TECHNOLOGY RETARGETING FOR IC LAYOUT

John Lakos
Mentor Graphics Corporation
Warren, NJ 07059

Abstract

The ability to recognize polygon-based layout as a collection of objects representing circuit elements connected by path-based wires, enables existing designs implemented using an older fabrication process to be reimplanted quickly in a new process. The approach taken here, based on layout generator technology, is to create a collection of parameterized circuit objects that, with appropriate arguments, are able to represent the devices (e.g., transistors, contacts) implicitly described in the flattened design. The recognition engine is fully programmable, is independent of any particular technology or device set, and is not restricted to manhattan or even octilinear geometries. In this paper, we describe a novel three-phase approach to object recognition: device recognition, exact wire recognition, and wire synthesis. We believe exact wire recognition to be entirely new and cover it in detail. Experimental results demonstrate the effectiveness of the algorithms on actual layout.

1. Introduction

The electronics industry has amassed a huge inventory of intellectual property (IP) in the form of polygon-based layout of integrated circuit (IC) designs. Computer-aided design (CAD) layout tools of the past [1] represented transistors and contacts as the intersection of polygons on separate layers. More recently, the trend has been toward object-based layout [2,3,4,5].

1.1. Object Based Layout

Treating devices as objects has several advantages:

- Grouping semantically related geometries on multiple layers to form a single cohesive unit enables devices to be manipulated at a level of abstraction higher than that of intersecting polygons.
- Creating a netlist is no longer an extraction process [6,7,8,9]; the netlist is explicit in the internal representation.
- Each device type (e.g., `NTRAN`) can be parameterized to form a generator. Instead of providing separate cells for each transistor size, a single generator element can adapt to any valid size based on the current value of its arguments, further increasing flexibility.
- By extending the set of parameter types to include non-scalar primitive values, we are able to describe procedurally a bent-gate transistor simply by defining its gate region with a `Path` argument.
- Layout compaction is simplified; an entire device moves naturally as a unit.

Often the only available representation of these designs is the GDSII stream format used during fabrication (for the following reasons):

0-89791-847-9/97/0006/$3.50 DAC 97 - 06/97 Anaheim, CA, USA

- The design was originally implemented using a polygon-based layout tool.
- The original design was developed elsewhere and only the GDSII mask data is provided.
- The object-based tool that created the layout is no-longer available.

The industry needs a way to recover the objects (devices and wires) that were used to create the original layout.

1.2. Retargeting Process Overview

The entire retargeting process assumes the existence of a persistent, hierarchical repository of transistor-level layout information. The steps of this process are illustrated in Figure 1.

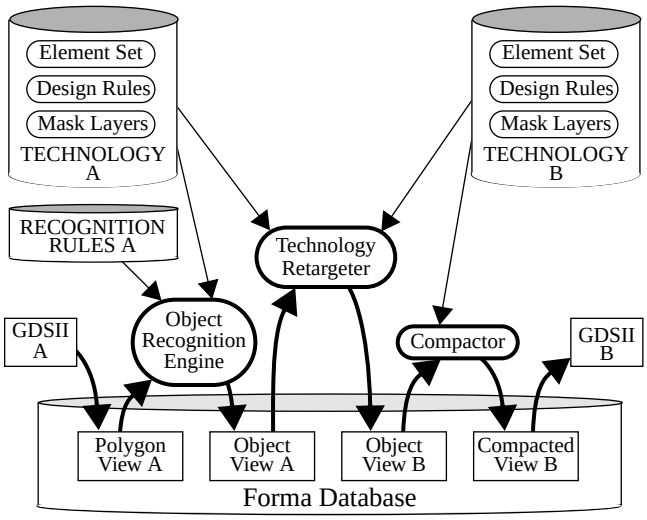


Figure 1: Retargeting Process Data Flow

a. A file containing polygon data is converted to an equivalent representation in the form of a polygon view of the cell inside the Forma™ database. (Forma is described in Section 1.3).

b. The polygon view, along with a compatible technology defining appropriate mask layers, design rules, and element types, are supplied to the object recognition engine. In addition, recognition rules are supplied that describe to the engine *how* to recognize each of the devices and wires within the polygon view. The result is the object view of the same cell in the same technology as illustrated in Figure 2.

c. For technology retargeting to be meaningful, each device and wire configuration in the old technology will have a (procedural) mapping to its semantically equivalent object in the new technology. Using Forma's extension language, the technology retargeter can iterate over the devices and wires in Object View A, using the mapping to populate Object View B with the corresponding objects defined in the new technology.

d. With devices now represented as objects in the new technology, it is easy to use Forma's extension language to procedurally

resize, replace, or otherwise manipulate any device as needed, while Forma continues to preserve connectivity.

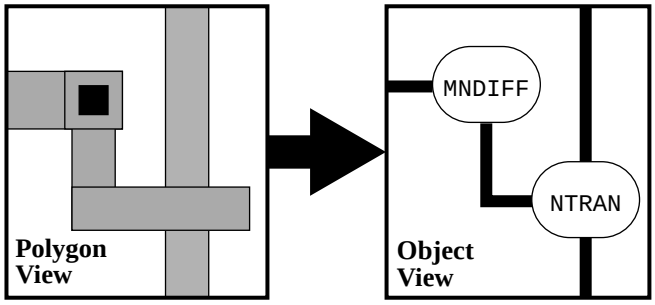


Figure 2: Object Recognition Process

e. With wires now represented as path-based objects, a compactor [10] is employed to move the devices as close to each other as the new design rules will allow, while preserving the connectivity.

f. It is then a trivial matter to iterate over all the objects in the compacted view and extract their geometries to create new polygon data for fabrication in the new process.

1.3. The Forma Database

In Forma, a design consists of a *library* containing a collection of *cells*. A cell can have more than one *view* (layout, schematic, bounding box, etc.); the appropriate view is determined by the context in which it is used.

Each view holds its own collection of *elements*. Forma supplies a generic set of built-in element types (e.g., `RECTANGLE`, `POLYGON`, `PATH`, `TEXT`, `INSTANCE`), and also a programmable element type, called an *extensible element* [11], which can be configured using Forma's extension language, *Genie* [12], to create types specific to an IC process (e.g., `NTRAN`, `PTRAN`, `M1M2`).

Elements represent the devices and wires in a view. Unlike previous layout tools [1,2,3,4,7,8,9], Forma is not limited to a hard-coded element set. Instead, each view is associated with exactly one technology defining the types of elements that can be stored in it.

Elements are programmable objects. Element attributes that can be modified (e.g., the `width` of a `RECTANGLE`) belong to the `EDITABLE` category, but not all modifiable attributes are necessarily independent of each other. For example, we could also set the `lowerLeftCorner` attribute of a `RECTANGLE`, potentially affecting both its `length` and `width` attributes.

The `DEFINING` category allows element authors to supply an orthogonal subset of `EDITABLE` attributes that completely characterize an element. In other words, clients can cause an element to attain any achievable configuration simply by setting its `DEFINING` attributes.

2. Device Recognition

The issues surrounding the extraction of a netlist from mask artwork are well understood. [6,7,8,9] Device recognition goes a step further by replacing portions of the polygon layout with objects that represent the equivalent layout, but as a cohesive unit. Previous attempts at device recognition [13,14] assumed a fairly restrictive class of device types. The underlying flexibility in creating almost arbitrary elements in Forma, coupled with a small but powerful, general purpose recognition language enables devices from a wide range of fabrication process technologies to be recognized easily.

2.1. Overview of the Recognition Process

Device recognition is accomplished by programming the object recognition engine with a sequence of actions, enabling it to make educated guesses about what values to supply as `DEFINING` attributes that would most likely configure a particular element to match the geometries surrounding a given shape.

The overall recognition process has two phases:

1. Setup Phase—Describe, to the recognition engine, how to recognize geometries in a view as devices defined for a particular technology.

2. Recognition Phase—Invoke the (now programmed) recognition engine to process the Input view, recognize the devices, and place the correspondingly-configured technology elements in the Output view.

Initially, the recognition engine is invoked with the specified technology defining the available mask layers and element types. During the setup phase, the object recognition engine assimilates a series of instructions, creating an internal representation of how each device would be recognized. The final instruction is then to process the Input view to create the Output view.

2.2. Setup Phase

The first step in the setup phase is to describe how to use boolean mask operations (as in [7]) to synthesize key geometries (called *seed* shapes) on supplemental layers that identify potential devices to be recognized. For example, to identify all likely P-transistor gate regions in a CMOS process having mask layers `POLY` and `PDIFF`, we can synthesize a new layer with geometries that are the logical AND of geometries on those layers:

```
LAYERDEF ptran = POLY AND PDIFF
```

Next we describe a sequence of recognition blocks identifying devices that we hope to recognize. Each block associates an element in the technology with (1) a *seed* layer and (2) a list of recognition statements describing how best to configure this device relative to the current (seed) shape on the seed layer. For example, to recognize a path-based P-transistor (`TPC`) whose gate region is identified by `POLY` crossing `PDIFF`, we might create a recognition block as follows:

```
DEVICE TPC ptran
  ORIGIN = ZERO_POINT
  channel = PATH_OF SEED_SHAPE
  TRY
END
```

The block header associates element type `TPC` (made available by the current Forma technology) with the (derived) seed layer `ptran`. The first statement within the recognition block alters the origin of the element from the center of the bounding box of the seed shape (default) to the point value (0, 0). The only `DEFINING` attribute of this device is the `Path` value of its `channel`. The second statement applies the path recognition operation to the geometry of the current seed shape in order to "guess" that value. The third and final statement instructs the recognition engine to check (using boolean mask operations) to see if the geometries of the `TPC` element as currently configured are contained in the original layout. If so, the device will be declared recognized; otherwise it will not. (The recognition language is discussed further in [15].)

The order in which recognition blocks are specified is significant. For example, path-based elements capable of matching the geometries of bent-gate transistors are also capable of matching those of conventional ones (e.g., `TN`). Placing the recognition block for the simpler element ahead of the block for the more complex one may result in a more efficient representation of the device.

3. Exact Wire Recognition

Once all devices have been recognized, the remaining unrecognized geometry (still in polygonal form) is presumed to represent the wires that connect them. Our goal is to represent this geometry, instead, as path-based wire elements, each having a centerline and a width. A routing-based approach to synthesizing manhattan wires is presented in Section 4. Here we describe what we believe to be a new technique for recognizing all-angle wires.

3.1. Overview of Wire Recognition

As with object recognition, the first phase (setup) is to specify on which layers wire recognition is to occur. When specifying a layer we must also identify the element to use and the name of the element attribute for the engine to use to reconfigure the path. If the element is not specific to that layer, then the name of the element attribute indicating the layer must also be specified:

```
WIRE POLY "WIRE" peditable layer
```

This `WIRE` recognition statement programs the engine to look on layer `POLY` for paths (represented as polygons) using element `"WIRE"` with `Path` attribute `peditable` (short for "primary editable") and `Layer` attribute `layer`.

For each wire recognition statement (in the order specified)
  Configure element with current `Layer` (if needed).
  Create empty set of `Path`s recognized by the current statement.
  For each `Polygon` in the current `Layer` of the Unrecognized view
    Guess `Path` whose perimeter is the current `Polygon`.
    Convert `Path` to `Polygon` and compare with current one.
    If not same, continue with next `Polygon` on current `Layer`.
    Configure element with newly identified `Path`.
    Verify all geometry of element is contained in Support view.
    If not, continue with next `Polygon` on current `Layer`.
    Add copy of element to Recognized view.
  Subtract geometry of recognized elements from Unrecognized view.

Figure 3: High-Level Wire Recognition Algorithm

During the recognition phase, after device recognition is complete, the sequence of wire recognition commands is applied to the geometries remaining in the Unrecognized view. As described in Figure 3, the engine attempts to guess a path whose perimeter exactly matches that of the polygon (see Section 3.2). If successful, the specified wire element is configured and tested against the geometries in the Support (Recognized + Unrecognized) view. If the element is contained in the Support view, it is copied into the Recognized view and its geometry is subtracted from the Unrecognized view.

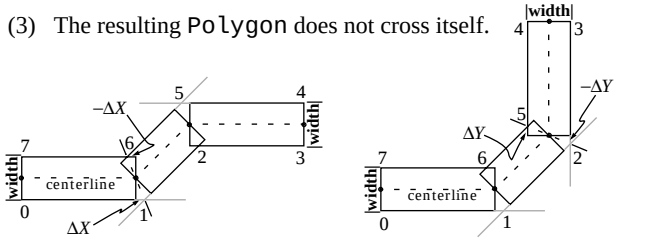
3.2. Path Recognition

An important subtask of recognizing wires and bent-gate transistors is to convert a given `Polygon` to a `Path` representing equivalent geometry if one exists. In our approach to path recognition, we make the following assumptions:

(1) The `Polygon` to be recognized was at one time represented as a `Path` with *even* `width` and its `centerline` *on grid*.

(2) The `Path` was converted to a `Polygon` by calculating the vertices of the perimeter using real coordinates and then independently rounding them to the closer integer (*grid*) coordinate.

(3) The resulting `Polygon` does not cross itself.



(a) Left Then Right Bend **(b) Two Consecutive Right Bends**

Figure 4: Converting a Path to a Polygon

Figure 4 shows two non-manhattan paths. Constructing the perimeter of either path amounts to identifying integral vertices 0 through 7. Using real numbers initially, we calculate parallel lines a half width to the left and right of each segment of the centerline. We can then round the points of intersection for consecutive segments to obtain the gridded approximation of the perimeter

An important observation is that each vertex on the centerline will always be the average of the two corresponding points of intersection if they exist. (A degenerate case occurs in Figure 4b when the width increases to the point where vertices 5 and 6 merge.) If an end is not known, it can be found through trial and error:

```
for (i = 0; i < N; ++i)
  for (j = 1; j < N/2; ++j)
    if (edge[i + j] is not approximately parallel to edge[i – j]) break;
    if (edge[i + j] is not in same direction as edge[i – j]) break;
  if (j > N/2) break; // i is probably the index of an "end" edge
```

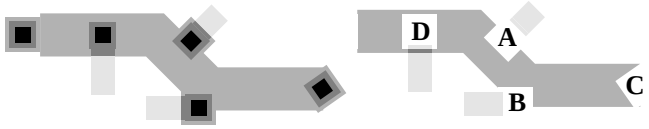
Once an edge corresponding to an end of the path is known, it is easy to walk the perimeter of the polygon, averaging corresponding vertices (e.g., 7&0, 6&1, 5&2, 4&3) to recover the original centerline. We then verify the result by converting the path back to a polygon and check that the two polygons are equivalent (modulo rotation of vertex numbers).

By convention, *edge[i]* connects *vertex[i]* to *vertex[i+1]*, and all arithmetic on edge and vertex indices is modulo the number of edges, *N*. We use the notion of *approximately* parallel because we cannot rely on the corresponding edges of a segment to be exactly parallel and on grid (for example, where the ΔY and ΔX of the middle segment in Figure 4b are relatively prime).

Rectangular polygons can be recognized as short-wide or long-narrow paths. Circuit designers along with CAD tools (such as a compactor) may be confused by an arbitrary orientation of the centerline. Edge weights *0 (not adjacent)*, *1 (partially adjacent)*, and *3 (fully adjacent)* with respect to recognized geometry (see Section 3.3) are used to influence the choice. The opposing edges with the greater combined weight are treated as ends. In the event of a tie, a long-narrow wire is preferred.

3.3. Path Reparation

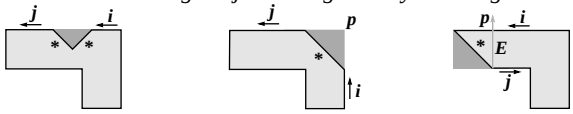
As device geometries are extracted from the unrecognized view, it is not uncommon for overlapping geometries to be lifted from wires resulting in a geometry that is not a simple path. The four common types of damage are illustrated in Figure 5: device **A** eclipses an edge of the wire, device **B** eclipses a corner of the wire, device **C** eclipses an entire end of the wire, and device **D** eclipses part of the interior of the wire.



(a) Before Devices are Recognized **(b) After Devices are Recognized**

Figure 5: Damage to Wire During Device Recognition

** edge adjacent to geometry in Recognized view*



Condition **A** Condition **B** Condition **C**

∃ edge indices *i* and *j* such that for a polygon with *N* edges:

A: 1. $(j - i + N) \bmod N > 2$
2. edge[*i*] and edge[*j*] are not touching recognized geometry
3. edge[*i* + 1] ... edge[*j* – 1] are touching recognized geometry
4. edge[*i*] is approximately parallel to edge[*j*]
5. edge[*i*] is approximately colinear with edge[*j*]
6. edge[*i*] has positive projection on edge[*j*]

B: 1. $(j - i + N) \bmod N > 1$
2. edge[*i*] and edge[*j*] are not touching recognized geometry
3. edge[*i* + 1] ... edge[*j* – 1] are touching recognized geometry
4. edge[*i*] is not approximately parallel to edge[*j*]
5. extension of edge[*i*] intersects negative extension of edge[*j*] at point *p*
6. point *p* is not part of either edge[*i]* or edge[*j]*

C: 1. $(j - i + N) \bmod N > 1$
2. edge[*i*] and edge[*j*] are not touching recognized geometry
3. edge[*i* + 1] ... edge[*j* – 1] are touching recognized geometry
4. edge[*i*] is approximately parallel to edge[*j*]
5. edge[*i*] is not approximately colinear with edge[*j*]
6. edge[*i*] has negative projection on edge[*j*]

Figure 6: Perimeter Damage Detection Criteria

In order to recognize a `Path`, it is first necessary to repair the wire geometry. The problem is formulated as follows: Given (1) a closed `Polygon` with $N > 2$ edges, and (2) an array of *weights* identifying whether each edge is *fully adjacent*, *partially adjacent*, or *not adjacent* to geometry in the Recognized view, modify the `Polygon` in a way that it can be recognized as a `Path` that is still contained in the Support view.

The procedure for repairing (perimeter) damage of types **A**, **B**, and **C** is to iterate over the edges of the polygon looking for conditions that indicate a repair is needed. Each of the three types of perimeter damage has a corresponding detection and replacement routine. Each detection routine takes the current polygon configuration, edge adjacency information, and an edge index, and returns the index of a second edge satisfying the condition or –1 if no such edge exists. If the condition is detected, the corresponding replacement routine is called with the two edge indices to repair the detected damage.

Figure 6 illustrates the conditions under which each type of perimeter damage will be detected for the specified polygon, edge adjacency, and edge indices *i* and *j*. The corresponding replacement actions are given in Figure 7.

A:	a. set edgeWeights[*i*] = PARTIALLY_ADJACENT b. remove vertex[*i* + 1] ... vertex[*j*]
B:	a. set edgeWeights[i] = PARTIALLY_ADJACENT b. set edgeWeights[j] = PARTIALLY_ADJACENT c. remove vertex[*i* + 1] ... vertex[*j* – 1] d. set vertex[*j*] to intersection of (extended) ray[*i*] and –ray[*j*]
C:	a. set edgeWeights[*i* + 1] = FULLY_ADJACENT b. remove vertex[*i* + 2] ... vertex[*j* – 1] c. set edge *E* = (point *p* where perpendicular to edge[*j*] at vertex[*j*] intersects edge[*i*], vertex[j]) c.' set edge *E* = (vertex[*i* + 1], point *p* where perpendicular to edge[*i*] at vertex[*i* + 1] intersects edge[*j*]) d. foreach *k* in { *i* + 1 ... *j* } translate edge *E* to its left until vertex[*k*] is not to the left of *E* e. set vertex[*i* + 1] = *E*[0] and vertex[*j*] = *E*[1]

Figure 7: Replacing Vertices to Repair Path

Prior to wire recognition, each polygon in the Unrecognized view with *holes* (damage type **D**) is converted to a simple polygon and a list of simple holes. Using boolean mask operations, each hole is compared with the corresponding layer in the Support view. Hole shapes *not* contained in the Support view (rare) are subtracted from the simple polygon

3.4. Path Partitioning

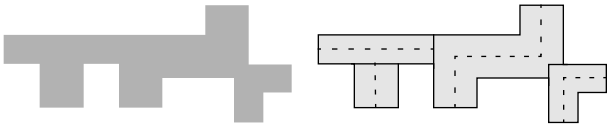
Polygons that result from intersecting wires (see Figure 8) cannot be recognized as a single path. In such cases, we must try to decompose that polygon into sub-polygons that can be recognized. This generalization of the exact wire recognition problem is to find a set of paths:

(1) whose geometry is contained in the Support view,

(2) that minimizes the area *not* covered by the original polygon,

(3) of minimal cardinality (i.e., number of distinct paths), and

(4) having minimal combined number of segments (or corners).



(a) Original Polygonal Wire (b) Optimal Path-Based Wires

Figure 8: Polygonal Wire Requiring Decomposition

Using dynamic programming [16] our plan is to try all reasonable partitions of the polygon and select the one that minimizes the cost criteria above. Figure 9 illustrates the top-level recognition procedure, `recognizePaths` and its associated (C++) dynamic programming data structure, `Cache`. The input to the procedure is an array of paths to hold the optimal solution, the original polygonal wire, the edge adjacency (weights) with respect to recognized geometry, and a technology-dictated minimum wire width. The original polygon is copied and repaired using the techniques described in section 3.3. The recursive algorithm `recognizePaths1` (described below) is then invoked to obtain (the index of) an optimal solution.

```
struct Cache {
   PathCache d_paths;              // current set of unique paths
   PolygonCache d_polygons;    // current set of unique sub-polygons
   Int32Matrix d_solutions;       // map each sub-polygon to best set of paths
   Float64Array d_scrapArea;    // left-over-area cost for each solution
   Int32Array d_numPaths;       // path-count cost for each solution
   Int32Array d_numSegments; // segment-count cost for each solution
   int d_minPathWidth;             // minimum width for path on this layer
};

double                                          // return total unrecognized area
recognizePaths(PathArray *result,    // loaded with best solution
                const Polygon &shape,                // original polygon
                const Int32Array &edgeWeights,   // w.r.t. adjacency
                int minPathWidth)      // avoid design rule violations.
{
   Cache data; // records solutions with cost for dynamic programming
   data.d_minPathWidth = minPathWidth; // initialize scalar value in struct
   Polygon shape2 = shape;
   Int32Array edgeWeights2 = edgeWeights;
   repairPath(&shape2, &edgeWeights2);
   int bestSolutionIndex = recognizePaths1(&data, shape2, edgeWeights2);
   Int32Array& pathIndices = data.d_solutions[bestSolutionIndex];
   for (int i = 0; i < pathIndices.numElements(); ++i)
      result->append(data.d_paths[pathIindices[i]]);
   return data.d_scrapArea[bestSolutionIndex];
}
```

Figure 9: High-Level Wire Partitioning Strategy

The `Cache` data structure contains two separate associative arrays to hold the unique sub-polygons and paths encountered during exhaustive search. The `Cache` also contains a mapping from each polygon to the set of paths that optimally represents it. Three parallel arrays are used to track the *cost* of each solution. The minimum path width is also stored for reference.

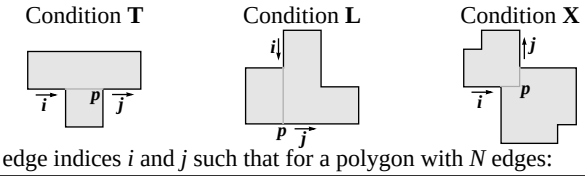
```
Assume that any damage to the polygon has already been repaired.
Lookup the current shape in the polygon cache; if found return its index.
If we can recognize the polygon as simple path
   If the path's width is below the minimum
      Add to cache solution with no paths and return its index.
   Add to cache solution with recognized path and return its index.

                  // No easy solution; we're forced to divide
                  // the polygon into two sub-polygons.

int bestSolution1 = –1, bestSolution2 = –1; // invalid index values
For each edge index, i, of the polygon
   For each edge index, j, of the polygon
      For each way to partition the polygon based on edges i and j
         Divide the problem into two problems: shape1 and shape2.
         Repair each sub-polygon (if needed).
         Call this procedure recursively on each subproblem.
         If the new combined solution improves on the current one
            Record the new indices in bestSolution1 and bestSolution2.
   If bestSolution1 now holds a valid index (≥ 0)
      If the combined left-over area of the solutions is zero
         If the combined number of segments is exactly 2
            break; // any solution with 2 segments leaving 0 area is best
If bestSolution1 holds a valid index
   Add to cache solution combining both best solutions and return its index.
```

Figure 10: Recursive Wire Partitioning Algorithm

When the algorithm returns to the top-level, the `Cache` (initially empty) contains all of the sub-polygons resulting from recursive partitioning. The integer value returned identifies the index of the original polygon in the polygon cache—the corresponding row in the solution matrix contains the indices in the path cache for the optimal path-based representation. The final 4 lines in Figure 9 extract the optimal solution and load the paths into the client-supplied `PathArray`, returning the amount of unrecognizable geometry as a `double`.

The recursive partitioning algorithm (Figure 10) exploits dynamic programming techniques to avoid solving any subproblem more than once. On entry, we *lookup* the index of polygon in the cache. If found, the optimal partition for this polygon has already been determined; we immediately return the index as the solution. Otherwise we try to recognize the polygon as a simple path. If successful, we enter the path in the path cache, add the polygon to the polygon cache, create a corresponding row in the solution matrix consisting of just the one index of the recognized path, and return the solution index. If this polygon cannot be recognized as a simple path, we must partition the polygon into two sub-polygons, call the function on each sub-polygon recursively, and merge the individual solutions to form a solution for this polygon. How we partition the polygon will determine the quality of our solution.

The approach taken here is to try *all* reasonable partitions and install the solution corresponding to the partition resulting in a path-based representation of minimal cost as defined above. Although the number of arbitrary polygonal partitions is essentially infinite, we have been able to characterize a discrete subset of useful partitions using 3 disjoint criteria: **T**, **L**, and **X**. (See Figure 11.)



∃ edge indices i and j such that for a polygon with N edges:

T: 1. $(j - i + N) \bmod N > 2$
2. edge[i] is approximately parallel to edge[j]
3. edge[i] is approximately colinear with edge[j]
4. edge[i] has positive projection on edge[j]
5. edge[i] has negative orientation w.r.t. edge[$i + 1$]
6. ray[i] intersects ray[$j - 1$] at point p
7. ray[i] does not intersect any other edge

L: 1. $(j - i + N) \bmod N > 1$
2. Either edge[i] is not approximately colinear with edge[j] or edge[i] has negative projection on edge[j]
3. edge[i] has negative orientation w.r.t. edge[$i + 1$]
3'. edge[$i - 1$] has negative orientation w.r.t. edge[i]
4. ray[i] intersects edge[j] or vertex[j]
4'. –ray[i] intersects edge[j] or vertex[j]
5. ray[i] does not intersect any other edge or vertex at point q such that $|q - \text{vertex}[i]| < |p - \text{vertex}[i]|$; if $|q - \text{vertex}[i]| == |p - \text{vertex}[i]|$ then q must not be part of an edge whose index falls between i and j
5'. –ray[i] does not intersect any other edge or vertex at point q' such that $|q' - \text{vertex}[i + 1]| < |p' - \text{vertex}[i + 1]|$; if $|q' - \text{vertex}[i + 1]| == |p' - \text{vertex}[i + 1]|$ then q' must not be part of an edge whose index falls between j and $i + 1$

X: 1. $(j - i + N) \bmod N > 1$
2. edge[i] is not approximately parallel to edge[j]
3. ray[i] intersects –ray[j] at point p
3'. –ray[i] intersects ray[j] at point p'
4. ray[i] does not intersect any other edge at point q such that $|q - \text{vertex}[i + 1]| \le |p - \text{vertex}[i + 1]|$
4'. –ray[i] does not intersect any other edge at point q' such that $|q' - \text{vertex}[i]| \le |p' - \text{vertex}[i]|$
5. –ray[j] does not intersect any other edge at point r such that $|r - \text{vertex}[j]| \le |p - \text{vertex}[j]|$
5'. ray[j] does not intersect any other edge at point r' such that $|r' - \text{vertex}[j + 1]| \le |p' - \text{vertex}[j + 1]|$

Figure 11: T, L and X Partitioning Criteria

By far the most common, the **T** partitioning criteria identifies wires that form T-like connections (though not necessarily at 90 degrees). If two wire edges, edge[i] and edge[j], are colinear and edge[i] can be extended through the interior of the polygon to meet edge[j], then the polygon can usefully be partitioned by severing the geometry on either side of the extended edge. The algorithm requires that we copy the polygon into `shape1` and `shape2`, and then use the **T** vertex replacement algorithm (Figure 12) to remove unwanted vertices corresponding to the severed geometry. Although partitioning does not make use of `edgeWeights` directly, it is necessary for these algorithms to maintain the edge adjacency so that each sub-polygon can be repaired (if necessary) before attempting to recognize it as a path.

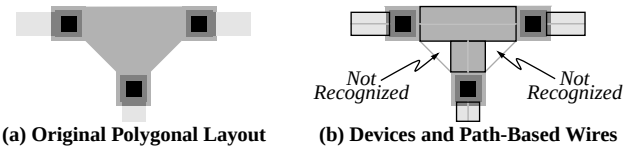
T: REPLACEMENT FOR SHAPE 1: (left of partition)
a1. if edgeWeights[i] == ALL_ADJACENT && edgeWeights[j] == ALL_ADJACENT
then leave edgeWeights[i] == ALL_ADJACENT
else set edgeWeights[i] = PART_ADJACENT
a2. Remove vertex[$i + 1$] ... vertex[j]
REPLACEMENT FOR SHAPE 2: (right of partition)
b1. set edgeWeights[j] = ALL_ADJACENT
b2. remove vertex[$j + 1$] ... vertex[i]

Figure 12: Vertex Replacement For T Criteria

The **L** criteria is particularly useful for recognizing multi-width wires and tapered power busses. The **X** criteria is used to recognize overlapping wires. The vertex replacement algorithms for **L** and **X** partitions are similar in concept to the one for **T** partitions, but considerably more involved; they can be found in [15,17]. Figure 8b shows how each of these criteria are applied to recognize the optimal path-based representation of a single complex wire.

4. Wire Synthesis

A significant fraction of viable, full custom designs were originally conceived entirely as polygons. Arbitrary polygonal wires do not necessarily lend themselves to "exact" path-based representation (see Figure 13a). Still, we would like to take advantage of IP invested in such designs. Here we present an alternative to exact recognition that synthesizes manhattan paths (suitable for compaction) that are (1) contained within the original layout and (2) preserve connectivity (see Figure 13b).



(a) Original Polygonal Layout **(b) Devices and Path-Based Wires**

Figure 13: Synthesizing Manhattan Paths From Polygons

Each polygon remaining in the unrecognized view after device recognition represents geometry that may be required to connect two nodes originally on the same net. We use Forma's connectivity extraction capability to infer net information for geometries in the Recognized view. For each unrecognized polygon, we create a routing region that is the union of the unrecognized polygon and all adjacent geometry in the Recognized view. The problem now reduces to finding, for a specified width, (1) a minimal set of paths (2) of minimal length (3) with minimal total number of segments (corners), that restores the adjacent geometry in the recognized view to a single net. The details of the solution employed can be found in [15,18].

5. Experimental Results

The object recognition engine is implemented in C++ as a hierarchical collection of 62 components (about 45,000 lines, 40% commentary). Each component has an associated test driver (35,000 lines, 38% commentary) to exercise and validate that functionality independently of other components (as recommended in [19]). Forma, also implemented in C++, is an order of magnitude larger.

Throughout the recognition process we exploit efficient elementary vector calculus techniques to avoid explicit use of expensive trigonometric functions [15]. Figure 14 illustrates the two steps of recognizing first devices and then wires for a simple CMOS (half) shift register cell. Initially (a) the recognized view is empty. After all devices are recognized, the geometry is divided among the two views (b). Finally after exact wire recognition, all but the right well region has been recognized as devices and path-based wires (c).

The algorithms described here have been applied both in isolation and together on a variety of hypothetical test cases. The recognition engine

has been used to convert entire CMOS standard cell libraries from GDSII polygons to circuit objects and path-based wires. Sample data included bent-gate transistors and non-manhattan geometries. Some cells were the output of object-based layout tools [2,3] converted back to polygonal form while others were conceived as polygons [1].

In all cases, CMOS transistors and contacts were recognized correctly 100% of the time. For output originally created in an object-based layout tool, wire recognition was perfect (except for notch filling and some well regions). It was also noticed that some tools have an off-by-one error in the way they generate the perimeter of a path on a gridded system. We were able to compensate by introducing an optional vertex tolerance and then rounding off-by-one centerline coordinates to the nearest multiple of 5.

For hand-generated polygonal layout, the results were still surprisingly good. Designers implementing 45-degree wires often follow a pattern consistent with path based recognition. We noticed, however, that the database resolution affected whether a digitized non-manhattan polygon was in fact an accurate representation of the rounded perimeter coordinates of a path. Selecting the appropriate scale enabled us to solve this problem.

The executable size on disk (including Forma) is 10.7 megabytes. Runtime for the example in Figure 18 was 24.1 CPU seconds (40 wall) running on a SPARC 20 model 51 workstation with 64 megabytes of memory. The device recognition performance is roughly linear in the number of devices (due in part to the geometric sorting of objects within Forma). Exhaustive search for some wires can occasionally become prohibitive despite our dynamic programming approach. It has therefore been necessary to provide limits on how much time is spent on each polygonal wire (1) looking for a solution that covers all the unrecognized material, and (2) once found, how much additional time we are willing to spend trying to improve on that solution.

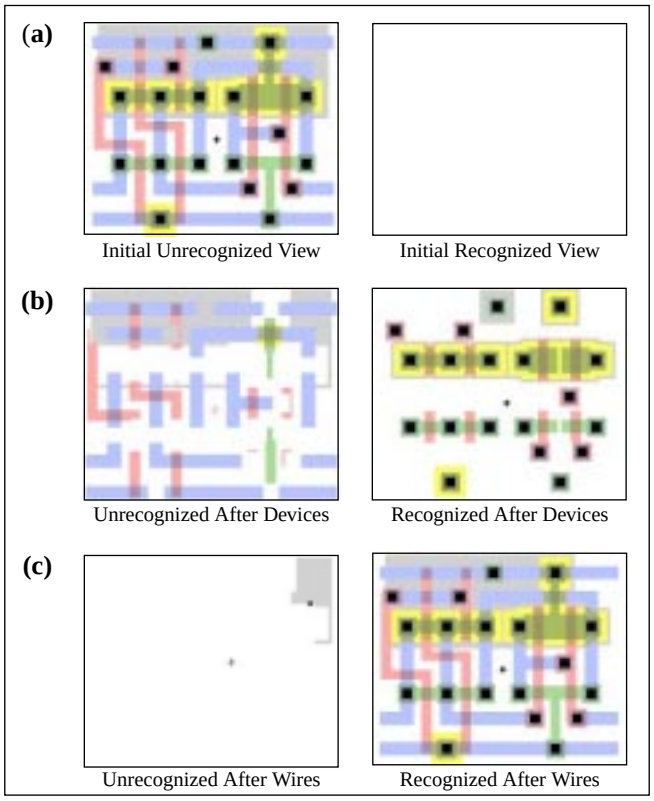


Figure 14: Result of Device and Wire Recognition

6. Conclusions

We have demonstrated a viable and general flow for extracting IP from existing polygonal IC layouts in older process technologies and retargeting them to new technologies. At the core of this flow is the ability to recognize an extensible set of devices and path-based wires as objects solely from their polygonal representations. In addition to a profoundly general device-recognition capability, the major contribution presented here is the ability to repair, partition and recognize complex all-angle wires efficiently. For polygonal layout originally conceived as objects, the recognition capability is nearly 100%. Wire synthesis effectively augments exact recognition where arbitrary polygons are used to interconnect devices.

7. Further Work

At present, we are not able to recognize devices that contain multiple seed shapes on a seed layer (such as arrayed contacts and bipolar devices). By embedding recognition commands in a much more powerful language (such as Genie [12]) we would be able to write sophisticated recognition rules based on the *context* in which the device is used. Also, more work is required to predict under what conditions exact recognition requires limiting, and how to improve performance under those conditions.

My thanks to Joe Cicchiello, Richard Eesley, Gad Gruenstein, and Alfred Schmidt for their direct contributions to this research.

8. References

[1] *Calma Reference Manual*, Calma Company, General Electric Corporation, Fairfield, CT, 1982.

[2] Matheson, T. G., C. Christensen, and M. R. Buric, *A software environment for building core microprocessor compilers*, Proc. ICCD, pp. 221–224, 1985.

[3] Draney, M., R., Method and apparatus for recording and rearranging representations of objects in a coordinate system, U.S. patent #4829446, 1989.

[4] *IC Station*, Mentor Graphics Corporation, Wilsonville, OR.

[5] Duh, J., T. G. Matheson, and E. Hepler, *Efficiently embedding expertise in high-density process-portable, standard cell generators*, Proc. IEEE Custom Integrated Circuit Conf. pp. 497–500, 1995.

[6] Bryant., R. E., *A switch-level model and simulator for MOS digital systems*, IEEE Transactions on Computers, Vol. C-33:160-177, 1984.

[7] *Dracula*, Cadence Design Systems, San Jose, CA.

[8] *Checkmate*, Mentor Graphics Corporation, Wilsonville, OR.

[9] *Caliber*, Mentor Graphics Corporation, Wilsonville, OR.

[10] Boyer, D. G., *Symbolic layout compaction review*, Proc. 25th Design Automation Conf., pp. 383–389, 1988.

[11] Matheson, T. G., *Integrated circuit design apparatus with extensible circuit elements*, U.S. patent pending.

[12] *The Genie programming language*, Mentor Graphics Corporation, Wilsonville, OR.

[13] *Polygon to Symbolic*, Cadence Design Systems, San Jose, CA.

[14] Dufourd, J., *The stickizer: a layout to symbolic converter*, Proc. ICCAD, pp. 534-537, 1989.

[15] Lakos, J. S., *Ph.D. dissertation*, Computer Science, Columbia University, New York, NY, expected 1997.

[16] Bellman, R. E., [1957], *Dynamic Programming*, Princeton University Press, Princeton, NJ.

[17] Lakos, J. S., *Exact (all-angle) wire path recognizer*, U.S. patent pending.

[18] Lakos, J. S., *Sub-virtual-grid point-to-point router*, U.S. patent pending.

[19] Lakos, J. S., *Large Scale C++ Software Design*, Addison Wesley, Reading, MA, 1996.