.

A final problem area concerns connections on multiple planes. The major difference between this and the single plane case is the existence of contacts. Like circuits, contacts exist on two or more planes, and must move as a unit on all planes at the same time. Because of this similarity, the easiest way to implement contacts is to encapsulate them as a small circuit. This is mostly independent of the choice of wire representation, but again, directed box representations will be unable to represent a wire connecting to a contact on a corner.

In summary, the most powerful representation for expressing physical geometry is the purely physical representation Paint. Next are the two skeletal representations, Space Bins and Tagged Edges, which can express all the important cases. The weakest representations are the directed box representations, Joint Patches, Overlapped Segments, and Wire Extension. These are unable to describe the special cases of wires connecting to the corners of objects, which restricts the use of junctions and contacts. In addition, Wire Extension must treat short wires as special cases to express them, and Joint Patches has only a limited ability to express short jogs. Overlapped Segments might be able to escape the limitations of this class if a means for overlapping arbitrary

numbers of tiles could be found, but this has not yet been developed into a usable technique. Wire Extension is marginally more powerful than Joint Patches, because it could treat short wires as special cases, and has no difficulty in expressing short jogs.

3.3. Representational Power -- Semantic Issues

If the only requirement that a wire representation needed to satisfy was the ability to represent a static arrangement of mask geometry, then Paint would be the preferred representation, as shown in the previous section. But this is not the case. There is the issue of electrical connectivity. Furthermore, users want to be able to manipulate wires, not just add and delete them. In the process of designing and routing a chip, it is often necessary to move objects around: to make space for additional wires between two existing blocks, to close up gaps where routing space is no longer needed, to fit a slightly larger or smaller version of a redesigned cell into a hole that is no longer the right size. If the only recourse at such times is to rip up all the partially completed routing and start over, much effort will be lost.

There are at least two possible models of how a VLSI wire should respond to the forces of objects moving around it to remain connected with its endpoints. The first model, which I will call the "elastic pole" model, views a VLSI wire segment as an object with fixed width but variable length, that can stretch or shrink to accommodate motions of its endpoints parallel to its length, but which will act as a rigid object when affected by motions perpendicular to its length. A wire consists of one or more of these segments connected together. The other model, which I will call the "wet noodle" model, regards a wire as something that dangles limply from a terminal, like a wet noodle or a piece of string. In this model, the wire will bend, twist, and stretch as much as necessary to avoid exerting any force on anything, only transmitting forces when the pile-up of wires becomes so great that there is no more room to bend. This is the model used in Magic's "plowing" operation [Scot83]. The following analysis will focus on the elastic pole model because it is simpler to work with, uses fewer wire segments, and potentially offers the user more control over the final result.

In the elastic pole model, one issue is whether "jog points" can be represented in a wire. A jog point is a point where a straight section of wire is allowed to shear to form a jog if one side of the wire is moved. One can view a jog point as a zero length wire segment perpendicular to the two segments it is connected to. If one of these two segments is moved, the jog point can stretch as a normal segment would to maintain the connectivity between its endpoints. This is illustrated in Figure 12, which shows the process of reversing a jog. If jog points are represented, then position (c) will result regardless of whether the move is done in a single step, or in two steps with (b) as the intermediate position. If jog points are not represented, then position (c) will not result if (b) is used as an intermediate position. Instead, the two vertical segments will be merged into a single segment, which will then move to the right as a unit.

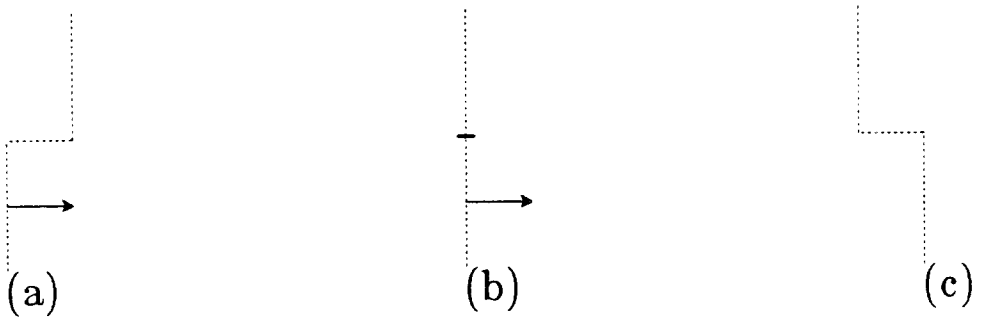

Figure 12: Reversing a Jog. (a): Initial Position. (b): Intermediate Position. (c): Desired Final Position. This figure illustrates three stages in reversing the direction of a jog segment. The bottom vertical segment is being moved to the right. As it moves, the horizontal jog segment shrinks to zero length, then stretches on the other side of the upper vertical segment. At zero length, it forms a jog point (marked by the horizontal line in (b)).

How easily can different representations represent jog points? For Space Bins, a zero length segment is just a special case of an ordinary wire segment. Tagged Edges requires additional tagging information to do so. The space boxes on either side of the wire are split at the jog point, but this is not enough to indicate a jog point, since this situation could also occur naturally because of edges of objects on either side. Therefore, these boxes must be tagged to indicate a jog point at that corner. Joint Patches can represent jogs only with limitations. It could represent a jog point with a corner tile that has segments attached to opposite sides rather then adjacent ones. The problems this representation has with short jogs would cause restrictions on where a

jog point can occur. Overlapped Segments can represent jogs provided it can overlap multiple segments. Wire Extension is well suited to representing jogs; in fact, this representation was inspired by the problem of reversing a jog (see Figure 13). It can either represent the jog point implicitly, by having two consecutive segments in the same direction, or explicitly, by using electrical pointers to link a zero length segment between two such segments. Paint, by definition, only represents the physical geometry of a wire, and a wire with a jog point has the same mask geometry as a wire without one. This is a good reason why a Paint based system, such as Magic, might prefer to use the wet noodle model, where jog points are not significant.

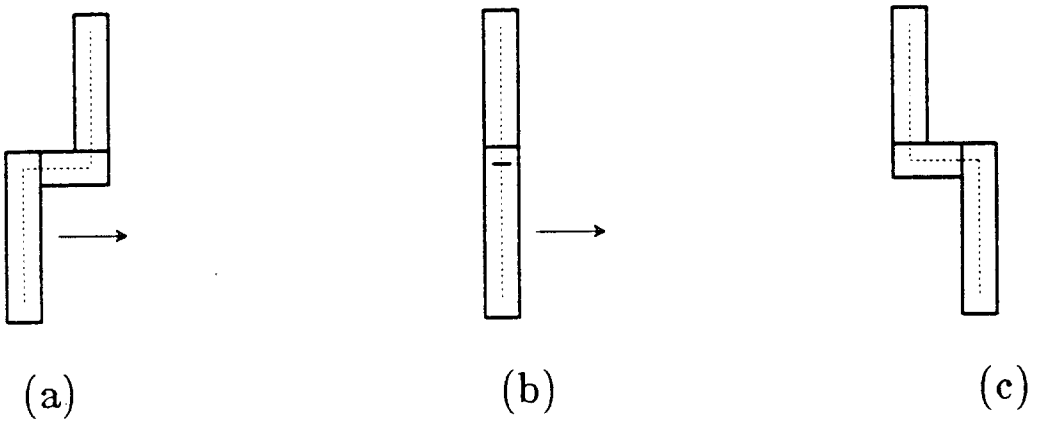

Figure 13: Reversing a Jog in the Wire Extension Representation. (a): Initial Position. (b): Intermediate Position. (c): Final Position. This figure shows how Wire Extension handles the problem of Figure 12.

Another semantic issue involves the manipulation of multi-point nets with multiple widths. Two examples are shown in Figures 14 and 15. In each case, an intermediate segment (x) is generated that must be of the proper width. There are two questions here - one is how to recognize that a decision must be made, and the other is how to decide what width to give the intermediate segment. The latter decision is independent of the choice of wire representation, but the former is not. Skeletal representations have no particular difficulty in recognizing these situations, for the program can examine all segments coming in to a junction and check their widths. Directed box representations may have difficulty in representing these junctions, as noted above, but if these can be overcome by the use of "filler" tiles, it should at least be possible to identify the segments involved in a junction and how wide they are. Purely physical representations have the greatest

difficulties in recognizing these problems. Since these representations do not identify tiles with individual segments, the program is faced with the difficulty of determining the segment structure of the junction from the physical geometry. This includes the problem of determining which dimension of a segment represents the width -- a short fat segment in the vertical direction looks the same as a long skinny segment in the horizontal direction.

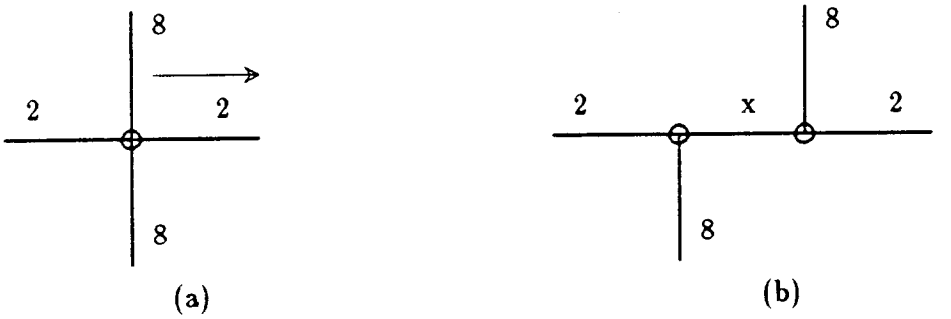

Figure 14: Numbers indicate segment widths. (a): Initial Position. (b): Final Position. The width of the segment labeled x should be 8.

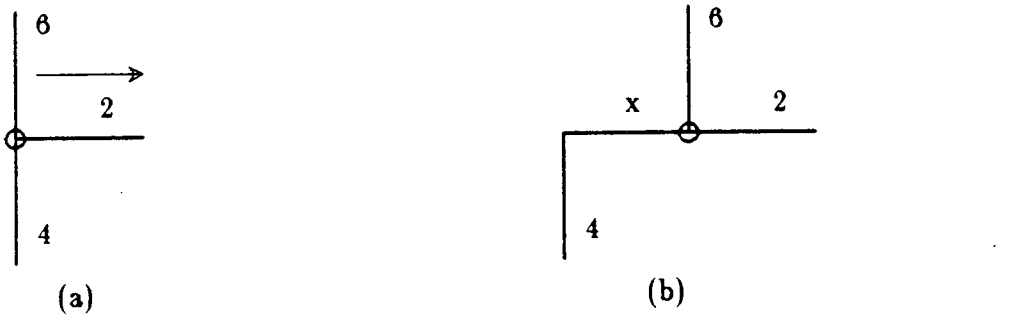

Figure 15: Numbers indicate segment widths. (a): Initial Position. (b): Final Position. The width of the segment labeled x should be 4.

One approach to this problem might be to attach width attributes to the paint tiles to indicate the desired width. This solution, though, is likely to cause more problems than it would solve. It would introduce to Paint some of the problems faced by directed box representations, such as the short wire problem. And because Paint tiles bear no particular relation to the segment structure of a wire, it would be difficult to properly propagate the correct width attributes in a tree structure of varying wire widths, such as are used to distribute Vdd and Ground.

Magic's "plowing" operation [Scot83], which is based on the purely physical representation Paint, deals with these semantic issues either by finessing them or ignoring them. It uses the wet noodle model of wires, in which jog points are not significant. It operates on edges of material rather than wire segments so it does not need to deduce the segment structure. When new segments are created by plowing, they are created at minimum width. Even worse, when wide wires are subjected to a plowing step, they get compressed to minimum width. This means Magic, as of the date of this writing, will not automatically generate or maintain the correct widths in Figures 14 and 15, but requires the user to go in and fix these situations up afterwards by widening too narrow segments back to the proper width.

In general, then, the purely physical representations have the most difficulty with semantic issues because they do not represent information other than the physical geometry of the wires. Skeletal and directed box representations both seem to be adequate for semantic issues, although the directed box representations may have difficulty with some of the physical issues associated with junction formation.

3.4. Efficiency Issues

One efficiency issue is search distance -- how far the program must search from a wire to detect possible interactions with other objects. For directed box and purely physical representations, the search distance is bounded by the size of the largest design rule, since the representations capture all the area occupied by any feature. However, skeletal representations have a potentially unbounded search distance, since wires can have any width (see Figure 16). The program knows the width of the wire it is searching from, but not the width of any wire it is searching for until it finds it. The search distance could be bounded by the radius of the widest wire on the plane plus the largest design rule interaction, but this could be unnecessarily large if a plane had a few wide wires (e.g. power and ground) and many narrow ones.

Most of the representations discussed here require storage that grows linearly with the number of objects represented (circuit boxes and wire segments). For the purely physical and

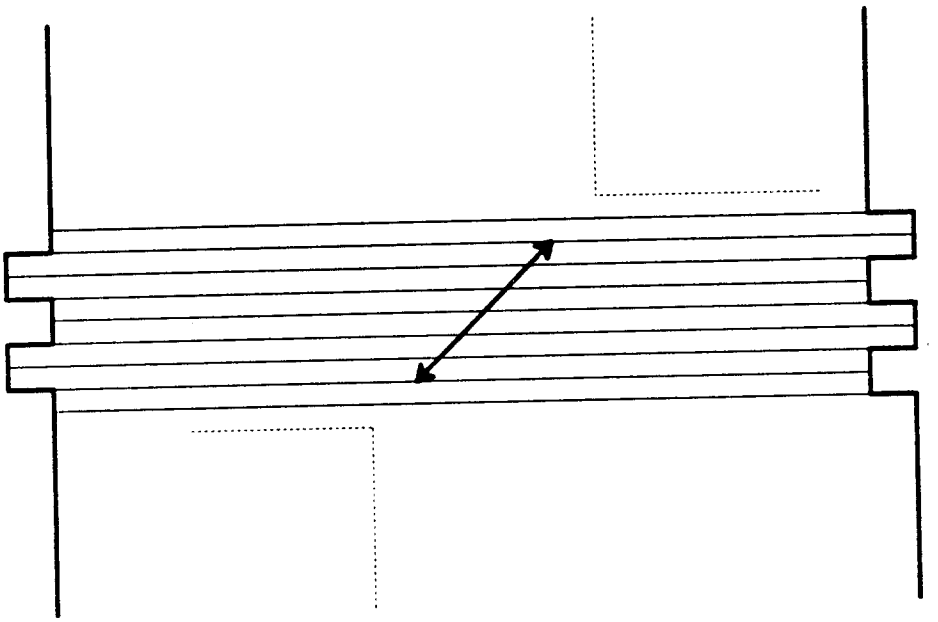

Figure 16: The search distance for possible interactions is potentially unbounded with skeletal representations, unlike purely physical and directed box representations.

directed box representations, this is clear because the number of additional space tiles needed to separate n boxes is at most $3n+1$ [Lips79, Oust84]. Tagged Edges appears to require linear storage also (see below). However, Figure 17 shows that the possible storage requirements for the space bin representation can be as much as $O(n^2)$.

An informal argument to justify the apparent linear storage requirement for Tagged Edges starts by regarding each wire segment as a box of infinitesimal width. The space between circuits and wires can then be broken into space tiles as in the directed box and purely physical representations, requiring at most $3n+1$ space tiles. This partition may result in some tile edges being only partially occupied by a (horizontal) wire segment. This can be fixed by subdividing these tiles with perpendiculars from the ends of such segments. This will add at most 4 tiles for each such segment, so the total number of space tiles will be bounded above by $7n+1$. Finally, some of the resulting tiles may be merged together without violating any restrictions to produce a final result such as Figure 3. This will not increase the total number of tiles.

In [Oust84], a comparison was made between the space per tile required for a corner-stitched

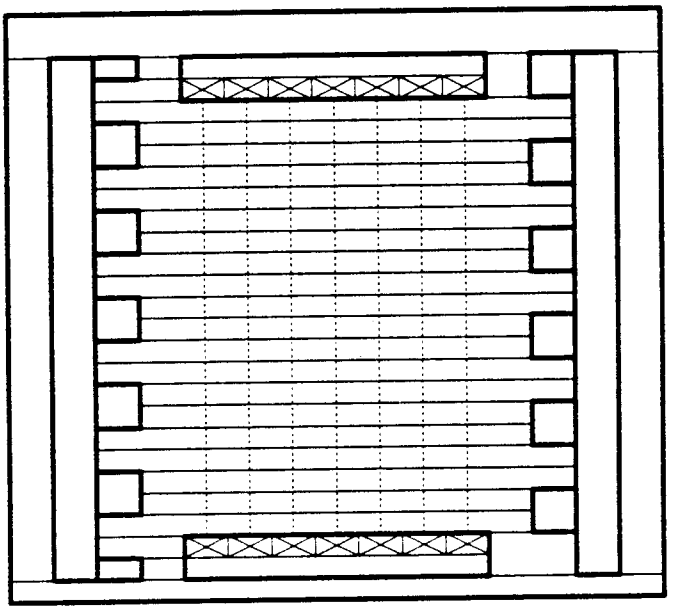

Figure 17: Wire fragmentation in the space bin representation. Because every wire passes through almost every bin, the number of wire segments can be $O(n^2)$, if there are $O(n)$ wires and $O(n)$ bins.

representation (equivalent to the Paint representation discussed above), and the space per tile for the non-corner stitched data representation used in the graphics editor Caesar (which split objects into maximally horizontal strips, but did not tile the space between them). Table I extends this comparison to all the representations discussed above (including an alternate Wire Extension representation in which wires are expanded to include half the minimum spacing between objects, which requires additional electrical pointers to trace the connectivity of a wire), and adds the actual space used in the WICRD prototype system as an indication of how implementation-sensitive these numbers may be. Table II uses the data from Table I to calculate the total space required for the example shown in Figures 2 - 7 in each representation.

Table I: Space Required Per Tile

Representation	Solid Boxes		Space Boxes		Wire Segs.		Terminals[1]	
Caesar	4	Coords		N/A	4	Coords	4	Coords
	1	Link			1	Link	1	Link
Total	5	Words		N/A	5	Words	5	Words
	(20	Bytes)			(20	Bytes)	(20	Bytes)
Paint	2	Coords	2	Coords	2	Coords	4	Coords
	1	Tile Type	1	Tile Type	1	Tile Type	1	Link
	4	Stitches	4	Stitches	4	Stitches		
Total	7	Words	7	Words	7	Words	5	Words
	(28	Bytes)	(28	Bytes)	(28	Bytes)	(20	Bytes)
Space	2	Coords	2	Coords	2	Coords	4	Coords
Bins	1	Tile Type	1	Tile Type	1	Tile Type[2]	1	Link
	4	Stitches	4	Stitches	2	Elect. Ptrs.	1	Elect. Ptr.[3]
			4	Seg. Ptrs.	2	Seg. Ptrs.		
					1	Width		
Total	7	Words	11	Words	9	Words	6	Words
	(28	Bytes)	(44	Bytes)	(36	Bytes)	(24	Bytes)
Tagged	2	Coords	2	Coords		N/A	4	Coords
Edges	1	Tile Type	1	Tile Type			1	Link
	4	Stitches	4	Stitches				
			2	Seg. Width				
			2	Seg. Type				
Total	7	Words	11	Words		N/A	5	Words
	(28	Bytes)	(44	Bytes)			(20	Bytes)
Joint	2	Coords	2	Coords	2	Coords	4	Coords
Patches	1	Tile Type	1	Tile Type	1	Tile Type	1	Link
	4	Stitches	4	Stitches	4	Stitches		
Total	7	Words	7	Words	7	Words	5	Words
	(28	Bytes)	(28	Bytes)	(28	Bytes)	(20	Bytes)
Overlapped	2	Coords	2	Coords	4	Coords	4	Coords
Segments	1	Tile Type	1	Tile Type	1	Tile Type	1	Link
	4	Stitches	4	Stitches	4	Stitches[4]		
Total	7	Words	7	Words	9	Words	5	Words
	(28	Bytes)	(28	Bytes)	(36	Bytes)	(20	Bytes)
Wire	2	Coords	2	Coords	2	Coords	4	Coords
Extension	1	Tile Type	1	Tile Type	1	Tile Type	1	Link
(not	4	Stitches	4	Stitches	4	Stitches		
expanded)								
Total	7	Words	7	Words	7	Words	5	Words
	(28	Bytes)	(28	Bytes)	(28	Bytes)	(20	Bytes)
Wire	2	Coords	2	Coords	2	Coords	4	Coords
Extension	1	Tile Type	1	Tile Type	1	Tile Type	1	Link
(expanded)	4	Stitches	4	Stitches	4	Stitches		
					2	Elect. Ptrs.	1	Elect. Ptr.[3]
Total	7	Words	7	Words	9	Words	6	Words
	(28	Bytes)	(28	Bytes)	(36	Bytes)	(24	Bytes)
WICRD[5]	4	Coords	4	Coords	4	Coords	4	Coords
(actual)	1	Tile Type	1	Tile Type	1	Tile Type	1	Tile Type
	8	Stitches	8	Stitches	8	Stitches	8	Stitches
	2	Links	2	Links	2	Links	2	Links
	2	Elect. Ptrs.	2	Elect. Ptrs.	2	Elect. Ptrs.	2	Elect. Ptrs.
	11	Other	11	Other	11	Other	11	Oth:
Total	28	Words	28	Words	28	Words	28	Words
	(112	Bytes)	(112	Bytes)	(112	Bytes)	(112	Bytes)

Notes to Table I:

(1) Assumes that all terminals are not part of the tile structure, but are found on a separate linked list (this corresponds to the practice in both Magic and Caesar).

(2) Assumes that the type field includes information about whether the segment is horizontal or vertical, and what material it is composed of.

(3) Assumes only one wire per terminal.

(4) This is a guess, as the actual pointer structure for Overlapped Segments is not known.

(5) As can be seen from the table, the actual space required for tiles in WICRD is considerably larger than for the theoretical representations compared in the rest of the table. Implementation decisions that affected this include making all tiles use the same format for simplicity (thus space tiles have unused electrical pointers), representing potentially redundant information explicitly in each tile, rather than deducing it from the rest of the tile structure, and including some additional fields in each tile rather than in a separate dynamic data structure. These tradeoffs make good sense for an experimental system such as WICRD; in a production version, more attention could be paid to minimizing space at the expense of ease of implementation.

Table II: Space Required for Example (bytes)

Representation	Solid Boxes		Space Boxes		Wire Segments		Terminals		Total
	#	Size	#	Size	#	Size	#	Size	Size
Caesar	1	20	0	N/A	3	20	1	20	100
Paint	1	28	8	28	3	28	1	20	356
Space Bins	1	28	4	44	4	36	1	24	372
Tagged Edges	1	28	10	44	0	N/A	1	20	488
Overlapped Segments	1	28	8	28	3	36	1	20	380
Joint Patches	1	28	8	28	5	28	1	20	412
Wire Extension (not expanded)	1	28	8	28	4	28	1	20	384
Wire Extension (expanded)	1	28	8	28	4	36	1	24	420
WICRD (actual)	1	112	8	112	5[1]	112	1	112	1680

Notes to Table II:

(1) Includes the additional box WICRD uses to make the connection between the terminal and the wire base, as explained in Section 4.1.4.

3.5. Conclusions of Analysis

Table III summarizes the relative effectiveness of the three wire representation classes for each of the three areas studied. When two classes are designated as "Good" in a category, I mean that either is likely to be acceptable, and that neither is obviously superior to the other. In the category of Physical Issues, the Purely Physical representations are identified as "Best" because Paint can express any possible Manhattan shape.

Table III: Effectiveness of Wire Representation Classes			
Class	Physical Issues	Semantic Issues	Efficiency Issues
Skeletal	Good	Good	Worst
Directed Box	Worst	Good	Good
Purely Physical	Best	Worst	Good

No one class is superior to the others in all three areas of interest. The best representation class depends partly on the relative importance the CAD system designer places on each area. However, it also depends on how significant a particular class's weaknesses are to the individual application.

Skeletal representations are weakest on efficiency issues. The potentially unbounded distance searched for interacting wires is a problem for all representations in this class. It is unclear how much of a problem this would be in practice. Space Bins has a particular problem with storage efficiency, since it can have an n^2 worst case, where n is the number of objects represented (circuit boxes plus wire segments, before any wire segment splitting required by the representation). Tagged Edges has a linear bound on its storage complexity, though it may have a higher coefficient than other representations. However, the number of boxes required will probably not be much worse than twice the number required for other representations.

An informal argument to justify this estimate, along the lines of the previous informal argu- [illegible] that if there are no wire segments, Tagged Edges has the same $3n+1$ upper bound on the number of space boxes as the other representations. Each wire segment will add at most 4 space boxes to this bound, before merging. However, unlike the other representations, Tagged Edges does not require any space for the wire segment itself. Thus the total number of boxes required will only increase by at most 3 per wire segment. If there are n objects represented, and

they are all wire segments (the worst case for Tagged Edges), other representations will require at most $4n+1$ total boxes (n wire segments and $3n+1$ space boxes), while Tagged Edges will require at most $7n+1$ boxes, less than twice the upper bound for the other representations. Average case figures are harder to estimate. However, Table II shows that for the particular example studied, the total number of boxes is actually less for Tagged Edges than for any of the directed box or purely physical representations, and the total space is comparable to the other representations.

Directed Box representations are worst on physical representation issues. A common theme here is the difficulty of connecting to the corner of an object. This difficulty affects wires connecting to terminals, multi-point nets, and multiple level nets. Although this difficulty can be addressed by the use of "filler" tiles, this is an *ad hoc* solution that disrupts the mapping between segments and tiles, and may result in further complications. Other physical representation difficulties (short wires, short jogs) are specific to particular representations, as previously discussed.

Purely Physical representations are worst on semantic issues because they only represent physical geometry. Thus operations on the segment structure must deduce that structure from the available geometry. When two different structures result in the same geometry, as with jog points, the program cannot distinguish between them. One possible solution, adopted in the Magic system, is to restrict the semantics of wire movement, and allow the user to fix things up afterwards if necessary.

4. WICRD - A Prototype tool for Incremental Wire Manipulation

WICRD, standing for Wire Incremental Compaction, Routing, and Displacement, was developed as a prototype to implement a particular wire representation (Wire Extension) based on space boxes with corner stitches, in order to study the feasibility of the approach and examine related issues. As a prototype, it has a fairly restricted view of the world. Yet within the limitations of this framework, we can draw useful conclusions about the application and use of the data representations and algorithms in more realistic situations.

4.1. The WICRD Model

WICRD manipulates circuit boxes with terminals, wires, and empty space boxes in a single tiled plane. It includes a built-in maze router. In discussing the WICRD model, we will be referring to Figure 18, a representation of what the WICRD user might see in a typical display, and Figure 19, the WICRD system's view of the same scene, which includes some additional objects not visible to the user.

4.1.1. Solid Boxes

The basic object in WICRD is the solid box. It is a rigid rectangular box with no visible internal structure that occupies part of the tiled plane. We may view it as part of a "protection frame" [Kell82b] around a piece of a lower-level macro that has been included into the design at the current level.

4.1.2. Macro Frames

A macro frame in the WICRD world, is a set of one or more solid boxes that have been "stapled" together so that they will move as a single rigid unit. Although the constituent solid boxes may be adjacent (thus allowing us to form rectagons), they need not be. Thus a macro frame will appear as one or more disjoint rectagons. We may view a macro frame as a complete protection frame around the pieces of a lower-level macro.

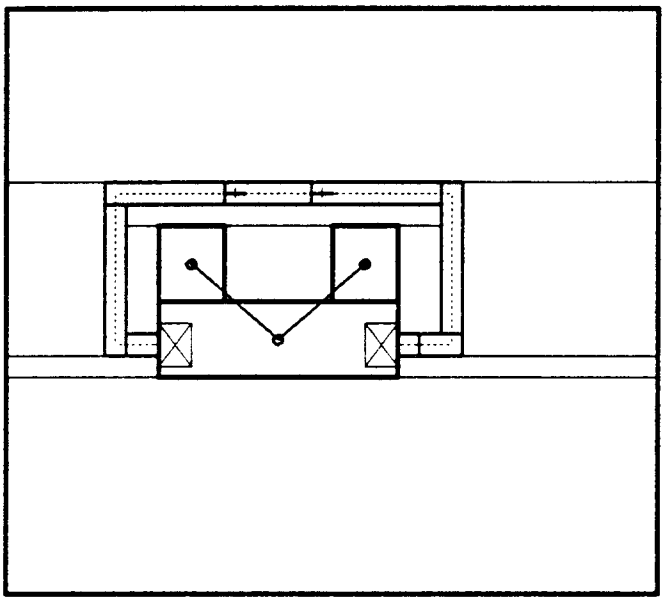

Figure 18: Sample WICRD Display (User View). The three solid boxes shown form a macro frame, shown by the lines joining the circles in the center of each box. There are two terminals on the bottom box (marked with Xs), and a wire connecting the two terminals. The dashed line shows the centerline of the wire. There are two zero length vertical segments in the top part of the wire, shown by the solid lines parallel to the centerline (to show the horizontal extent of the segments), and a dashed line at right angles to the centerline (to show the center of the zero length segment). The light solid lines show the partition of the space into space boxes.

The ability to define macro frames with disjoint pieces has two conceptual functions. One is that this might represent pieces of a macro that are joined together by rigid geometries on some other level, but which allow other geometries on the current level to pass between them. The other, more important function is that this capability provides us with a mechanism for communication between levels. Since it is not necessary to have a physical connection between the pieces of a macro frame, the separate pieces can occur in different tile structures. The movement algorithms will ensure that all pieces of a macro frame always maintain the same relative positions with respect to one another through any movement operation. When we discuss the extension of the WICRD model to multiple levels, a key concept will be that all interactions between objects on different levels are mediated by macro frames.

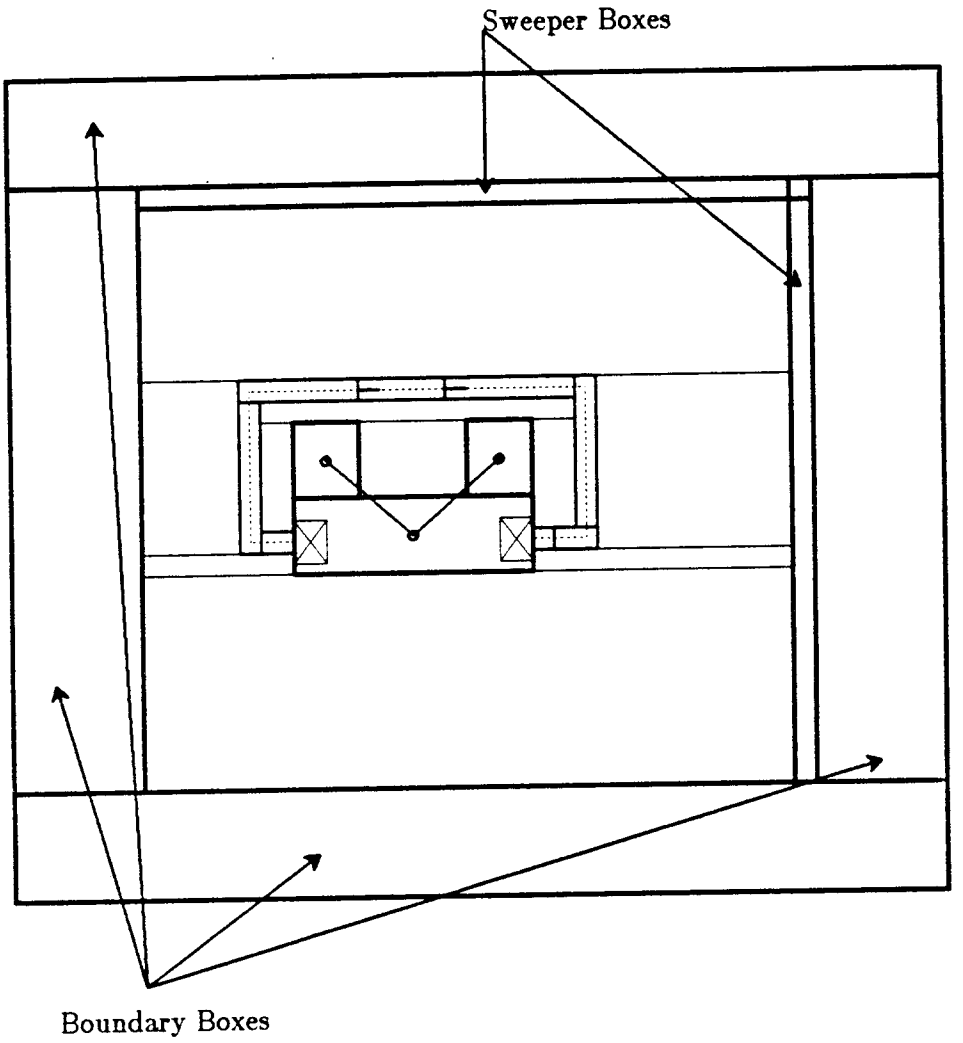

Figure 19: Sample WICRD Display (System View). This figure shows the same scene as Figure 18, but also shows the additional tiles around the periphery of the user area that are not visible to the user. Boundary boxes define the frame of the tile structure, and provide fixed reference points to search for the other tiles in the structure. Sweeper boxes are used in the compaction algorithm. The small square box in the upper right hand corner is identified as a sweeper box to the program. Unlike the other two sweeper boxes, it is never moved, and serves merely to complete the tile structure.

4.1.3. Terminals

In WICRD, all terminals are attached to some solid box. They represent the places at which wires can be connected to the box. Each terminal is a rectangle contained within the solid box it is attached to, with at least one of its edges flush with some edge of the box. Terminals are not allowed to overlap each other. Because they are contained within the solid box they are attached to, they are not part of the tile structure. They move with the box they are attached to, and if it is deleted, all terminals attached to it are deleted also. A wire can be connected to a terminal

along any edge that is flush with a box edge if that edge of the terminal is at least as wide as the wire and the wire is contained within the span of the terminal.

In retrospect, it would probably make more sense to attach terminals to a macro frame rather than to one of the solid boxes in the macro frame. If a macro frame is an arbitrary rectagon, we have to decompose it into rectangular solid boxes to represent it. If terminals are attached to the individual solid boxes, this decomposition is significant, since terminals cannot span solid box boundaries. If terminals are attached to the macro frame, then the exact decomposition of the macro frame into solid boxes is not important, since the terminals can span the solid box boundaries. This would allow the decomposition to be done by the program in whatever fashion was most convenient.

This should certainly be done for a production system, where the macro frames would be predefined as part of the lower-level macros being included. Since WICRD is an experimental prototype, the macro frames can be created and destroyed by the user within WICRD; thus macro frames are not as persistent as they would be in a production environment.

4.1.4. Wires

Wires in WICRD are represented with the Wire Extension representation discussed above. Each wire starts with a wire base, a square box one wire width on a side, and is built up from it by successive extensions equal to the length of each segment. For implementation convenience, WICRD requires that all wire bases be attached to some terminal, although the other end of the wire may dangle. WICRD only represents two point connections that have constant width over their entire length: multi-point nets and variable-width wires are not represented. Also for implementation convenience, WICRD defines all wires to have the same constant width: 16 pixels. This was chosen to be twice the alignment grid spacing of 8 pixels. This allows us to select points along the centerline of the wire, and build sub-minimum width jogs (of 8 pixels).

WICRD's wires consist of alternating x and y direction segments. Any segment or series of segments may have zero length. If two segments in the same direction are joined by a zero-length

segment in the orthogonal direction, that segment defines a jog point where the wire will divide when one of the other segments is pushed sideways. To maintain a consistent and symmetrical representation for zero length segments, they are not part of the tile structure, and are defined to lie along the centerline of the previous segment of the wire. If several zero length segments occur in succession, they will lie crosswise at the end of the previous non-zero length segment of the wire (see Figure 20). These segments are located by the electrical pointers from the rest of the wire.

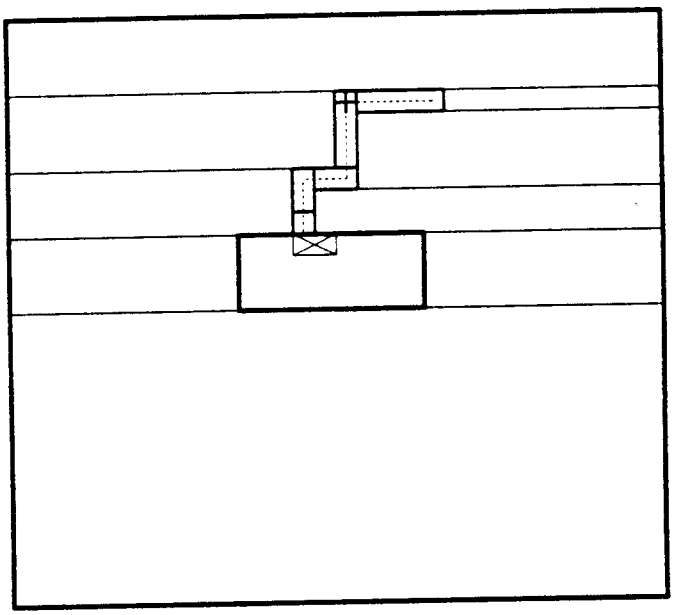

Figure 20: Successive Zero Length Segments. The upper part of the highest vertical segment contains two successive zero length segments (which appear as a solid line cross at the end of the segment). This vertical segment connects to the horizontal zero length segment, which connects to the vertical zero length segment, which connects to the regular horizontal segment extending to the right.

In addition to the boxes representing a wire, a special connection box is used to make the connection between the wire and a terminal. This is a zero-length box that is not part of the tile structure, but is linked to the wire and the terminal by the electrical pointers. It is defined to lie along the edge between the terminal and the wire. The connection boxes and the wire base regularize the mapping between the internal box structure of the wire and the centerline that is displayed for the user. Each wire segment contributes a section of the centerline that has the same length as the segment, but is offset by half a wire width towards the back of the segment. Each connection box contributes a section of the centerline half a wire width long, extending from

the middle of the connection box into the wire. The wire base does not contribute to the centerline. This mapping's regularity enables the display routines to avoid special treatment of the different ways wires can connect to terminals, and its reversibility could allow the program to identify the wire segment corresponding to a piece of the centerline pointed at by the user.

There are, then, five different types of boxes used to represent wires in WICRD: wire base boxes, x and y direction wire boxes, and x and y direction connection boxes. These different types are listed in Table IV below.

Table IV: Types of Wire Boxes

Box Type	In Tile Structure?	Description
WIREBASE	Yes	Base box for wire, one wire width on a side.
XWIRE	Unless zero length	X direction wire segment
YWIRE	Unless zero length	Y direction wire segment
XCONNECT	No	X direction connecter between wire and terminal
YCONNECT	No	Y direction connecter between wire and terminal

4.1.5. Space

The space between boxes is broken up into rectangular tiles. These tiles are maximally horizontal strips between features (see Figure 18).

4.1.6. Limitations

WICRD is implemented with only one level. However, the movement algorithms readily generalize to multiple levels, using the paradigm that all interactions between levels are mediated by macro frames that span the levels. Section 4.2.2 shows how multiple levels may be simulated using the current WICRD structure.

Because WICRD uses the Wire Extension representation, it is subject to all the limitations of this representation discussed in the previous section. It does not include any special treatment for short wires, so wires less than a wire width long are not allowed. This limitation could be removed by adding such special treatment. In addition, all the limitations noted in Section 4.1.4 apply to wires in WICRD.

4.2. WICRD Algorithms

The three types of wire manipulation algorithms in WICRD are movement, compaction, and routing. Of these, movement is easily the most important and complex algorithm. Compaction turns out to be just a special case of movement. The routing algorithm is interesting but simplistic because of WICRD's single level of wiring.

4.2.1. Movement

The movement algorithm will be presented here much as it was developed: beginning with a very simple underlying algorithm, and then progressively adding complications until we get the final form of the algorithm. The complications are presented in roughly the same order as they were developed chronologically.

4.2.1.1. Basic Structure

The movement algorithm in WICRD is a three-pass algorithm. We start with a box (solid or wire) that we want to move, and the direction and distance that we want to move it. In the first pass, we compute how far all other affected objects will have to move as a result of this move, and we may wind up reducing the distance of the move because of a collision with one or more immovable objects. In the second pass, we take all boxes that were scheduled to move in pass 1, and pick them up from the tile structure. In pass 3, we put all the boxes back into the tile structure in their new locations.

Why do we use three passes? By separating the planning phase (pass 1) from the execution (passes 2 and 3), we are able to do the complete computation using the current position of each object. If we hit an immovable object, we may have to undo part of the move, which is much easier if nothing has actually been moved yet. By dividing the execution of the move into pass 2, where we pick everything up, and pass 3, where we put everything down, we ensure that we will never put one box on top of another, as long as we computed the correct move distances in the first pass.

4.2.1.2. Moving Pure Solid Boxes

To examine the movement algorithm, let us look first at the simplest case: moving pure solid boxes. In Figure 21, we see a typical example. We want to move the box labeled "A" to the left as shown. The algorithm is a variation of Dijkstra's shortest path algorithm.* We maintain a priority queue of boxes that have been scheduled but not scanned, ordered by the current distance each is known to move. We scan the boxes in order of decreasing move distance, searching the rectangle this box will move through. If any boxes are found in this rectangle, we compare the distance they are currently scheduled to move with the distance the current box would push them. If the current box would push them further than any distance they are currently scheduled to move,

*Dijkstra's algorithm is described on pages 70-71 of [Lawl76].

we update the distance and schedule (or reschedule) the new box with the new distance. We repeat this process until there are no more boxes to be scanned. More formally, we have:

Algorithm M1: (Simple Box Movement Algorithm)

Data: Boxes numbered $1..n$.

$D[1..n]$, the distance each box is to move.

d, the distance box 1 (the initial box) is to move.

Q, a priority queue of boxes ordered on D, with the largest values of D at the top.

dist(a,b), the distance between boxes a and b in the direction of motion.

$D[1] \leftarrow d$; $D[2..n] \leftarrow 0$;

$Q \leftarrow \{1\}$;

while $(Q \neq \emptyset)$ **do:**

$b \leftarrow$ head(Q); $Q \leftarrow Q - \{b\}$;

Scan a rectangle $D[b]$ out from b in the direction of motion;

for all boxes s in this rectangle, **do:**

Schedule $(s, D[b] - \text{dist}(b,s))$;

enddo

enddo

Procedure Schedule(box, distance);

if $D[box] < distance$ **then**

$D[box] \leftarrow distance$;

if $box \notin Q$ **then**

$Q \leftarrow Q \bigcup \{box\}$;

else update position of box in Q;

endif

endif

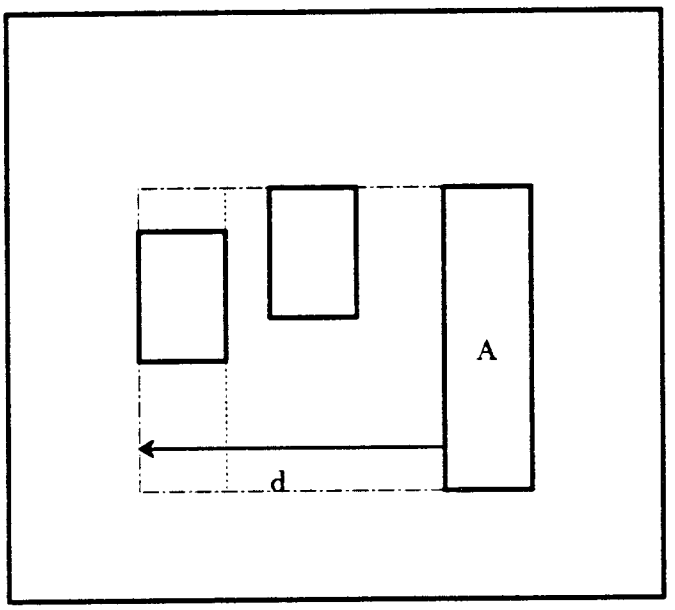

Figure 21: Moving Solid Boxes. The box labeled A is to be moved left by a distance d. The dot-dashed line shows the area swept out by box A; the dotted line shows the final position of the box.

4.2.1.3. Immovable Objects

Now we want to add the concept of *immovable objects* to the algorithm. Immovable objects act as barriers to movement; when a box moves into an immovable object, it stops moving as it hits the object. Movement now means "move as much of this distance as you can without causing some box to move into an immovable object." To do this, the algorithm keeps track not of the absolute distance each box is to move, but the difference between the global move distance and the distance the particular box is to move. When an immovable object is encountered, we take the distance it would normally be forced to move, and subtract this from the global move distance. Thus the amount each box moves is reduced by just enough to keep the immovable object from having to move at all. If a box winds up with a movement delta that is greater than the final global move distance, it is not moved at all. Thus, we can initialize box deltas to $+\infty$ (called NOMOVE) to indicate no movement. The interpretation of the recorded numbers as deltas rather than distances inverts the sense of priorities for the priority queue and the scheduling subroutine: now small numbers represent higher priorities than large ones. Thus the direction of

the test in the Schedule subroutine is reversed: the delta is updated only if the new value is smaller than its previous value. This results in:

Algorithm M2: (Box Movement with Immovable Objects)

Data: Boxes numbered $1..n$.

d, the distance box 1 (the initial box) is to move.

$D[1..n]$, the deltas between d and the amount each box is to move.

Q, a priority queue of boxes ordered on D, with the smallest values of D at the top.

dist(a,b), the distance between boxes a and b in the direction of motion.

$D[1] \leftarrow 0$; $D[2..n] \leftarrow$ NOMOVE($+\infty$);

$Q \leftarrow \{1\}$;

while ($Q \neq \phi$ **and** $D[\text{head}(Q)] \leq d$) **do:**

$b \leftarrow$ head(Q); $Q \leftarrow Q - \{b\}$;

Scan a rectangle $d - D[b]$ out from b in the direction of motion;

for all boxes s in this rectangle, **do:**

if s is an immovable object **then**

$d \leftarrow \min(d, D[b] + \text{dist}(b,s))$;

else Schedule (s,$D[b] + dist(b,s)$)

endif

enddo

enddo

4.2.1.4. Macro Frames

Macro frames are implemented by chaining solid boxes together with a linked list. Each solid box has a pointer to the head of the macro frame it belongs to, and a pointer to the next box on the chain. An isolated box points to itself with the head pointer, and has a null next pointer. Algorithm M2 is changed to use macro frames as its basic units, rather than boxes. A delta is computed for each macro frame. When any box of a macro frame is found in a scan, the

head of the macro frame is placed on the queue. When a macro frame is scanned, each box of the macro frame is scanned individually, using the delta computed for the whole frame. This results in:

Algorithm M3: (Macro Frame Movement with Immovable Objects)

Data: Boxes numbered 1..n.

Macro frames numbered 1..m.

d, the distance frame 1 (the initial macro frame) is to move.

$D[1..m]$, the deltas between d and the amount each frame is to move.

Q, a priority queue of frames ordered on D, with the smallest values of D at the top.

dist(a,b), the distance between boxes a and b in the direction of motion.

mh(a), the head of the macro frame that box a belongs to.

$D[1] \leftarrow 0$; $D[2..m] \leftarrow$ NOMOVE($+\infty$);

$Q \leftarrow \{1\}$;

while ($Q \neq \emptyset$ **and** $D[\text{head}(Q)] \leq d$) **do:**

$a \leftarrow$ head(Q); $Q \leftarrow Q - \{a\}$;

for all boxes b in macro frame a, **do:**

Scan a rectangle $d - D[a]$ out from b in the direction of motion;

for all boxes s in this rectangle, **do:**

if s is an immovable object **then**

$d \leftarrow \min(d, D[a] + \text{dist}(b,s))$;

else Schedule(mh(s), $D[a]$+dist(b,s));

endif

enddo

endoo

enddo

4.2.1.5. Terminals

Since terminals are not part of the tile structure, they are not directly involved in the computation of move distances in the absence of wires. They are connected to their base box by a circular linked list. When the box position is updated, the positions of all terminals attached to it are updated by the same amount.

4.2.1.6. Wires and Electrical Connectivity

To motivate the wire movement algorithm currently used in WICRD, the wire movement algorithm originally used in WICRD will be presented first. Then some problem cases will be examined, and finally, the revised algorithm that solves these problems will be presented.

The basic principle involved in moving wires is that wire boxes perpendicular to the direction of movement act like ordinary boxes, while wire boxes parallel to the direction of movement stretch or shrink as necessary to maintain connectivity with the boxes they are connected to at either end. This is implemented within the framework of Algorithm M3, with boxes being scheduled on a priority queue, and scanned as they are removed from the queue. However, the actions taken when a box is scheduled or scanned depend on the box type. The differences between the default actions given in Algorithm M3 and the actual actions taken are discussed below for each of the different box types. For purposes of discussion, horizontal (x direction) movement will be assumed. The vertical case is symmetrical, reversing the roles of x and y.

Circuit boxes are handled in a normal way, as specified in Algorithm M3. They are scheduled and scanned as part of the macro frames they belong to. However, one more step is added when the box is scanned: the program checks all the terminals attached to the box to see if any one of them is connected to a wire. Any connection boxes that are found are scheduled with the same delta as the box being scanned. This implies that y direction wires will move by the same amount as the terminal (if any) they are attached to; this could be changed to allow such wires to slide along the terminal by increasing the delta appropriately.

Y direction wire segments (YWIREs) are similar to isolated circuit boxes that belong to their

own macro frame. However, in addition to directly pushing and being pushed by other boxes, they also affect and are affected by what happens to the boxes they are electrically connected to at either end (if any). The YWIRE may have been scheduled when one of these boxes was previously scanned. Conversely, when the YWIRE is scanned, both of these boxes are checked to see if they should be scheduled. If such a box is a WIREBASE, or a YCONNECT type box, it is scheduled with the same delta as the YWIRE. If it is a XWIRE box, it is scheduled with a dummy movement delta (see discussion of XWIREs below). These are the only types of boxes that may legally be connected to a YWIRE box.

X direction wires (XWIREs) will generally be reconstructed from the final positions of the electrically connected boxes on either end. The program must make certain that all XWIRE boxes whose endpoints may have moved are checked for stretching or shrinking, and that any cases that are not adequately handled by this general rule are treated correctly. Two such cases are shown below in Figures 22 and 23.

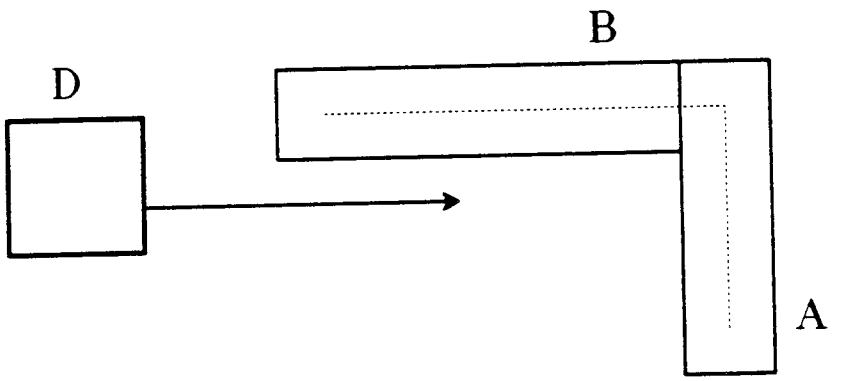

Figure 22: XWIRE box B will have to shrink because of the pressure from box D. The new position of box B cannot be computed by observing only the final positions of the boxes at either end, because box B is the last box in the wire.

To accomplish these goals, the program uses the scheduling mechanism of algorithm M3, but introduces a new "flag value" in the delta field that is used only for XWIREs. This value is called DUMMYMOVE, and is larger than any legitimate delta value, but smaller than the $+\infty$ value (called NOMOVE) used in algorithm M3 to initialize the delta fields of all but the first object in the move. Of course, in practice NOMOVE is itself simply a large number that is larger than any

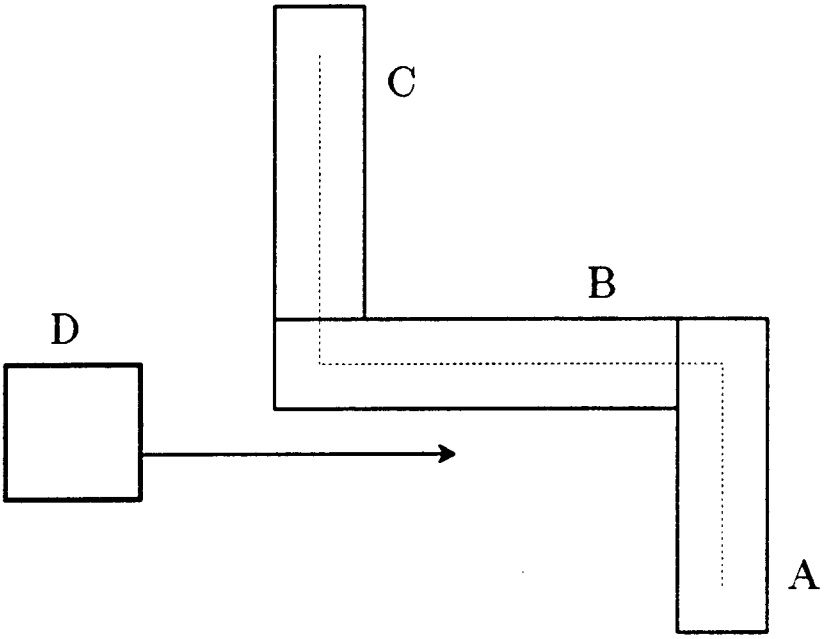

Figure 23: YWIRE box C will have to move because of the pressure on box B from box D. Here the program will have to ensure that box C is correctly scheduled, even though the only direct pressure is on box B.

legitimate delta value. A delta of DUMMYMOVE identifies an XWIRE box whose endpoints may have moved, and thus will need to be recomputed once the algorithm has computed final positions for all the other boxes. When an XWIRE box with a delta of DUMMYMOVE is scanned, it is put onto the queue of boxes to be moved, and the rest of the scanning procedure is omitted. In addition, XWIRE boxes may be scheduled with legitimate deltas when some other box is scanned. This is what happens in the cases illustrated in Figures 22 and 23. In these cases, box B is scheduled with the same delta computation as an ordinary box. When the box is scanned, however, the only boxes it can cause to be scheduled are the adjacent electrically connected boxes (since an XWIRE box can't push on another box in an x-direction move). In the case shown in Figure 22, box B will not cause any other boxes to be scheduled, so its delta will be used only to compute its new endpoint. In the case shown in Figure 23, box C will be scheduled with the same delta as box B when box B is scanned. This type of scheduling applies to the next box along the wire (if any) when the XWIRE runs opposite to the direction of movement, such as in Figure 23. In the reverse case, when the XWIRE runs in the same direction as the motion, the previous box

along the wire (box A in Figure 23) is scheduled. In almost all cases, this latter scheduling is redundant but harmless, since the previous box will be pushed by the same boxes that are pushing the XWIRE. The only case where this wouldn't be true is if the XWIRE box was the original box selected to be moved. In that case, it will stretch or shrink while dragging the trailing endpoint through the distance of the move. In Figure 23, if box B was moved to the left, it would drag box A along; if it was moved to the right, it would drag box C along.

A wire base box (WIREBASE) acts like an ordinary box, except that it always schedules the adjacent electrically connected boxes with the same delta as itself when it is scanned, and can be scheduled by them in the same manner.

Connection boxes (XCONNECT and YCONNECT) can only be scheduled by their electrically adjacent neighbors, since they are not part of the regular tile structure. Likewise, it is only these neighbors that are affected when the connection boxes are scanned. One of these neighbors will always be a terminal. This is traced back to the circuit box it is attached to, and the macro frame this circuit box belongs to is scheduled with the same delta as the connection box. The other neighbor will be either a WIREBASE, a YWIRE, or an XWIRE box. A WIREBASE or YWIRE will be scheduled with the same delta as the connection box; an XWIRE will be scheduled with a DUMMYMOVE delta. Obviously, it will be redundant to schedule one of these two neighbors, since it must have been the one that scheduled the connection box in the first place. But since we don't know which one is redundant, it is safe to schedule both.

Zero length wires (either XWIREs or YWIREs) are something of an exception to the above discussion, since they are not part of the regular tile structure. They will be scheduled by the electrically connected boxes as shown above, but additional work is necessary to make sure they don't get left behind when something pushes on the rest of the wire they belong to. The general approach used by the original WICRD algorithm discussed here was to have the code that searched for boxes in the scanned rectangles also check for zero length wire segments embedded in any wire boxes that were found, and include them in the list of found boxes as long as they fell within the search region. Some of the problems with this approach, and with other aspects of the

above movement algorithm will be discussed in the next section.

4.2.1.7. Difficult Cases

Although the above rules work well for many cases, there are some examples where they fail to work effectively. One example is shown in Figure 24. When box D moves to the left, it pushes on YWIRE box A, and both A and D push on XWIRE box B. Since YWIRE box C is not affected by the move, the final position will result in XWIRE B being placed on top of box D, as shown in Figure 25. Obviously, we need box C to move to the left also. The problem with this situation is that it is very similar to the related situation shown in Figure 26, where box D now clears the bottom of box B. Here it is correct to leave box C where it is, resulting in the final position shown in Figure 27. This is an example of reversing a jog, as shown back in Figure 13.

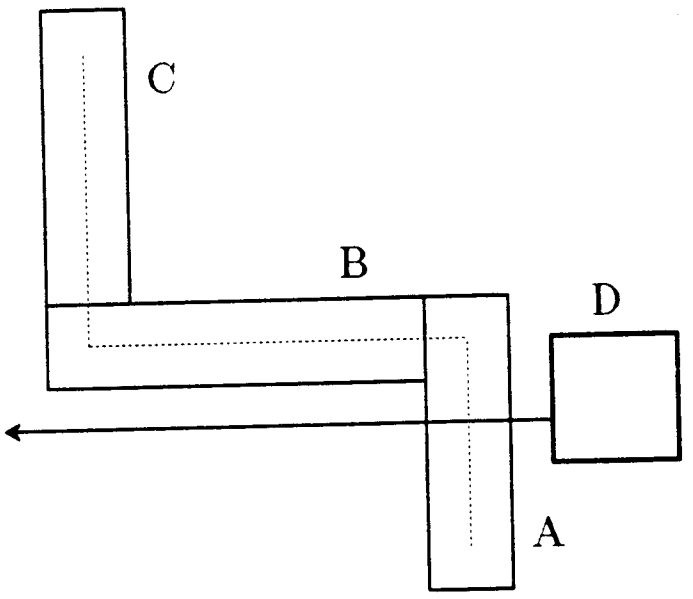

Figure 24: A difficult example for the original WICRD algorithm. Box C is not moved, resulting in the situation shown in Figure 25.

The difficulty is that both of these situations look the same locally, when the algorithm is to consider what to do about box C. Since box C is not within the rectangle swept out by box D, the only place in the algorithm where box C might be considered is when processing box B (since

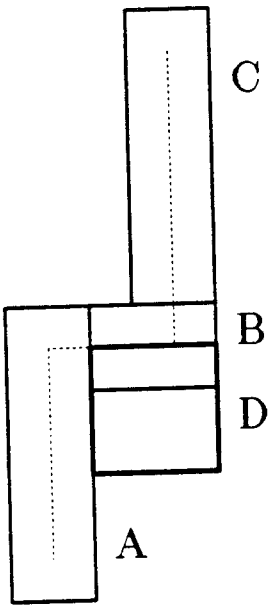

Figure 25: The result of the motion shown in Figure 24. Since box C has not moved, box B collides with the final position of box D when B is generated from the final positions of boxes A and C.

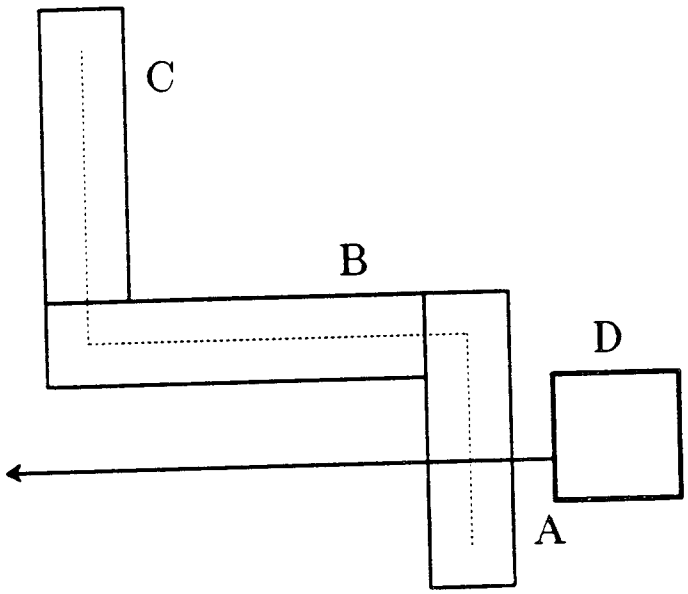

Figure 26: A related situation to the one shown in Figure 24. Here we want box C to stay where it is, resulting in the final position shown in Figure 27.

C is one of the boxes electrically connected to B). But the deltas of boxes A, B, and D are the same in both cases. Thus there is no way to tell the difference between these two cases when we

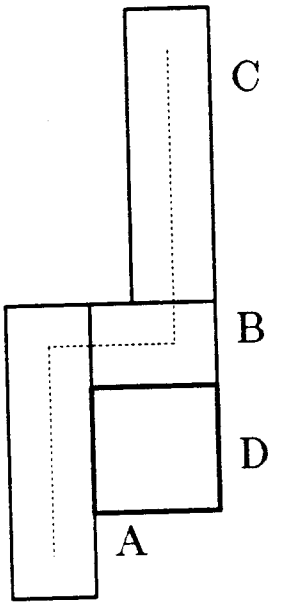

Figure 27: The final result of the move shown in Figure 26.

are processing box B.

Another difficult case for the original WICRD algorithm is shown in Figure 28. This case involves the handling of zero length segments. Since zero-length YWIRE segment C falls within the rectangle swept out by YWIRE segment A, it is pushed ahead of segment A. This results in the situation shown in Figure 29, where XWIRE segment D falls on top of segment A. Once again there is an analogous situation where pushing the segment along seems to be the right thing to do. This is shown in Figure 30, which results in the situation shown in Figure 31. This is another situation where the algorithm needs more global information in order to decide what to do. Looking only at segments A and C, it cannot accurately decide whether to move segment C or not – it needs to know which way segment E goes first.

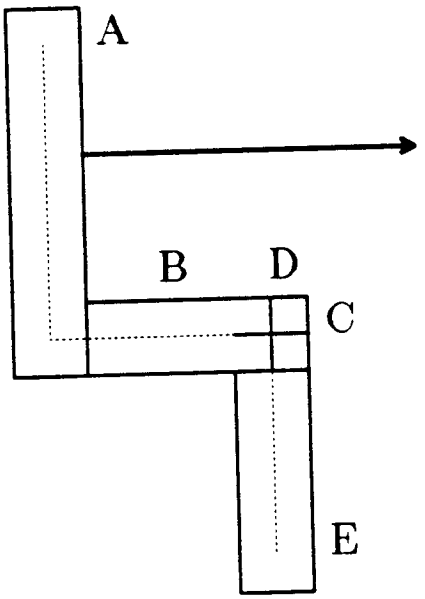

Figure 28: A difficult example involving zero length wire segments. YWIRE segment A is moving to the right, as shown. The rest of the segments along the wire are: XWIRE segment B, zero-length YWIRE segment C, zero-length XWIRE segment D, and YWIRE segment E.

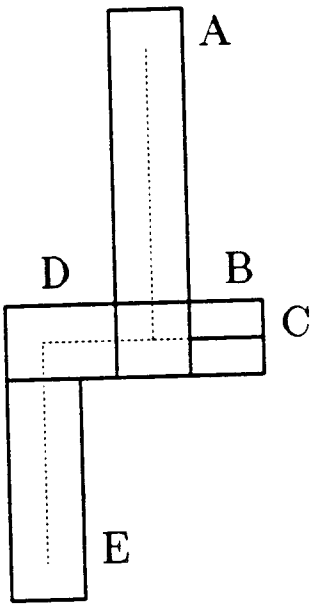

Figure 29: The final position of the move shown in Figure 28. Because segment A pushed segment C ahead of it, XWIRE segment D has to fall on top of segment A to connect the final positions of segments C and E.

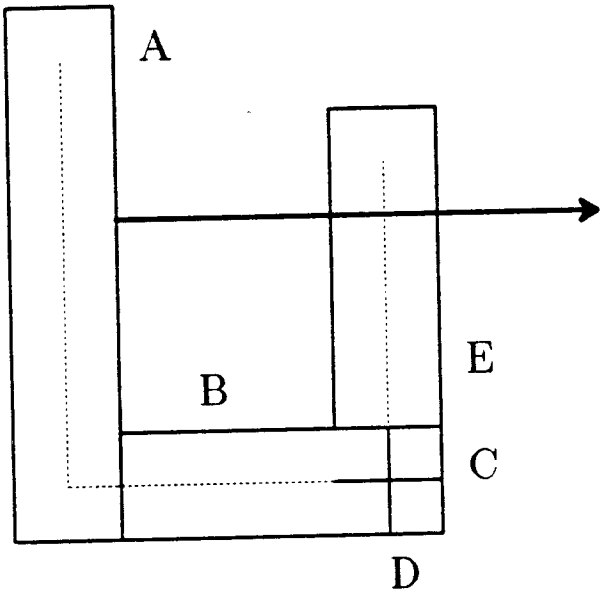

Figure 30: A related situation to the one shown in Figure 28. Here not pushing segment C along would result in an error.

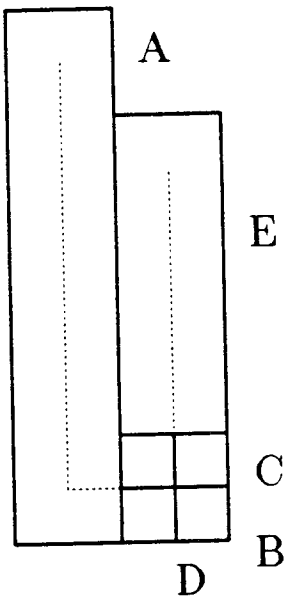

Figure 31: The final position from the move shown in Figure 30.

4.2.1.8. Solution -- Virtual Extensions

For both of the preceding difficult cases, the solution involves distinguishing between other segments in the same net and segments in other nets. The solution to the first case was inspired by the Overlapped Segments representation presented back in Section 2. If segment C were represented as an Overlapped Segment, it would extend down to the bottom of segment B. Then box D would push on the bottom of segment C in Figure 24, forcing it to move, but would miss the segment in Figure 26, leaving it alone. In order to allow the reversing of the jog, segment A would have to leave segment C alone in both cases because they are in the same net.

Although the Overlapped Segments representation is not readily implementable, the program can achieve the same effect by deriving the area that would be occupied by a segment in this representation from the Wire Extension representation of the segment. The Overlapped Segments representation of a segment is derived from its Wire Extension representation by extending the segment backwards one wire width into the previous segment. This additional area I call the *virtual extension* of the wire segment. The movement algorithm is modified so that the virtual extension of a segment is invisible when a scan is done from another segment in the same wire, but is visible (causing the segment to be scheduled) when a scan is done from any other object. Figure 24 now results in the final position shown in Figure 32.

Just as other objects can push on the virtual extension of a wire, the virtual extension needs to be able to push on other objects. In Figure 24, if segment C had been moving to the right, it would need to push on box D, just as box D needed to push on segment C when moved to the left. Thus, when scanning out from a YWIRE segment, it is necessary to scan out from the virtual extension of the wire for boxes that are not part of the same wire.

Incidentally, the virtual extension concept also handles the problem shown in Figure 23 without needing to schedule XWIRE box B with the move distance. However, the program still needs to schedule XWIRE segments to handle XWIREs with loose ends (as in Figure 22), so this code has been left intact.

The solution to the second problem case involves special treatment for zero-length YWIRE

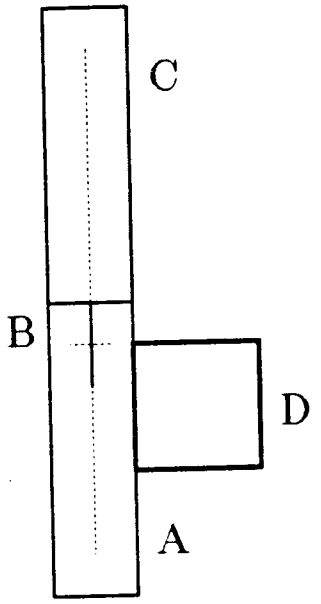

Figure 32: Final position of the move shown in Figure 24 produced by the revised algorithm.

segments. Objects not in the same net will find them and push them along like ordinary YWIRE segments (the segments are "found" by checking for electrically connected zero-length segments when any ordinary wire segment is found in the search area). However, a YWIRE segment in the same net will "catch" the segment against its back edge and push it along, reducing any intermediate XWIRE segment to zero length. This results in the final positions shown in Figures 33 and 34.

These modifications add up to the final WICRD movement algorithm.

4.2.2. Compaction

One of the benefits of having a good movement algorithm is that compaction comes along "for free" – it is just a special case of movement. WICRD uses the sweeper boxes shown in Figure 19 to compact the circuit. The program compacts the display in either the x or y direction by attempting to move the appropriate sweeper box all the way across the display to the boundary box on the other side, and then return it to its original position. If there is anything in the display, the program will not be able to move the sweeper box all the way across, but will have

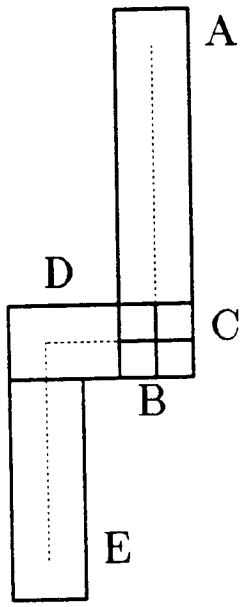

Figure 33: Final position of the move shown in Figure 28 produced by the revised algorithm.

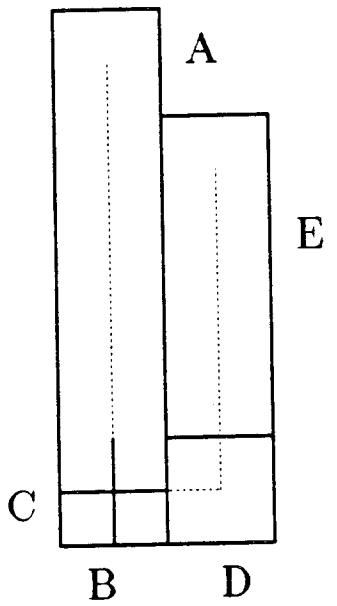

Figure 34: Final position of the move shown in Figure 30 produced by the revised algorithm.

the move distance reduced when objects run into the boundary box, which is an immovable object. The objects are compacted against the boundary box.

In general, this idea can be extended to compact portions of a circuit by defining an

immovable barrier along one edge of the region to be compacted, and defining a moving "broom" along the opposite edge. The broom can be used to sweep all the objects in between into the barrier, compacting them in the process.

This form of compaction only operates in one dimension at a time, like the algorithms in CABBAGE [Hsue79] and PYTHON [Bale82]. Those programs operated on the entire circuit at a time. WICRD demonstrates that a good incremental movement algorithm can achieve the same effect, but is more general, since it can be used to make small local changes to the circuit as well as global ones.

4.2.3. Routing

The routing algorithm in WICRD is a simple maze router based on Lee's algorithm [Lee61]. The choice of algorithm was based on the nature of routing problems in WICRD: connecting two terminals on a single wiring plane with obstacles. The single wiring plane rules out multi-plane algorithms such as channel routers and switchbox routers. The arbitrary positioning of terminals and obstacles tends to rule out river routers. So a maze router is the logical choice. Lee's algorithm was chosen as the base algorithm because it is certain to find a path between two points if one exists.

The algorithm that is conventionally known as "Lee's Algorithm" is a special case of the more general algorithm presented in [Lee61]. The conventional restriction is to finding the shortest path of orthogonally connected unblocked cells connecting two cells in a rectangular grid. This case was presented by Lee as one of his examples, but the algorithm as he presented it allowed for a general symmetric definition of the "neighborhood" of a cell, and the minimization of multiple monotonic functions along a path.

The conventional interpretation of Lee's Algorithm is that the algorithm is finding a path for a manhattan wire one cell wide. The problem with applying this interpretation to WICRD is that WICRD's alignment grid spacing is 8 pixels, while the wire width is 16 pixels. The wire width is too coarse a grid spacing, because it would rule out paths that required a wire to jog by

an odd number of half wire widths. In order not to rule out legal paths, the alignment grid spacing must be used. But this requires an extension of the conventional interpretation of the algorithm, because wires are two cells wide.

The WICRD routing algorithm begins by overlaying the user area with a rectangular grid, with each grid intersection corresponding to an alignment point. Each cell of the grid corresponds to an 8 pixel by 8 pixel area of the screen. The user has specified a source and a target box for the search in invoking the router. The source box must be a terminal; the target can be either a terminal or the free end of a wire. The program initializes each cell of the grid with an integer value corresponding to the type of box under that cell: 0 for empty space, -1 for the target box, -2 for the source box, and -3 for blockage (any other kind of box).

The program searches for the shortest path that could be swept out by a 2 cell by 2 cell square sliding orthogonally one cell at a time from a position adjacent to the source to a position adjacent to the target. It regards each cell as the potential position of the lower left corner of the square. A cell is only considered available if it is free and the three cells above and to its right are also free. Positive integers are used to mark available cells that have been reached in the expansion from the source, as in the classical Lee's Algorithm. First those available cells that correspond to positions of the square adjacent to the source terminal are marked with 1's, then available cells immediately adjacent to these are marked with 2's, and so on until an available cell that corresponds to a position of the square adjacent to the target is reached. Once this happens, the program traces back through the positively numbered cells to find the path back to the source. It adds or extends the wire along this path.

The operation of the router is illustrated in the three figures below. Figure 35 shows the problem to be routed, Figure 36 shows the resulting routing, and Figure 37 shows a portion of the cell map at the end of the search.

An alternate, superior approach would be to extend the obstacles, source, and target one cell downwards and to the right in initializing the cell map, so that the only cells marked as empty are valid positions for the lower left corner of the square. The expansion and trace back would then

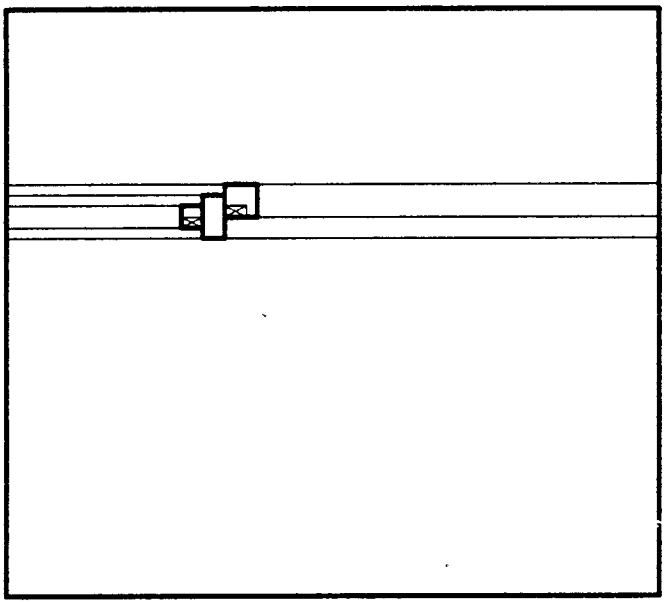

Figure 35: A problem to be routed. The terminal on the left is the target; the terminal on the right is the source.

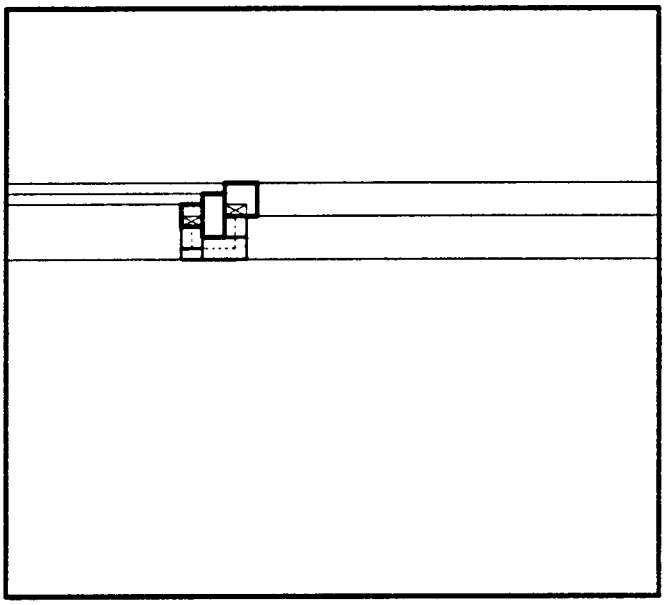

Figure 36: The final result of the routing algorithm. The wire is based on the target terminal (on the left), and extends back to the source.

be the same as in the classical version of Lee's Algorithm. This approach would be more efficient because only one value would have to be checked for each cell expanded into, rather than four.

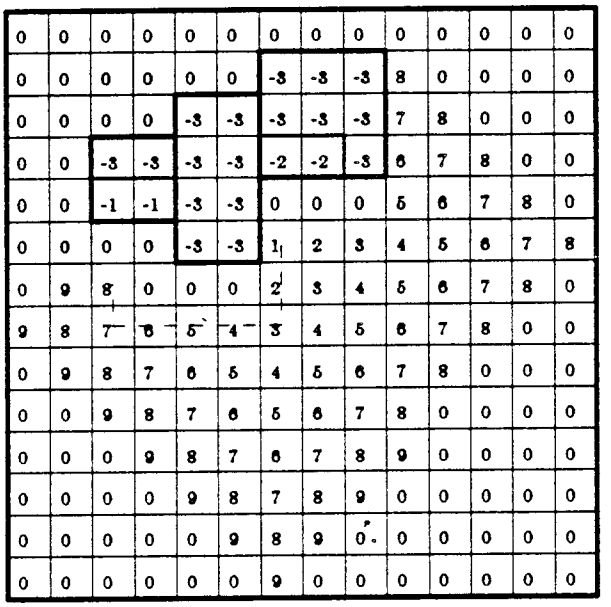

Figure 37: An closeup of the cell map at the end of the search. The route chosen in the trace back phase is shown by the dashed line. It represents the path followed by the lower left corner of a 2 by 2 square sliding from the source to the target. The target was found when the algorithm attempted to expand the cell labeled 8 at the end of the path. Only some of the cells marked 8 had been expanded when the target was found. Note that only one cell (marked 1) was considered adjacent to the source terminal, because the entire side of the 2 by 2 square represented by the cell must be adjacent to the terminal to qualify.

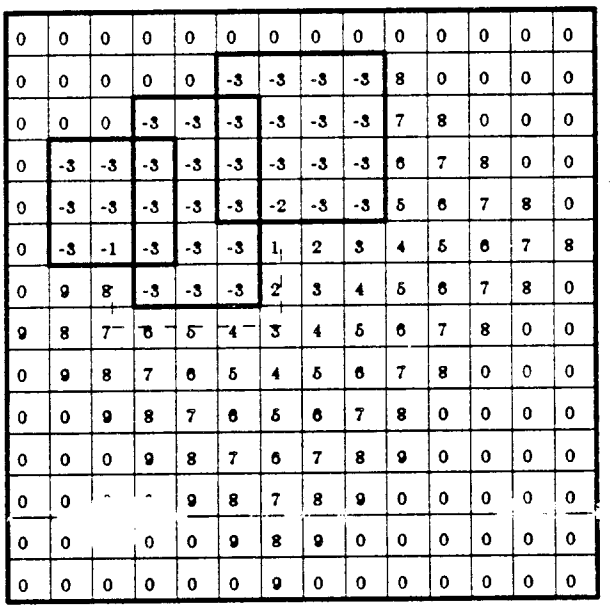

Figure 38: How the cell-map from Figure 37 would look with extended blockages. Some care needs to be taken with the source and target terminals.

Figure 38 shows how the cell map from Figure 37 would look under this approach.

4.3. Examples

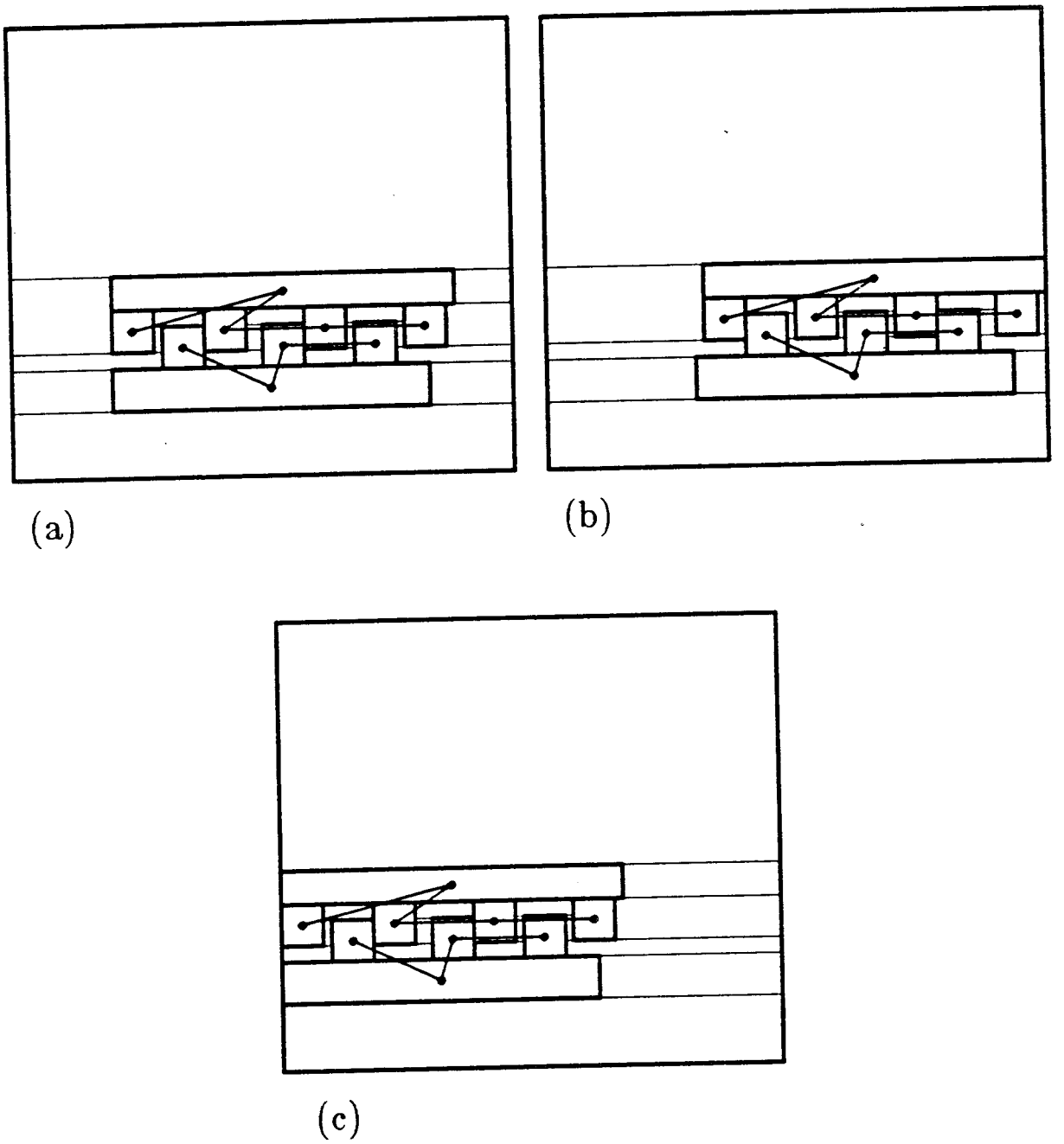

Figure 39: (a): Two macro frame "combs" with interlocking "teeth." (b): The top "comb" has been moved to the right, pushing the other comb along with it. (c): The top "comb" has been moved all the way to the left, pulling the other comb along.

An example of moving macro frames is shown by the two interlocking macro frame "combs"

in Figure 39. The two combs slide slightly relative to each other, but all boxes within a single comb always retain the same relative position to one another.

An example of compaction with macro frames is shown in Figure 40. Notice how the macro frames constrain the compaction.

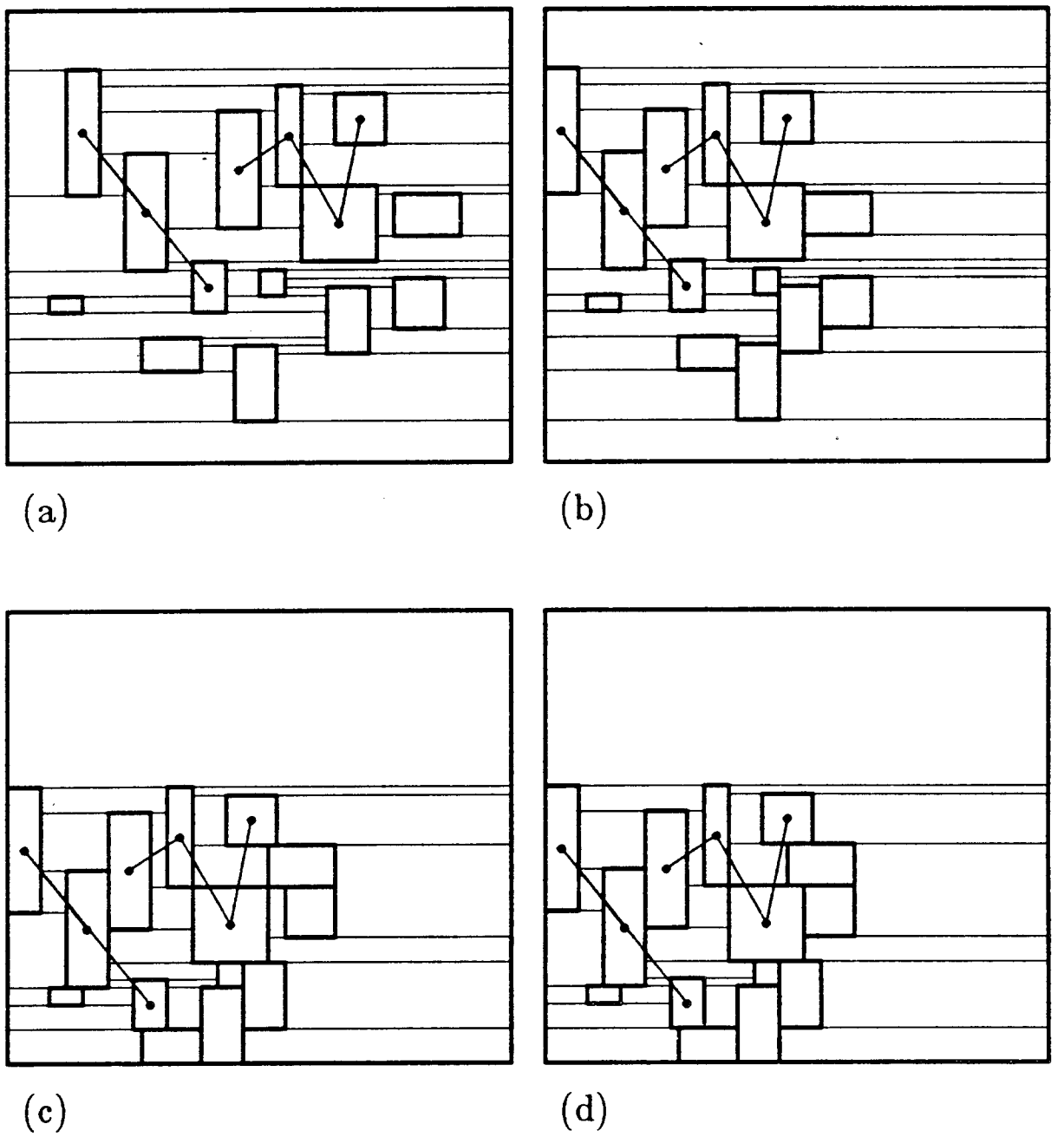

Figure 40: Four Stages in Compaction with Macro Frames. (a): Initial Position. (b): After X Compaction. (c): After Y Compaction. (d): After another X Compaction.

An example of automatically routing a wire is shown in Figure 41.

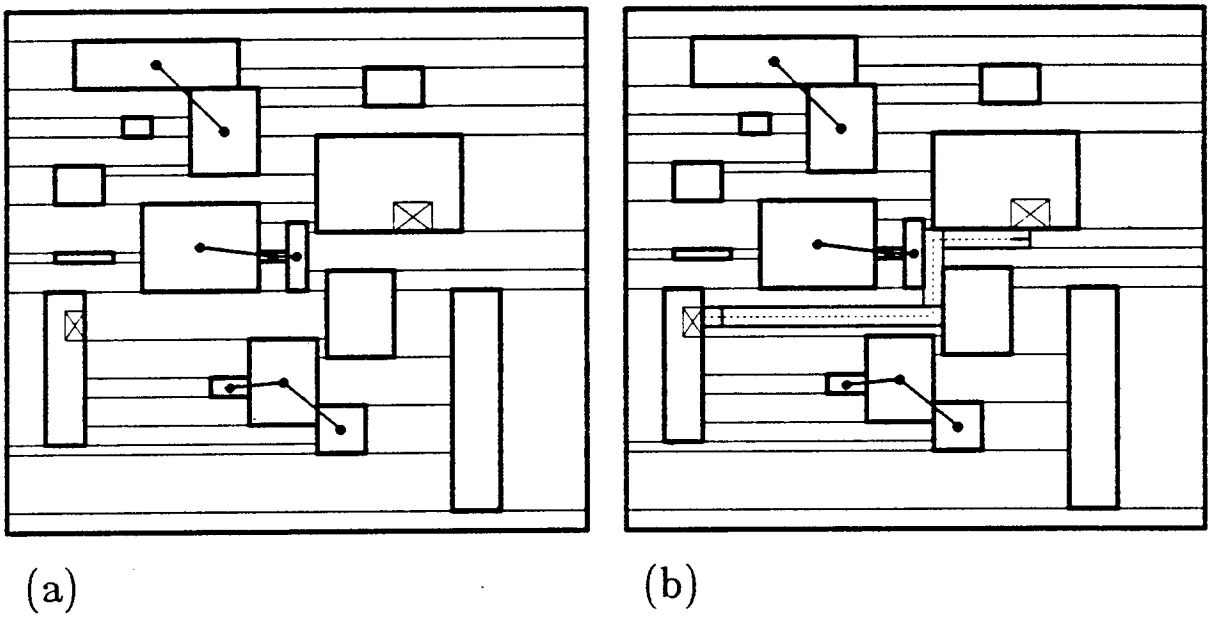

Figure 41: Routing a Wire. (a): Initial Position (the left terminal is the current selection, the right terminal is the pushed selection). (b): After Routing.

An example of moving boxes around with wires attached is shown in Figure 42.

4.4. Extensions

The WICRD model presented above does not reflect the full complexity required for a real IC layout tool. In the sections below, possible extensions to the WICRD model are discussed: extended design rules, multiple levels, multi-terminal nets and multiple wire widths.

4.4.1. Extended Design Rules

The WICRD model discussed so far finesses the question of design rules by assuming that wires and circuit boxes incorporate enough space around their edges to ensure that no two adjacent boxes can ever cause a design rule violation. This works reasonably well in many cases. For example, under Mead-Conway design rules, metal could be expanded by 1.5λ, polysilicon by 1λ, and diffusion by 1.5λ. This is sufficient to ensure that all minimum spacing requirements are met:

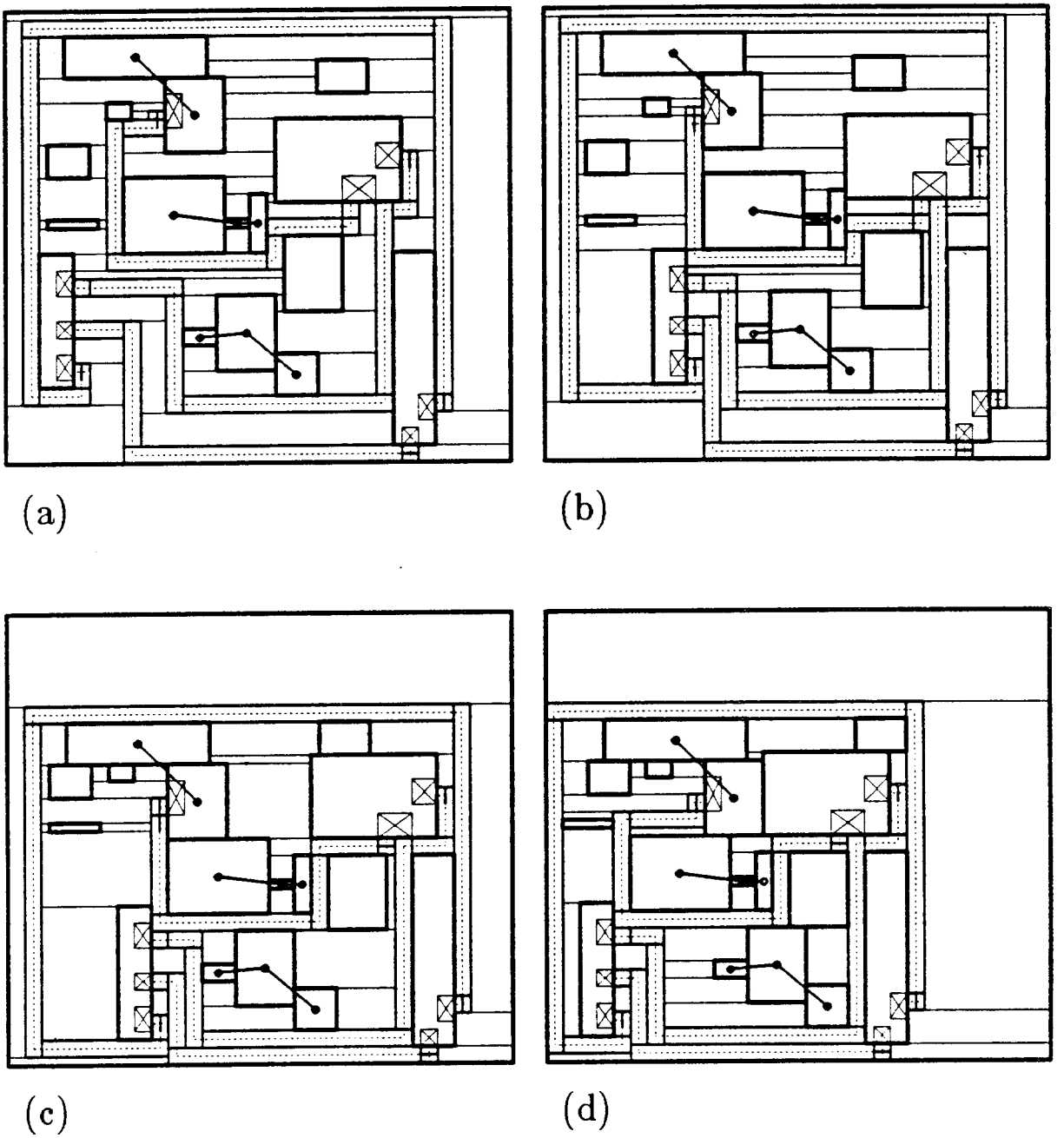

Figure 42: Four Stages in Moving Boxes with Wires. (a): Original Position. (b): After Moving Lower Left Box to the Right. (c): After Y Direction Compaction. (d): After X Direction Compaction.

metal-metal spacing (3λ), poly-poly spacing (2λ), diffusion-diffusion spacing (3λ), and poly-diffusion spacing (1λ). However, this approach results in an overly conservative poly-diffusion spacing, since the above expansions enforce a minimum spacing of 2.5λ even though only 1λ is required. Reducing the diffusion expansion to 0λ (no expansion) would fix this, but would no

longer enforce the required diffusion-diffusion spacing. This would need to be enforced with special treatment. More complex design rules may have many cases that require either special treatment or overly conservative design rules.

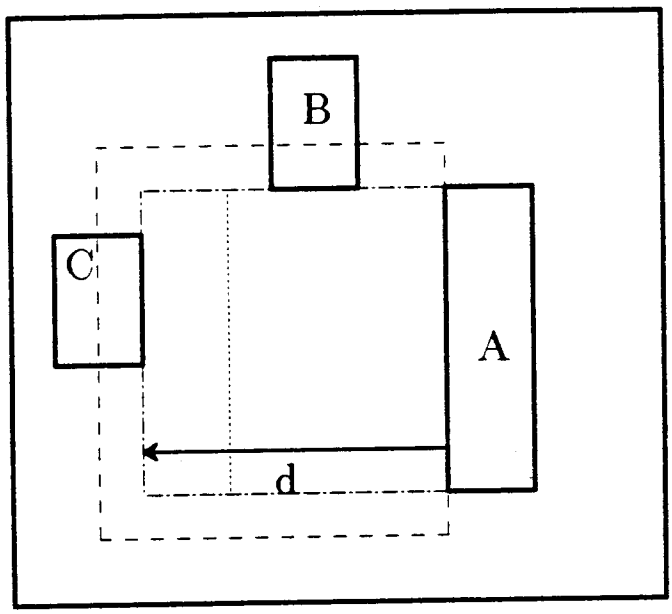

Figure 43: Moving solid boxes with extended design rules. The box labeled A is to be moved left by a distance d. The dot-dashed line shows the area swept out by box A; the dotted line shows the final position of the box. The dashed line indicates the extent of the area potentially affected by moving box A. Boxes that fall between the two areas (such as B and C) must be checked individually to see if they are affected by the design rules.

Figure 43 shows how the movement algorithm can be extended to supply the special treatment required by more complex design rules. The area swept out by box A is expanded by the maximum possible design rule interaction of other objects with box A. Objects located in the original rectangle swept out by box A would be pushed along as before, except that the distance they were moved would be increased by the minimum spacing required between them and box A. Objects not located in the original rectangle but which intersect the expanded rectangle (such as boxes B and C in Figure 43) would be checked individually to see if they should be pushed along by the movement of box A, based on the kind of objects and the design rules involved. Note that if there is a non-zero vertical spacing restriction between boxes A and B in Figure 43, box B will need to be pushed along by box A even if the distance between the final position of box A and the

current position of box B exceeds the minimum spacing rule. This is because other effects of the move may bring boxes A and B closer together, either by forcing box B to move or by reducing the move distance because of a collision with an immovable object. The program will not be able to detect the interaction between boxes A and B due to one of these effects, so it will have to move box B initially to ensure that these effects do not cause a spacing violation.

This in turn means that segments in the same wire must be treated specially in order to allow jogs to be reversed. The consecutive vertical segments of a wire are in the same kind of relative position as boxes A and B in Figure 43. If these were segments of the same wire, box B must not be affected by the movement of box A in order to reverse the jog properly. The Wire Extension representation was inspired by this problem of reversing a jog, with the idea being that the vertical segments of the wire would slide under one another automatically without the need to distinguish between segments of the same wire and segments of other wires. *But this is only true if the wire already incorporates the necessary space to ensure design rule correctness.* With extended design rules, the program needs to make this distinction anyway, so Wire Extension no longer offers this advantage over other representations.

Taylor and Ousterhout have observed in [Scot83] that if spacing violations already exist in a layout, attempting to move one of the objects involved could cause an infinite loop in the movement algorithm unless special precautions are taken to detect this case and break the cycle. The current implementation of WICRD finesses this problem because it is not possible for the user to create a spacing violation if all required space is incorporated into the tiles. However, under extended design rules this problem could occur in WICRD. The program would have to either adopt a solution similar to MAGIC's, or else ensure that no spacing violations could ever be created. The first alternative is more flexible, because the design rules could be changed after layout had started. Changing design rules is never a pleasant experience, but it is not unheard of in industry when one is pushing the limits of a new technology. A program that is flexible enough to allow for such changes will be more valuable than one that is not.

4.4.2. Multiple Levels

Although WICRD works with a single level, multiple levels can be simulated with the current WICRD structure. The user can divide the screen into two regions and use links to create structures that span both levels, such as contacts and circuit boxes. Compaction can be simulated by hand. Automatic routing between the simulated levels is not available, although routing within a level is still possible.

Figure 44 shows an example in which the three contacts are the only structures occupying both levels. The WICRD screen simulating two levels is shown together with an illustration of the combined layout. Figure 45 shows the same scene after the circuit on level 1 has been moved to the right. The bottom wire collapses to minimum size and pushes on its contact, which forces the bottom level 2 circuit to move to the right. This circuit then pushes on one of the wires connected to the other level 2 circuit.

4.4.3. Multi-Point Nets and Multiple Wire Widths

Extending the WICRD model to handle multi-point nets and nets with multiple wire widths is likely to be difficult. One would encounter the problems noted previously because of the directed box representation. The program would have to either enforce arbitrary constraints on movement to avoid generating any problem constructs, or else use an approach including the use of filler tiles where needed. This last approach would increase the complexity of the program considerably. Not only would there be yet another type of tile for all the algorithms to deal with, but the filler tiles would not be mapped to a particular wire segment, and would pose difficulties in how they would be linked to the associated wire.

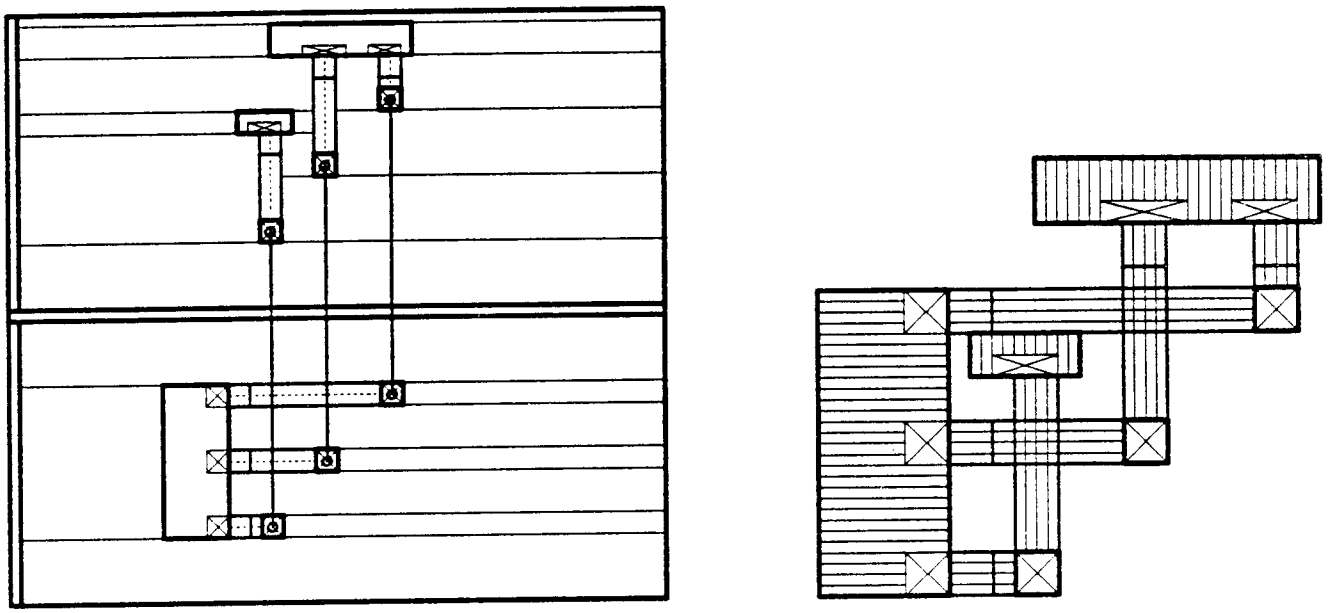

Figure 44: The left side of the picture shows the WICRD simulation. The right side of the picture shows the corresponding layout. Horizontal cross-hatching indicates level 1; vertical cross-hatching indicates level 2. The three contacts occupy both levels.

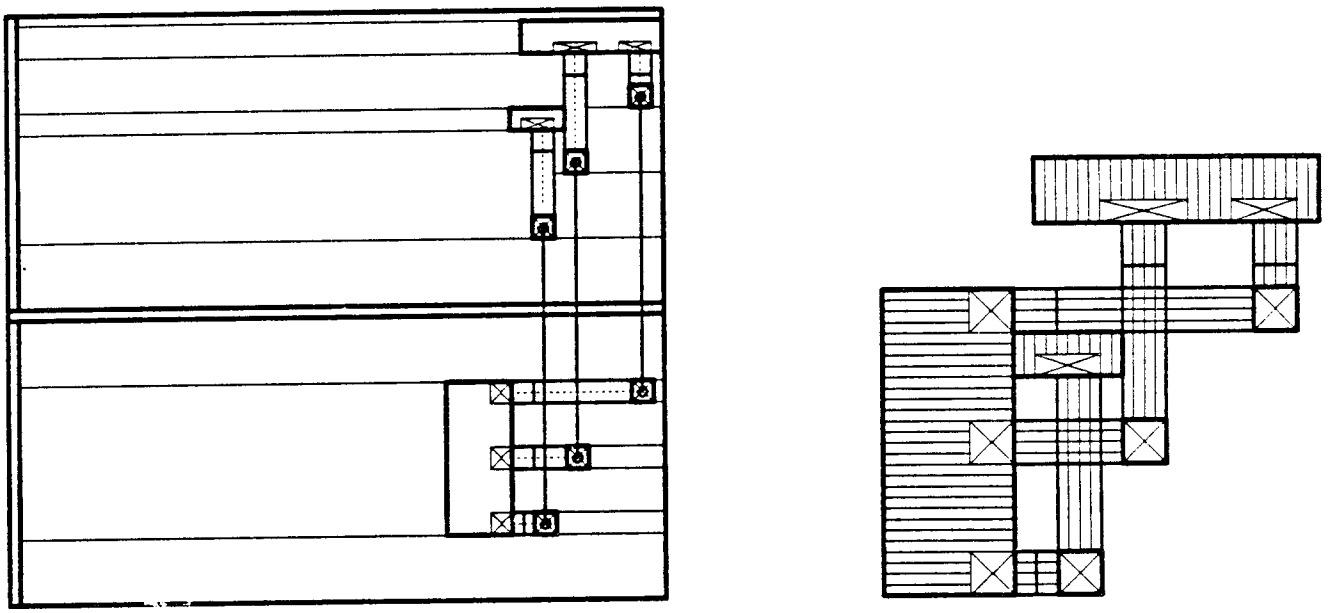

Figure 45: The example from Figure 44 after the level 1 circuit box has been moved to the right. The left side of the picture shows the WICRD simulation. The right side of the picture shows the corresponding layout. Horizontal cross-hatching indicates level 1; vertical cross-hatching indicates level 2.

5. Conclusions

In comparing the merits of the different classes of wire representation, no one class is clearly superior to the others on all counts. However, the directed box representations, explored in detail in WICRD, are limited in capturing the full range of physical forms of wires.

The idea of directed box representations was to combine the advantages of skeletal representations with the purely physical ones by having the tiles of a wire occupy the physical space associated with its geometry while retaining a straightforward mapping between individual tiles and the segments of the wire. The Overlapped Segments representation is the theoretically ideal directed box representation. There is one box per segment that occupies all the physical space associated with that segment. That is why this representation is so good at expressing the semantics of how moving segments interact with other objects.

The problems with this representation arise when we try to force it into the framework of non-overlapping rectangular tiles. The area associated with a segment intrinsically overlaps whatever it connects to. To deal with the area of overlap, we can either isolate the overlaps into separate tiles, as in Joint Patches, or we can incorporate the area into one of the segments, as in Wire Extension. Neither approach is completely successful. In particular, both representations have difficulty connecting to the corners of other objects. This difficulty is a recurring theme that restricts connections to terminals, connections to other wires, and connections to contacts between levels.

If these were merely restrictions on the designer's freedom when creating an initial design, they might be acceptable as part of the price of design automation. But they also restrict the program's ability to move objects around, since it cannot be allowed to create a forbidden construct. Figures such as the "pinwheel junction" are the natural result of moving wires by fractional wire widths.

One approach to this problem of connecting to corners involves the use of "filler" boxes to fill in any area of these corner connections that doesn't fit into one of the segments. This is a variation on the Joint Patches theme of isolating overlap areas into separate tiles. However, filler

tiles occur more irregularly than corners do in Joint Patches. It can be difficult to determine how to associate filler tiles with wire segments, particularly in a multi-wire junction, or where there are multiple overlaps. Because of this, it is difficult to define how filler tiles should interact with other objects. At the very least, I would expect them to make WICRD's movement algorithm much more complex.

Is Wire Extension an overly specialized representation? Based on my experience in writing WICRD, I think that it is. This representation was inspired by the problem of reversing a jog in a wire, with the idea that it would allow jogs to reverse automatically, without any need for special treatment from the program. But as we have seen, its special effectiveness on this particular problem depends on the assumption that wires incorporate sufficient space to ensure design rule correctness. When extended design rules are used, the program needs to apply the same kind of special treatment that it would for other representations. Meanwhile, the specialized structure that worked moderately well on this one particular problem kept getting in the way when I tried to extend the representation to solve other problems. The asymmetrical relationship between the position of the segment's centerline and the position of the corresponding tile posed special problems both in coding algorithms, and in trying to come up with a consistent way to represent multi-way wire junctions, in addition to the physical representation issues already discussed.

In retrospect, I think it is most important for a wire representation to do a good job on the physical representation issues, and to have a simple, consistent paradigm for how wire configurations are represented. It is more important for a data structure to be flexible than intelligent: intelligence can be supplied by the program that manipulates it. I can always program jog reversal as a special case in the movement algorithm, but no amount of programming will give me a multi-way junction between wires of different widths unless the underlying representation can provide it.

For this reason, I recommend against using directed box wire representations for tile-based IC layout tools. I would prefer to use either a skeletal representation for the semantic information, or a purely physical representation for generality and efficiency. This conclusion does not

necessarily apply to wire representations that are not based on a paradigm of non-overlapping tiles. Overlapped Segments might well prove viable if it could be implemented effectively. But the Wire Extension representation used in WICRD does not seem to be a profitable approach. Although it might be possible to make it work, the potential returns do not appear to justify the effort required.

The WICRD compaction algorithm is a trivial application of the movement algorithm. This shows that compaction is a special case of movement. Any effective incremental movement algorithm can also be used to do compaction.

The WICRD routing algorithm shows how Lee's algorithm can be extended to cases where the alignment grid spacing is less than a wire width. The resulting algorithm will find the shortest legal path between two points, although it may get expensive both in space and time if the alignment grid spacing is too fine.

How does WICRD compare to Magic? The programs result from two different lines of development from the original corner stitching idea. WICRD, of course, is only a demonstration program, where Magic is a real design tool. WICRD relies on the directed box representation Wire Extension, whereas Magic uses the purely physical representation Paint. WICRD's movement algorithm is based on moving objects, whereas Magic's movement algorithm is based on moving edges. WICRD uses the elastic pole model for movement, whereas Magic uses the wet noodle model. The differences in these movement algorithms are based on the differences in the wire representations used. Both movement algorithms can also be used for compaction.

Which approach is better? The wet noodle model appears to be better suited to small, local incremental changes. If a small object pushes on the middle of a long wire, it probably won't wind up plowing the whole chip along in front of it, as it might in the elastic pole model. Experience with WICRD suggests that the elastic pole model tends to result in a few long wire segments with a bunch of zero length segments piled up at the ends. Although the user can theoretically exercise more control over the final result in the elastic pole model by manually positioning jog points, it is not clear that the users want or need that much control. Having to carefully position

all those jog points may be much more trouble than it's worth. On the other hand, Magic does not appear to have anything corresponding to the WICRD concept of an immovable object, which could be useful in performing localized compactions, and in restricting the area affected by a move. To implement this would require the kind of changes to the algorithm that were required in going from Algorithm M1 to Algorithm M2.

In short, the WICRD program itself does not appear to be a worthwhile basis for future development: the attempt to combine the best of two worlds in the directed box representation used in WICRD did not succeed. However, many of the ideas in WICRD may prove useful in future programs.

6. Acknowledgements

The WICRD program was based on a demonstration box packing program written by John Ousterhout. Many of the ideas in WICRD were developed out of discussions with Mike Arnold, Dan Fitzpatrick, John Ousterhout, Carlo Séquin, and Dave Ungar. The concepts described from MAGIC that are not a part of WICRD are due to the MAGIC implementation team: John Ousterhout, Bob Mayo, Walter Scott, George Taylor, and Gordon Hamachi.

This work was supported by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 3803, monitored by Naval Electronic System Command under Contract No. N00039-81-K-0251.

7. Appendix: WICRD Configuration and User Interface

This appendix covers the hardware configuration WICRD runs on and WICRD's user interface. The discussion on the user interface is divided into general comments, a description of the puck button interface, and a description of the command interface.

7.1. WICRD Hardware Configuration

WICRD uses the same hardware configuration as the VLSI graphics editor Caesar [Oust81]. There is an alphanumeric terminal for text input and output, a color graphic display terminal for viewing the layout, and an attached tablet with a four-button puck for pointing at locations on the graphic display. The graphic display terminal is typically an AED 512, although other types of displays are also supported. WICRD runs on a VAX-11/780 or 11/750 running Berkeley UNIX.*

7.2. WICRD User Interface -- General

WICRD uses texturing to indicate an object's degree of selection. Any box, wire, or terminal will always be in one of three states: the *current selection*, the *pushed selection*, or *unselected*. The intensity of the stipple pattern used to display the object indicates its degree of selection. A box or terminal that is the current selection is displayed in a solid color; a wire that is the current selection is displayed with a solid centerline. A box or terminal that is the pushed selection is displayed with a diamond cross-hatched texture; a wire that is the pushed selection is displayed with a dashed centerline. Unselected boxes and terminals are displayed with a fine checkerboard texture; unselected wires are displayed with dotted centerlines. In all cases, the current selection has the most intense visibility, the pushed selection has intermediate visibility, and unselected objects have the least intense visibility.

Another feature of the WICRD user interface is the availability of an on-line "explain"

*UNIX is a trademark of Bell Telephone Laboratories.

command documenting the WICRD commands. When the user requests an explanation of a given command, the program prints a descriptive paragraph or two about the command in question. It does this by looking in a directory called EXPLAIN for a file with the same name as the command the user is requesting help for. If it finds it, it prints the file on the alphanumeric display. Although this is not a difficult or conceptually sophisticated feature, I think it adds a helpful touch that is absent from many existing editors. These explain files are presented below in the section on the WICRD command interface.

7.2.1. WICRD Button Interface

The WICRD user locates points on the screen with a digitizer tablet and attached "puck." A cross hair on the screen follows the position of the puck on the tablet. The puck has four buttons, arranged in a diamond shape. The general function of each button was chosen to be the same as the function of the same button in Caesar, although the individual semantics are somewhat different, because of the different environments.

WICRD, like Caesar, uses a rectangular cursor to designate the area of the screen where an operation is to be performed. As in Caesar, two of the puck buttons position the cursor. The left button (white) moves the cursor so that the lower left corner of the cursor is at the cross hair. In general, this operation leaves the cursor size fixed; however WICRD will clip the cursor if necessary to keep it within the user area of the screen. The right button (green) changes the size of the cursor. It leaves the lower left corner of the cursor fixed, and stretches the upper right corner of the cursor to be under the cross hair. The cursor is not allowed to have a negative length or width, although it may have a zero length or width. However, circuit boxes and terminals can only be drawn under the cursor if it has non-zero length and width.

The top button (yellow) is the selection button, as in Caesar. If the cross hair is over a solid box, the box is selected. If the cross hair is over a terminal of a solid box, both the terminal and the box are selected. Which one is used as the selection depends on the following command. If the cross hair is over a wire box, the corresponding wire segment is selected. Selection is shown to

the user by using a more intense stipple pattern for the selected object. If the curtrack switch is on, the cursor will be positioned to fit the newly selected object.

The bottom button (blue) is the "paint" button, as in Caesar. It generally tries to make the area under the cursor look like the object pointed to with the cross hair, but in a much more limited way than Caesar. If the cursor is positioned over empty space, and the cross hair is over a solid box, WICRD will create a new solid box under the cursor. If the cursor is positioned over some or all of a solid box, and the cross hair is over a terminal on some solid box, WICRD will try to create a terminal the size of the cursor. Unlike Caesar, erasing objects is done based on the current selection, not the location of the cursor (although if the curtrack switch is on and the cursor hasn't been moved, the current selection will be under the cursor). If the cross hair points to empty space, the current selection (if any) will be erased. If the cross hair points to a solid box but not to a terminal, WICRD will first assume that the user wants to draw a solid box. If this fails (because the cursor is positioned over one or more solid boxes already, or if it has zero length or width), it will then assume the user really wanted to erase the current terminal, to make the current selection look like the object pointed to by the cross hair (solid box, but no terminal).

Rather than use the raw coordinates from the tablet, WICRD has an alignment grid, with grid lines every eight pixels. All coordinates coming from the tablet are truncated to the next lower grid position. Truncation was chosen rather than rounding because of the "paint" function described above. We wanted the object painted from to be the object visibly under the cross hair, which would not always be the case if rounding was used.

7.2.2. WICRD Command Interface

The following descriptions of WICRD commands are taken from the on-line documentation available in WICRD by using the explain command. The commands are presented in alphabetical order.

clear

Clear erases all boxes, terminals, and wires from the work area and the display.

connect

Connect joins a wire and a terminal. The terminal must have been previously pushed (see the "push" command). The wire must come in to the terminal head on, and must already come all the way up to the terminal (the display will show the wire centerline stopping one wire radius away from the terminal). The wire segment must be currently selected. After the command is executed, the centerline will appear to extend all the way up to the terminal.

curtrack on

curtrack off

Curtrack toggles a switch that determines whether the cursor (the green outlined box) tracks the current selection. When curtrack is on, the cursor will be adjusted to frame the currently selected box whenever a command is given or a button is pushed that selects a new box as one of its effects. When curtrack is off, the cursor position and size are only changed by the cursor position and cursor size buttons on the mouse.

debug on

debug off

Debug toggles a switch that determines whether the addresses of the box data structures are shown on the display. When debug is on, each space box, solid box, terminal, and wire is shown with a number superimposed on it. This number is the address (in decimal) of the data structure representing that box, or at least as many digits as will fit. This information is useful when debugging the program with sdb.

draw

Draw creates a SOLID box under the current position of the cursor. The effect is the same as pointing the cross hair at an existing SOLID box and pushing the "paint" button (however, see "termdraw" and "termerase" for possible side effects on terminals that are not present when the draw command is used). If the cursor is not positioned entirely over SPACE boxes, it is an error.

erase

Erase erases the currently selected SOLID box, if any. The box is unlinked from any other SOLID boxes to which it may be currently linked. All terminals on the box are erased. Any wires originating from (not terminating at) any of those terminals are also erased. If the currently selected box is a wire, the wire will be erased from that segment on. The effect of this command is the same as positioning the cross hair over a SPACE box and pushing the "paint" button.

explain commandname

Explain provides an explanation of commands in WICRD. Information is currently available on the following commands:

clear	explain	pop	route	xsmash
connect	gremlin	previous	split	xwire
curtrack	help	push	termdraw	ymove
debug	join	quit	termerase	ysmash
draw	mtrace	read	write	ywire
erase	next	redisplay	xmove	

gremlin filename

Gremlin writes the current arrangement of boxes and wires on the screen in a gremlin file. It will save them in the specified file. The boxes are left on the screen as they were. This command produces an editable and printable version of the picture on the screen.

help

Help displays a list of the legal commands of WICRD. When specifying commands to WICRD, any command may be abbreviated by its first few letters, as long as enough letters are given to uniquely identify the command.

join

Join links two SOLID boxes together. One box must have been previously pushed (see the "push" command), and the other must be the current selection. The link is shown graphically by a line connecting the centers of the two boxes. The program will force linked boxes to retain the same relative positions as they are moved around the screen (see "xmove," "ymove," "xsmash," and "ysmash"). Thus, all boxes linked together move as a unit. Boxes can be removed from the link via the "split" command.

mtrace on

mtrace off

Mtrace toggles a switch that determines whether a trace of the mazerunner search is written to file "maze.trace" when the "route" command is invoked. The default (for now) is on. When mtrace is on, the router takes longer to return to the user, because of the overhead of file processing. If the trace file already exists, it will be replaced the first time "route" is used with the mtrace switch on. Subsequent uses of "route" in the same session will append their traces to the end of the file.

next

Next advances the current selection to the next segment along a wire. If the next segment is a terminal, both the terminal and the SOLID box it is a part of will be selected. If a terminal at the head of a wire was previously selected, the selection will be advanced to the first segment of the wire (usually the "connection" box linking the terminal to the wire). Both "next" and "previous" may be used to determine the direction of a wire, by seeing which way the selection moves along it in response to the commands. These two commands are the only way to select a zero length segment of the wire, by first selecting a nearby segment of the wire, then stepping along it until the desired segment is reached.

pop

Pop is the inverse of "push." It selects the pushed box or terminal, and erases their status as the pushed box or terminal. Two pop's in a row will leave all the boxes unselected and unpushed.

previous

Previous moves the current selection back to the previous segment along a wire. If the previous segment is a terminal, both the terminal and the SOLID box it is a part of will be selected. If a terminal at the tail of a wire was previously selected, the selection will be moved back to the last segment of the wire (usually the "connection" box linking the terminal to the wire). Both "next" and "previous" may be used to determine the direction of a wire, by seeing which way the selection moves along it in response to the commands. These two commands are the only way to select a zero length segment of the wire, by first selecting a nearby segment of the wire, then stepping along it until the desired segment is reached.

push

Push pushes the currently selected wire, box, or terminal for later use in a two operand command, such as "connect" or "join." When an object other than the pushed selection is selected on the screen, the pushed selection is shown with a texture intermediate between that of the ordinary, unselected objects, and that of the current selection. Push can be undone with a "pop" command.

quit

Quit ends the program. No status is saved. If you want to save the current arrangement of boxes on the screen, you must save them with the "write" command before leaving the program with "quit."

read filename

Read restores an arrangement of boxes that was previously saved with the "write" command. In general, the screen should be cleared with the "clear" command before doing the "read." If boxes are already on the screen, the read command will try to merge the current arrangement with the saved arrangement in the file, which will work if the new boxes only occur where there are no existing boxes on the screen. However, if collisions occur, the resulting side effects can cause several bizarre errors. The read command should produce at least one error message if this happens.

redisplay

Redisplay just redraws all the boxes on the graphics screen. It can sometimes be used when the screen has gotten screwed up in some way.

route

Route uses a mazerunning router to try to connect two points with a series of wire segments. The point wired to must be a terminal, and should be already pushed with the "push" command. The other point can be either a terminal or the free end of an existing wire. This should be the selected object. To use the router, select and push the terminal to be wired to, then select the object to be wired from, and type "route." If the router can find a path for a wire connecting the two objects, it will do so. Otherwise, it will report "No route found." The router will not move any objects out of the way – it just looks to see if a wire can be inserted around any existing objects.

split

Split unlinks the selected SOLID box from any other SOLID boxes it may be linked to. It can be used to undo the effects of a "join" command.

termdraw

Termdraw draws a terminal under the cursor. The cursor must be positioned entirely over a single SOLID box, must coincide with at least one edge of that box, and must not overlap any other terminals on the box. If the cursor is correctly positioned, the effects of this command can be duplicated by pointing the cross hair at an existing terminal and pushing the "paint" button on the mouse. To be safe, you should position the cross hair at the lower left corner of the box you wish to paint from, since undesirable things can happen if the program thinks you're pointing at something else.

termerase

Termerase erases the currently selected terminal, if any. The effects of this command can be duplicated by positioning the cross hair over a SOLID box in an area not occupied by a terminal and pushing the "paint" button on the mouse, provided the cursor is not positioned over space (as otherwise, the program will assume you want to draw a SOLID box under the cursor). You should be careful how you point with the cross hair -- the safest way is to point at the lower left corner of the area you wish to paint from, since undesirable things can happen if the program thinks you are pointing to something else.

write filename

Write saves the current arrangement of boxes and wires on the screen for future use. It will save them in the specified file, where they can be restored with the "read" command. The boxes are left on the screen as they were. You will often want to do a "write" command before leaving the program with "quit."

xmove

Xmove moves a box in the x direction. The program will try to move the currently selected box in the x direction so that the left edge of the box coincides with the left edge of the cursor. It will push other boxes along if necessary. If the program cannot move the box all the way to its new position, because some box would have to be moved off the screen, it will move the box as far as it can without pushing any box off the screen.

xsmash

Xsmash compacts the current arrangement of boxes in the x direction. It uses a "snowplow" approach that moves boxes to the left on the screen until some box would be pushed off the screen if the "plow" were to move any further.

xwire

Xwire draws a wire in the x direction. It will draw a wire horizontally from the currently selected object (terminal or wire) until the centerline of the wire coincides with the left edge of the cursor. If the selected object already was an x direction wire, its length will be adjusted so that the centerline reaches to the left edge of the cursor. Note: if you intend to draw several segments of a wire (using "xwire" and "ywire"), you may want to turn cursor tracking off before you start (using the "curtrack" command), as otherwise the continual repositioning of the cursor may appear confusing. Since only the position of the lower left edge of the cursor is important when wiring, you may want to reduce the cursor to a single point, and then just position the cursor at the successive corners of the wire's centerline, followed by the appropriate ("xwire" or "ywire") command.

ymove

Ymove moves a box in the y direction. The program will try to move the currently selected box in the y direction so that the bottom edge of the box coincides with the bottom edge of the cursor. It will push other boxes along if necessary. If the program cannot move the box all the way to its new position, because some box would have to be moved off the screen, it will move the box as far as it can without pushing any box off the screen.

ysmash

Ysmash compacts the current arrangement of boxes in the y direction. It uses a "snowplow" approach that moves boxes down on the screen until some box would be pushed off the screen if the "plow" were to move any further.

ywire

Ywire draws a wire in the y direction. It will draw a wire vertically from the currently selected object (terminal or wire) until the centerline of the wire coincides with the bottom edge of the cursor. If the selected object already was an y direction wire, its length will be adjusted so that the centerline reaches to the bottom edge of the cursor. Note: if you intend to draw several segments of a wire (using "xwire" and "ywire"), you may want to turn cursor tracking off before you start (using the "curtrack" command), as otherwise the continual repositioning of the cursor may appear confusing. Since only the position of the lower left edge of the cursor is important when wiring, you may want to reduce the cursor to a single point, and then just position the cursor at the successive corners of the wire's centerline, followed by the appropriate ("xwire" or "ywire") command.

References

Bale82. Mark W. Bales, "Layout Rule Spacing of Symbolic Integrated Circuit Artwork," UCB/ERL M82/72, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, California (May 1982).

Hsue79. Min-Yu Hsueh, "Symbolic Layout and Compaction of Integrated Circuits," UCB/ERL M79/80, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, California (December 1979).

Kell82a. Kenneth H. Keller and A. Richard Newton, "KIC2: A Low-Cost, Interactive Editor for Integrated Circuit Design," *Digest of Papers, Compcon*, pp. 305-306 IEEE, (Spring, 1982).

Kell82b. K. H. Keller, A. R. Newton, and S. Ellis, "A Symbolic Design System for Integrated Circuits," *Proceedings of the 19th Annual ACM/IEEE Design Automation Conference*, Las Vegas, Nevada, pp. 460-466 (June 1982).

Lawl76. Eugene Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston (1976).

Lee61. C. Y. Lee, "An Algorithm for Path Connections and Its Applications," *IRE Transactions on Electronic Computers*, pp. 346-365 (September 1961).

Lips79. Witold Lipski, Jr., Elena Lodi, Fabrizio Luccio, Cristina Mugnai, and Linda Pagli, "On two-dimensional data organization II," *Fundamenta Informaticae II*, pp. 245-260 Annales Societatis Mathematicae Polonae, (1979).

Oust81. John K. Ousterhout, "Caesar: An Interactive Editor for VLSI," *VLSI Design* II(4) pp. 34-38 (Fourth Quarter 1981).

Oust83. John K. Ousterhout, Gordon T. Hamachi, Robert N. Mayo, Walter S. Scott, and George S. Taylor, "Magic: A VLSI Layout System," A Collection of Papers on Magic, Report No. UCB/CSD 83/154, Computer Science Division (EECS), University of California, Berkeley, California 94720 (December 1983). (To appear: 1984 IEEE/ACM

Design Automation Conference)

Oust84. John K. Ousterhout, "Corner Stitching: A Data Structuring Technique for VLSI Layout Tools," *IEEE Transactions on Computer-Aided Design* **CAD-3**(1) pp. 87-100 (January 1984).

Scot83. Walter S. Scott and John K. Ousterhout, "Plowing: Interactive Stretching and Compaction in Magic," A Collection of Papers on Magic, Report No. UCB/CSD 83/154, Computer Science Division (EECS), University of California, Berkeley, California 94720 (December 1983). (To appear: 1984 IEEE/ACM Design Automation Conference)