論文 91-28A-1-9

Sliced-Edge Trace 알고리듬을 이용한 계층적 Incremental DRC 시스템

(A Hierarchical and Incremental DRC System Using Sliced-Edge Trace Algorithm)

文寅鎬,* 金賢晶,** 吳晟煥,** 黃善泳*

(In Ho Moon, Hyun Jung Kim, Sung Hwan Oh, and Sun Young Hwang)

要 約

본 논문에서는 계층적인 incremental DRC를 수행하기 위한 효율적인 알고리듬을 제안하고, 이의 기능을 가지는 IC 레이아웃 에디터상의 incremental DRC 시스템의 구현에 관해 서술하고자 한다.

Sliced-Edge Trace 알고리듬은 하나의 다각형 또는 두개의 다각형간의 에지 교차에 의하여 분할된 에지들을 일정한 규칙에 따라 추적함으로써 resizing, Boolean operation등의 VLSI 패턴 조작을 수행하게 된다.

제안된 알고리듬은 레이아웃의 검증을 실시간에 수행할 수 있을 뿐 아니라 임의의 각도를 가진 다각형을 처리할 수 있으며, full-custom 방식의 모든 설계 규칙을 계층적으로 검사할 수 있다. 이 알고리듬은 UNIX 워크스테이션상에서 레이아웃 에디터의 개발에 사용되었으며 개발된 에디터는 VLSI 레이아웃을 생성하고 수정하기 위해 효과적으로 사용되고 있다.

Abstract

This paper presents an efficient algorithm for incremental and hierarchical design rule checking of VLSI layouts, and describes the implementation of a layout editor using the proposed algorithm. Tracing the sliced edges divided by the intersection of the edges either in a polygon or in two polygons (Sliced-Edge Trace), the algorithm performs VLSI pattern operations like resizing and other Boolean operations. The algorithm is not only fast enough to check the layouts of full-custom designs in real-time, but is general enough to be used for arbitrarily shaped polygons.

The proposed algorithm was employed in developing a layout editor on engineering workstations running UNIX. The editor has been successfully used for checking, generating and resizing of VLSI layouts.

I. 서 론

VLSI 설계 기술이 점점 발달함에 따라 최근 워크스테이션들의 연산 능력이 크게 향상되고 있다. 이에 따라 레이아웃 에디터등과 같이 워크스테이션상에서 대화식으로 수행하는 CAD 소프트웨어들의 경우 사용자와 컴퓨터간의 interaction 사이에 CPU의 많은 시간이 쉬는 상태로 남아있는 현상이 나타나게 되었

---

*正會員, 西江大學校 電子工學科 CAD 및 컴퓨터 시스템 硏究室

(CAD & Computer Systems Lab., Dept. of Elec. Eng., Sogang Univ.)

**正會員, 三星電子 半導體·富川 硏究所 CAD 개발팀

(CAD Develop. Team, Buchon Semiconductor R&D, Samsung Elec. Co.)

接受日字：1990年 7月 9日

다. 따라서, 컴퓨터의 효율적인 사용을 위하여 기존의 배취 모드로만 수행이 가능하였던 소프트웨어들을 대화식으로 수행하려는 시도가 진행되고 있다.

레이아웃 설계 검증을 위한 DRC(design rule check)의 경우 기존의 방법은 레이아웃이 완료된 후 설계오류가 있는지를 배취 모드로 수행하여 왔지만, 최근에는 레이아웃 작업을 수행하는 과정에서 DRC를 대화식으로 수행하는 레이아웃 에디터들이 등장하기 시작하였다.[11] 그러나, 대부분의 레이아웃 에디터들이 대화식으로 설계규칙(design rule)을 검사하는 과정에서 선폭(width)이나 선간격(space)등의 간단한 규칙만을 검사하거나, 또는 대화식으로 수행은 되지만 실제적으로 배취 모드로 동작하여 full-custom 방식의 레이아웃을 수행하는 데에는 부족한 점이 많이 있다.

이러한 문제점을 해결하여 full-custom 방식의 모든 설계 규칙을 계층적으로 검사할 수 있으며 대화식으로 실시간 동작이 가능하고, 또한 임의의 각도를 가진 모든 다각형(polygon)을 처리할 수 있는 효율적인 알고리듬을 제안한다. 제안된 알고리듬은 기존의 edge based 방식[2,10]과 polygon based 방식[1]을 동시에 적용한 대화식 DRC 수행에 적합한 새로운 알고리듬으로서, 하나의 다각형 또는 두개의 다각형간의 에지 교차에 의해 분할된 에지들을 일정한 규칙에 의해 추적함(sliced-edge trace)으로써 resizing, Boolean operation 등의 여러가지의 패턴 조작을 수행하게 된다.[15]

제안된 sliced-edge trace 알고리듬을 사용하여 incremental DRC 시스템을 구현 하였으며, IC 레이아웃 에디터상에서 대화식으로 계층적 DRC를 수행할 수 있도록 engineering 워크스테이션에서 개발하였다.

본 논문의 구성은 Ⅱ장에서 resizing 그리고 Boolean operation 등을 수행하기 위한 sliced-edge trace 알고리듬에 관해 기술하였고, Ⅲ장에서는 이 알고리듬을 사용하여 incremental DRC를 수행하기 위한 효율적인 자료 구조를 제시하였다. 그리고 Ⅳ장에서는 개발된 시스템의 DRC 수행에 관해 기술하였고, Ⅴ장에서는 sliced-edge trace를 기본으로 하여 개발된 DRC 알고리듬에 대해 기술하였다. Ⅵ장에서는 계층적인 DRC의 수행을 위한 알고리듬에 관해 기술하였다. Ⅶ장에서는 본 DRC 시스템과 타 DRC시스템과의 성능을 비교하였다.

Ⅱ. Sliced-Edge Trace 알고리듬

Sliced-edge trace 알고리듬은 다음의 조건들을 만족하는 레이아웃에 대해 적용되며 이를 만족하지 않는 레이아웃은 Ⅱ.3에 설명한 전처리 과정을 통해 변형한 후 적용된다.

1. 가정

본 알고리듬의 정상적인 수행을 위해서는 다음의 조건들을 만족하여야 한다.

a. 다각형의 self-intersection은 허용치 않는다.

b. 비정상적인 다각형(bad polygon 즉 re-entrant 또는 열려진 다각형등)은 허용치 않는다.

c. 연속적인 동일 버텍스(vertex)나 선폭 또는 자체의 선간격(notch)이 0인 다각형은 허용치 않는다.

d. 임의의 각도를 가진 다각형도 허용한다.

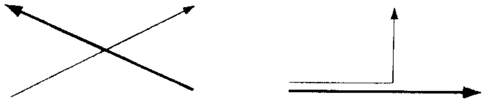
한편, sliced-edge trace 알고리듬에서는 다각형의 표현에 있어 버텍스 순서는 반시계 방향으로 정하고, 다각형의 첫번째 버텍스는 lower-left(y 좌표상의 가장 아래 위치하는 버텍스들중 x좌표상의 가장 왼쪽에 위치하는 버텍스)코너의 점으로 한다.

2. 교차(Intersection)의 처리

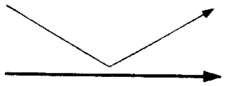
DRC를 위해 다각형을 이루는 에지들의 교차로부터 유효한 정보의 추출이 필요하다. 먼저 에지 교차 타입을 그림 1에서와 같이 다음의 3개의 타입으로 분류하며, 이는 Ⅱ.4의 pattern operation시 추적 과정의 다각형과 에지를 결정하는데 사용된다.

1)타입1 : 정상적인 에지와 에지간의 교차, 또는 인접 에지(abutting edge)가 존재하는 경우

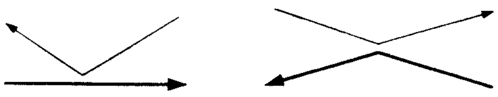
2)타입2 : 연속적인 두 에지 사이의 버텍스를 기준으로 다른 에지 또는 버텍스와, 같은 방향성을 가지고 인접되어 있는 경우



(a) Type 1



(b) Type 2



(c) Type 3

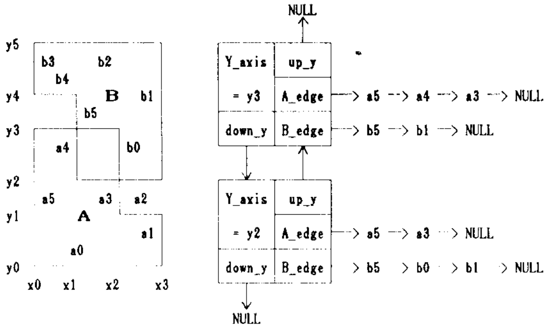
그림 1. 교차 타입의 종류

Fig. 1. Types of intersection.

3) **타입3 : 타입2와 동일한 경우이나 에지의 방향성이 다른 경우, 이 경우는 두번의 교차가 일어난 것으로 간주함.**[9]

(1) 다각형의 교차

다각형의 교차를 구하는 것은 결국 두 다각형의 에지들간의 교차를 구하는 것과 같다. 그러나, 이것을 모든 다각형에 적용한다면 시간적인 낭비가 심하기 때문에, 두 다각형의 겹쳐진 부분에 걸쳐 있는 에지들에 대해서만 그림 2 에서와 같이 sorted vertex link를 이용하여 효율적으로 다각형 교차를 구하게된다. Sorted vertex link는 두 다각형의 bounding box가 겹치는 영역에 걸치는 에지들의 버텍스들을 y 및 x축에 대하여 소트를 수행한 후 그 결과를 linked list로 가지는 것을 말한다. 그림 2(a)는 두개의 다각형(A와B)의 교차를 나타내며, 그림 2(b)는 그림 2(a)에 해당하는 sorted vertex link를 나타낸다. 이때, 두 다각형의 교차가 일어나는 영역은 y2에서 y3 그리고 x 0에서 x3가 된다. 또한, 각각의 y좌표는 두개의 link field를 가지며 각각의 link field는 이 영역내에 걸치는 각 다각형의 에지들을 버텍스들의 linked list로 가지게 된다.



(a) (b)

그림 2. Sorted vertex link의 예
(a) 다각형의 교차 (b) sorted vertex link

Fig. 2. An example of sorted vertex link.
(a) polygon intersection,
(b) sorted vertex link.

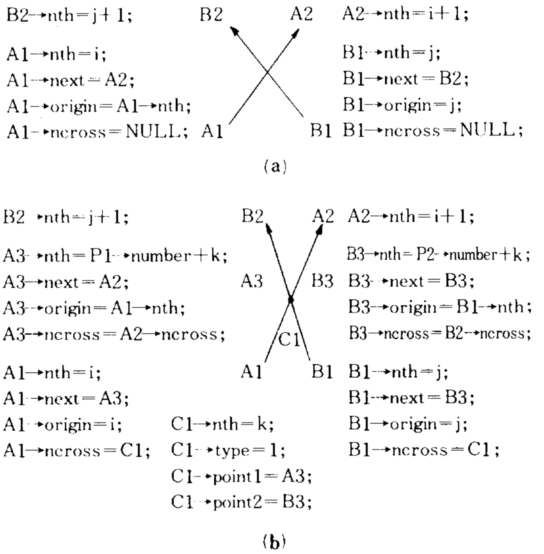
(2) 교차에 의한 에지의 분할

에지와 에지가 교차함에 의해 두개의 에지로부터 다음의 분할 과정에 의해 교차타입에 따라 두개 이상의 에지로 분할되어 진다. 이의 교차 전후의 자료 구조를 나타내기 위해 다음과 같이 point, cross 그리고 polygon의 자료 구조 변수를 정의하여 사용한다.

```
typedef struct  _point {
  int           x, y;        /*coordinate of vertex */
  short         nth;         /*nth vertex */
  short         origin;      /*original vertex number */
  struct _cross  *ncross;    /*intersecting point */
  struct _point  *next;      /*next vertex */
} POINT;
typedef  struct  _cross {
  short          nth;                  /*nth intersection */
  short          type;                 /*intersection type */
  struct _point  *point1, *point2;     /*vertex numbers of 1st
                                          and 2nd polygons at
                                          intersection point */
} CROSS;
typedef struct  _polygon {
  short          number;   /*number of vertices */
  int            bbox[4];  /*bound  box of polygon */
  struct _point  *start;   /*point to first vertex */
} POLYGON;
poygon *P1, *P2;           /*first and second polygons*/
point  *A1, *A2, *A3;      /*vertices of first polygon*/
point  *B1, *B2, *B3;      /*vertices of second polygon*/
cross  *C1,                /*cross point btw 1st & 2nd
                              polygons*/
```

이 자료 구조들을 사용하여 두 에지간의 교차를 처리하는 과정은 그림 3 에 나타내었다.

교차전의 자료 구조에서는 버텍스 A1은 버텍스 A2를 가리키고 버텍스 B1은 버텍스 B2를 가리키고 있다. 교차 후의 자료 구조에서는 교차에 의해 새로운 버텍스인 A3와 버텍스 B3가 생성되고 또한 교차



```
B2→nth=j+1;                    A2→nth=i+1;
A1→nth=i;                      B1→nth=j;
A1→next=A2;                    B1→next=B2;
A1→origin=A1→nth;              B1→origin=j;
A1→ncross=NULL;                B1→ncross=NULL;
```

(a)

```
B2→nth=j+1;                    A2→nth=i+1;
A3→nth=P1→number+k;            B3→nth=P2→number+k;
A3→next=A2;                    B3→next=B3;
A3→origin=A1→nth;              B3→origin=B1→nth;
A3→ncross=A2→ncross;           B3→ncross=B2→ncross;
A1→nth=i;                      B1→nth=j;
A1→next=A3;                    B1→next=B3;
A1→origin=i;     C1→nth=k;     B1→origin=j;
A1→ncross=C1;    C1→type=1;    B1→ncross=C1;
                 C1→point1=A3;
                 C1→point2=B3;
```

(b)

그림 3. 교차 전후의 자료 구조

Fig. 3. Data structures before/after intersection.

를 나타내는 struct _cross의 C1이 생성된다.

3. 전처리 과정 (Preprocessing)

본 알고리듬의 가정을 만족시키기 위해 모든 다각형 데이타의 입력시 반드시 전처리 과정을 거치게된다.

(1) 버텍스 Reordering

가정 c를 만족시키기 위해 다음의 과정을 수행한다.

[단계 1] 다각형의 버텍스 순서가 시계 방향인지 또는 반시계 방향인지를 결정한다.

[단계 2] 다각형의 lower-left 버텍스를 찾아 다각형의 첫번째 버텍스로 만든다.

[단계 3] 버텍스의 순서를 반시계 방향으로 만든다.

[단계 4] 연속적인 동일 버텍스나 연속적인 에지들의 선폭이나 선간격이 0인 에지들을 제거한다.

(2) 비정상적인 다각형 검사 및 Self-ORing

가정 a와 b를 만족시키기 위해 이 과정을 수행하며, Self-ORing은 그림 4 에서와 같이 한 다각형 내에서 겹치는 영역(overlap area)이 있는 경우 도넛츠 형태의 다각형으로 변환시켜 주는것을 말한다. 또한, 이 과정을 수행하면서 자동적으로 re-entrant 다각형, 열려진 다각형등의 비정상적인 다각형에 관한 검사가 동시에 수행된다.

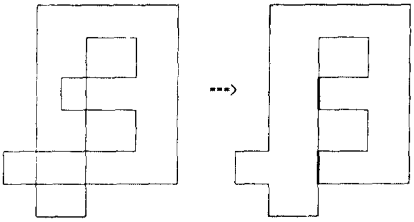


그림 4. Self-ORing의 예
Fig. 4. An example of Self-ORing operation.

(3) 비정상적인 다각형 검사 및 Self-ORing 알고리듬

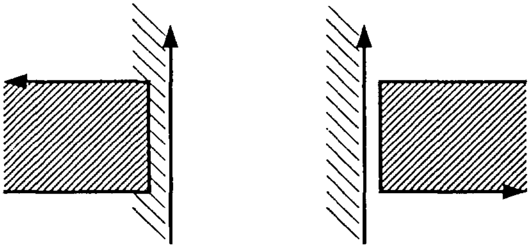
비정상적인 다각형을 검사함과 동시에 Self-ORing을 같이 수행하면서, 그림4에서와 같이 교차가 많이 일어난 에지를 중심으로 도넛츠 패턴의 내부 홀을 외부의 다각형 에지에 연결하기 위한 에지들의 겹침이 발생되도록 하기 위하여 각 에지마다 교차의 수에 해당하는 우선 순위를 주고, 이 우선 순위에 따라 추적을 진행하게 된다.

비정상적인 다각형의 검사 및 Self-ORing의 수행은 다음의 pseudo C 언어로 기술된 알고리듬에서와 같이 Self-ORing을 수행하는 과정에서 비정상적인 다각형의 검사가 동시에 수행된다.

```
int Self_ORing(polygon)
POLYGON *polygon;
{
  if(number of real_cross==0) {
    if(number of pseudo_cross>0)
      return(BAD_POLYGON)
    else
      return(GOOD_POLYGON);
  }
  /*check only for bounding edges from first edge in following
        while loop*/
  while(cur_cross_num>0) {
    if(real_cross exists on the reference edge) {
      if(direction of intersecting edge==entering_edge)
        return(BAD-POLYGON);
      else {
        priority of each edge-- ;
        cur_cross_num-- ;
      }
    }
    else
      if(reference edge==frist edge)
        break;
      else
        reference edge=next edge of reference edge;
  }
  if(cur_cross_num==0) return(BAD_POLYGON)
  /*check for remaining real_cross in following while loop */
  while(cur_cross_num>0) {
    order=ODD;
    for(each real_cross used in previous while_loop) {
      if(real_cross exsits on the reference edge) {
        if(cur_cross==starting cross)continue;
        reference edge=next edge which has higher priority;
        if(order==ODD) {
          if(direction of intersecting edge==entering_edge)
            delete intersecting edges from current cross point
            to previous cross point and make new edges to abut
            along reference edges;
          else
            return(BAD_POLYGON);
          order=EVEN;
        }
        else    /* order==EVEN */
          order=ODD;
      }
      else
        reference edge=next edge of current edge;
      if(cur_cross_num==0)break;
    } /* for */
  } /* while */
} /* Self_ORing */
```

위의 알고리듬에서 pseudo_cross는 그림5에서와같이 두 edge간에 실질적으로 겹쳐지는 영역이 전혀 없거나 또는 완전히 겹쳐지는 경우의 교차를 나타내며 두 에지가 하나의 다각형 내의 에지들이므로 교차가 일어나지 않은 것으로 간주한다. Real_cross는 pseudo_cross 이외의 일반적인 모든 교차를 나타내고 cur_cross_num은 현재 사용되지 않은 real_cross의 수를 나타낸다. 여기에서 pseudo_cross들을 real_cross로 취급하여도 동일한 결과를 얻게 되며, 단지 알고리듬상의 성능을 향상시키기 위하여 사용된다.

Priority는 각 edge상의 real_cross의 수를 나타내며, order는 알고리듬의 마지막 while loop상에서 추적을 시작한 교차 지점으로부터 홀수 또는 짝수번째로 현재의 교차지점을 만나는 것을 나타내고, reference edge는 추적을 진행하고 있는 에지 그리고 intersecting edge는 reference edge와 교차를 하는 에지를 나타낸다. 그림 6 에서와 같이 entering_edge 는 reference edge의 왼편으로 intersecting edge가 들어오는 경우의 에지를 나타내며, leaving_edge는 reference edge의 오른편으로 intersecting edge가 나가는 경우의 에지를 나타낸다.



(a) (b)

그림 5. Pseudo-cross의 예
(a) 겹침이 있는 경우
(b) 겹침이 없는 경우

Fig. 5. Examples of pseudo_cross.
(a) with overlap, (b) without overlap.

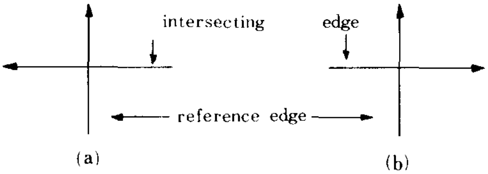


그림 6. 교차상에서의 에지의 방향성
(a) entering_edge (b) leaving_edge

Fig. 6. Edge direction at an intersection point.
(a) entering_edge, (b) leaving_edge.

4. Pattern Operation

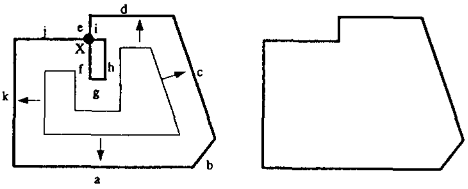
기본적인 pattern operation에는 크게 geometrical operation, Boolean operation, topological operation, checking operation의 네가지로 크게 나뉘어진다.[1]

(1) 하나의 다각형에 대한 수행 (Resizing)

Pattern operation 중 geometrical operation에 속하는 것으로서, 특히 resizing의 경우 DRC검사는 물론 새로운 레이아웃 레이어를 발생시키기 위한 기본적인 수행이다.[5]

2) Pre-resize 연산

그림 7(a)와 같이 resize하고자 하는 양만큼 모든 에지들을 expand (oversize의 경우) 또는 shrink (undersize 의 경우) 시킨다.



(a) (b)

그림 7. Pre-resize와 resize의 예
(a) Pre-resize 연산 (b) 확장된 다각형

Fig. 7. An example of pre-resize and resize.
(a) pre-resize operation,
(b) oversized polygon.

2) Oversize 연산

Pre-resize를 수행한 후, Self-ORing을 수행하면서 비정상적인 다각형으로 판명되는 교차 지점(그림 7(a)에서의 X점)에서 비정상적인 다각형을 형성하는 에지(그림 7(a)에서의 에지 f, g, h, i)들을 제거하여 출력한다.

3) Undersize 연산

Pre-resize를 수행한 후 다각형의 버텍스 순서가 시계 방향이면 NULL이 출력되며, 반시계 방향의 경우는 Self-ORing을 수행하면서 oversize에서와 마찬가지로 비정상 다각형으로 판명되는 교차 지점에서 비정상 다각형을 형성하는 에지들을 제거하여 두 개 이상의 다각형으로 나누어 출력한다.

(2) 두개의 다각형에 대한 수행

두개의 다각형에 대한 수행은 Boolean operation과 topological operation으로 크게 나눌 수 있고, 각각의 operation을 수행하는 과정을 설명한다.

1) Boolean Operation

두개의 다각형을 입력으로 하여 Boolean operation인 AND, OR, MINUS, XOR를 두 다각형에 적용하여 새로운 다각형을 만들어낸다.

[단계 1] 연산이 MINUS인 경우, subtractor 다각형(A-B의 경우B)의 버텍스 순서를 시계 방향으로 만든다.

[단계 2] Sorted vertex link를 사용하여 두 다각형간의 에지 교차를 구하고 에지들의 연결정보를 만든다.

[단계 3] 에지 교차가 없는 경우 한 다각형이 다른 다각형을 완전히 포함하는지 검사한다. 포함을 하는 경우 그림 8 과 같이 내부 다각형의 lower-left 버텍스로부터 바로 밑의 외부 다각형의 한 에지에 새로운 에지를 추가함으로써 간단히 결과를 구한다.

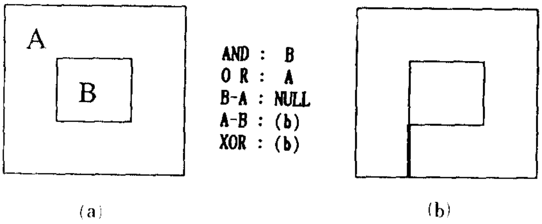


그림 8. 에지 교차가 없는 경우의 연산
Fig. 8. Operation in case of no edge intersection.

[단계 4] 임의의 에지 교차 지점을 선택한다. 단, XOR의 경우 두 다각형의 첫번째 버텍스 중 lower-left 버텍스로부터 가장 가까운 에지 교차지점을 선택한다.

[단계 5] 그림 9 와 같이 에지 교차 지점의 reference edge를 중심으로 두개의 각도(theta1, theta2)를 구한다. 이때, 두개의 각도가 동일한 경우는 reference edge를 따라 다음의 에지 교차 지점에서 다시 두개의 각도를 동일한 방법으로 구하여 사용한다. 여기에서 두 각도의 크기는 단계 6 의 표 1 에서와 같이 단지 두 각도간의 대소 비교에만 사용이 된다. 일반적으로 두 다각형의 에지간의 교차에서는 theta1이 180도가 되나, 두 다각형의 버텍스간의 교차 또는 에지와 버텍스간의 교차에서는 theta1이 180도가 아닌 임의의 각도를 가지게 된다.

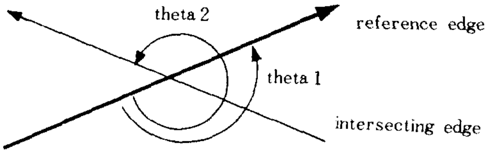


그림 9. 교차에 대한 두 각도의 계산
Fig. 9. Calculation of two angles for intersection.

[단계 6] Theta1, theta2와 연산(AND, OR, MINUS, XOR)에 따라 표 1 에서와 같이 추적을 시작할 다각형을 결정한다. XOR의 경우는 방향(forward, backward)까지 결정하며, backward의 에지 연결 정보를 추가로 가진다. 그러나, 표 1 은 단순히 추적을 시작하기 위한 출발점을 설정하는 것으로 반드시 표 1 에서와 같이 시작할 필요는 없지만 표 1 과 단계 7 의 그림10은 동시에 고려되어야만 된다.

표 1. 추적을 시작할 다각형과 방향의 결정
Table 1. Determination of starting polygon and direction for tracing.

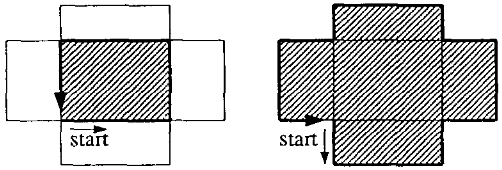
	theta1<theta2	theta1>theta2
AND	intersecting polygon	reference polygon
OR	reference polygon	intersecting polygon
MINUS	intersecting polygon	reference polygon
XOR	reference polygon backward	intersecting polygon forward

```
if (Intersection_Type==1) {
  if (current polygon==reference polygon) {
    polygon=intersecting polygon;
    edge=next edge of intersecting polygon;
  }
  else {
    polygon=reference polygon;
    edge=next edge of reference polygon;
  }
else/* Intersection_Type==2 || 3  */
    edge=next edge of the current polygon;
```

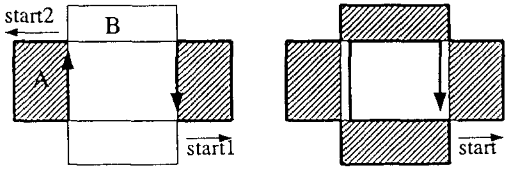
그림10. 추적 과정의 다각형과 에지의 결정
Fig. 10. Determination of the next polygon and edge during tracing.

[단계 7] 모든 에지 교차 지점을 통과하여 단계 4 에서 선택된 에지 교차 지점으로 되돌아 올 때까지 그림10에서와 같이 교차 타입에 따라 추적이 진행중인 다각형의 에지에서 다음 에지로 계속 진행할 것인지 또는 다른 다각형의 에지로 옮겨서 진행할 것인지를 결정한다. 표 1 과 그림10은 개념적으로 가장 단순한 경우만을 고려한 것이며, 실제의 구현 과정에서는 이보다 훨씬 많은 rule 들로 구성되어 있다. 또한, degenerate(에지 또는 버텍스의 겹침등)의 경우, 즉, theta1과 theta2가 같은 경우는 무조건 현재의 추적중인 에지를 따라 가다가 다음의 교차 지점에서 theta1과 theta2를 다시 구하여 추적을 진행하게 된다.

[단계 8] Sliced-edge trace로부터 얻은 다각형에 대해 버텍스 reordering을 수행한 후 다각형을 출력한다. 만일, 통과하지 않은 에지교차 지점이 더이상 없는 경우는 추적을 종료한다. 남아있는 경우는 단계 4 에서부터 단계 8 까지 반복적으로 수행한다. 단, XOR 의 경우는 모든 에지 교차 지점을 두번씩 통과하여야 한다. 그림11은 sliced-edge trace의 추적 과정을 나타낸다.



(a) AND (b) OR



(c) MINUS (A-B) (d) XOR

그림11. Sliced-edge trace의 예
Fig. 11. An example of sliced-edge trace.

2) Topological Operation

Topological operation은 두개 이상의 다각형들의 상호 위치 관계를 가지고 다음과 같은 조건들을 만족하는 다각형들을 선택하는 것을 말한다.

a. Touch, Not-touch에 의한 선택

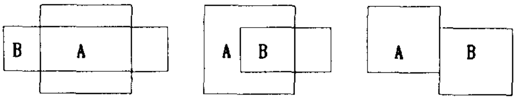
임의의 다각형을 선택하는 과정에서 다른 특정 다각형들과 도형상으로 붙었는지 또는 떨어져 있는지의 관계로써 선택함.

b. Inside, Outside에 의한 선택

임의의 다각형을 선택하는 과정에서 다른 특정 다각형들의 내부에 있는지 또는 외부에 있는지의 관계로써 선택함.

c. Cut. Abut에 의한 선택

임의의 다각형을 선택하는 과정에서 다른 특정 다각형들과 touch 상태에 있을때 겹쳐진 영역이 있는 경우(cut) 또는 없는 경우(abut)의 관계로써 선택함.



(a) (b) (c)

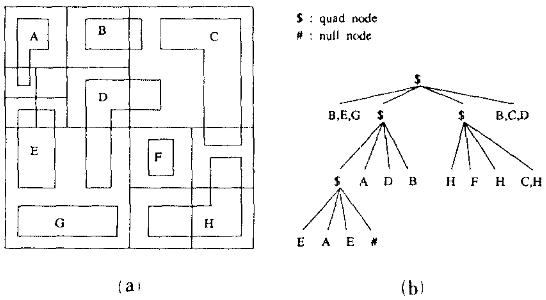
그림12. Cut과 abut의 예
(a), (b) cut의 경우 (c) abut의 경우
Fig. 12. Examples of cut and abut.
(a), (b) cut case, (c) abut case.

Ⅲ. Incremental DRC를 위한 자료 구조

VLSI의 방대한 데이타를 가지고 대화식으로 incremental DRC를 수행하기 위해서는 빠른 점 및 영역 검색(point & region search), 그리고 삽입등의 수행이 필수적이다. 본 incremental DRC 시스템에서는 효율적인 영역 검색을 위해 multiple storage quad tree[3,6]의 형태에 radix hash tree의 개념을 포함시킨 multiple storage radix hash tree[4]를 사용하였다. Incremental DRC의 수행은 레이아웃의 수정시 수정된 다각형을 중심으로 그 다각형에 대한 설계 규칙만큼 다각형의 bounding box로부터 확장시킨 영역내의 다른 다각형들에 대해서만 DRC를 수행하게 된다.

1. Multiple Storage Quad Tree

하나의 bucket(또는 sector)에 존재하는 object 의 수가 임계 수(threshold number)보다 많아지면 이 bucket은 네개의 sub-bucket으로 나뉘어지고, 각 object는 해당 bucket에 중복되어 할당된다(그림13). 이때 중복 할당의 경우 각 다각형의 bounding box를 이



(a)　　(b)

그림13. Multiple storage quad tree의 예
(a) 레이아웃의 예　(b) (a)에 대한 quad tree

Fig. 13. An example of multiple storage quad tree.
(a) a layout example,
(b) quad tree of (a).

용하여 할당하는 것이 아니라, 각 다각형을 여러개의 trapezoid로 분해(decomposition)하여 이 trapezoid들의 bounding box를 사용하여 해당 bucket에 할당하게 된다.[7]

Sub-bucket으로 분할하는 과정에서 트리의 깊이가 너무 커지는 것을 방지하기 위해 최소 bucket의 크기를 결정하여야 하며, 효율적인 메모리의 사용을 위하여 하나의 bucket 내의 다각형의 임계 수를 적절히 결정해 주어야 한다. 본 시스템에서는 경험적으로 레이아웃의 크기와 다각형의 갯수에 따른 적절한 임계수를 사용하고 있다.

2. Radix Hash Tree

효율적인 영역 탐색을 위하여 참고문헌 [4]에서 제안된 "multiple storage radix hash tree"의 방식으로부터 본 시스템의 quad tree 자료 구조에 맞도록 변형시킨 quad radix hash tree를 사용하였다. 각 bucket의 크기는 트리의 깊이에 따른 2의 멱수로 표시되며, 영역 탐색 과정에서는 주어진 탐색 영역의 크기로부터 bit-shift 연산이 사용된다.

3. 데이타 처리과정

Quad tree의 자료 구조의 영역 탐색 및 데이타 처리과정은 다음의 알고리듬과 같이 recursive하게 수행이 되며, 특히 데이타의 삽입시 데이타의 bounding box가 최상위 노드(top node)내에 포함되지 않는 경우는 상위 노드를 추가함과 동시에 최상위 노드의 위치를 변경해 주어야 한다. 이 과정을 pseudo C 언어로 기술하면 다음과 같다.

```
void(*application[ ])( )={region draw, point_search,
  region_search, insert_object,  delete_object};
void QuadTree(nd_ptr, window, func, object)
QuadNode *nd_ptr;      /*struct for quad node */
int window[ ];         /*specified window */
void (*func)( );       /*a function of application[ ] */
OBJECT *object;        /*specified object */
{
  if(func= =insert_object && nd_ptr= =top_node &&
    nd_ptr → window is bigger than window) {
    build_super_sector(object);
      /*makes a mother tree of nd_ptr and
         attach three null_subtrees of mother node */
    top_node=nd_ptr;
    return;
  }
  if(nd_ptr → num_of_object>=0)
    (* func)(nd_ptr, window, func, object);
  else
    for(each sub_tree)
           (*func)(nd_ptr, window, func, object);
}
void insert_object(nd_ptr, window, func, object)
{
  if(nd_ptr → num_of_object>THRESHOLD &&
    nd_ptr → window>MIN_SECTOR_WIDTH)
    build_sub_sector(nd_ptr); /* makes four subtrees */
  else
    add_object(nd_ptr, object);
        /* adds the object to linked_list of Quad_Node */
}
```

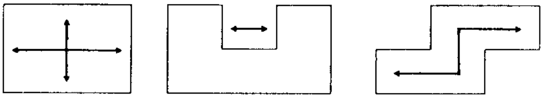
Ⅳ. DRC 수행

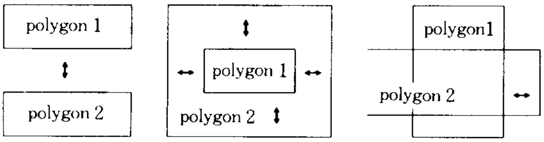
1. DRC 검사

DRC 검사에 있어서는 그림14에서와 같이 하나의 다각형에 대해서 수행되는 선폭, 선간격(notch), 면적, 각도 그리고 경로 길이(path length)등이 있으며, 두개의 다각형에 대해서 수행되는 선간격(space), 포함(enclosure), 여분(margin 또는 extension), 상호여분(co-margin), 그리고 교차(intersect)등의 검사가 가능하다. 이러한 DRC 검사는 사용자가 지정하는 기술정보를 입력으로 rule table을 작성하여 수행되며, 대화식으로 rule의 수정이 가능하다.

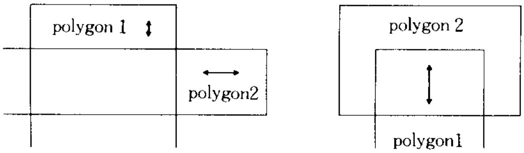
2. DRC 수행 방법

DRC를 수행하는 방법으로는, 레이아웃 데이타의 수정이 있을때마다 즉시 수정된 다각형에 대해서만 DRC를 수행하는 "real-time incremental DRC," 레이아웃 데이타의 수정이 있을 때마다 수정된 다각형의 고유 번호만을 스택에 저장하였다가 사용자가 수


(a) width (b) space (notch) (c) path length



(d) space (e) enclosure (overlap) (f) margin



(g) distance (h) intersect

그림14. 검사가 가능한 설계 규칙들의 예
Fig. 14. Examples of design rules which can be checked.

정된 다각형들에 대해 DRC를 수행하고자 할 때 스택에 있는 다각형들에 대해서만 DRC를 수행하는 “interactive DRC,” 사용자가 지정하는 특정 영역내에서 각 다각형들의 DRC flag가 체크되지 않은 다각형들에 대해서만 DRC를 수행하는 “window DRC,” 사용자가 지정하는 특정 영역내의 모든 다각형들에 대해 DRC flag에 무관하게 DRC를 다시 수행하는 “window DRC,” 사용자가 지정하는 특정 셀 내의 모든 다각형들에 대해 DRC를 수행하는 “cell DRC,” 그리고 배취 모드로 특정 셀에 대해 DRC를 수행하는 “batch DRC”등이 있다. 이들의 수행은 사용자가 상황에 따라 선택하여 사용이 가능하며, 대화식으로 변경이 가능하다. 이들의 수행과정을 pseudo C 언어로 나타내면 다음과 같다.

```
check_DRC (object)
POLYGON *object;        /*polygon to be checked */
{
  /*DRC_function_list:linked list of DRC functions */
  /* cur_fnt:current DRC_function_list */
  /* DRC_rule_table:point to DRC_function_list for each
  layer */
  if (object->DRC_flag= =NOT-CEHCKED || MODIFIED || ERROR) {
    switch (DRC_mode) {
    case REAL_TIME:
      cur_fnt =DRC_rule_table[object→layer]→DRC_function_list;
      while (cur_fnt ! =NULL) {
        (*cur_fnt→function) (object, cur_fnt→rule);
        cur_fnt=cur_fnt→next;
      }
      break;
    case INTERACTIVE:
      push_stack (object);
      break;
    }
  }
}
```

3. 레이어 발생 (Layer Generation)

Resizing, Boolean operation, 그리고 topological operation등을 사용하여 하나 또는 두 레이어의 전체 또는 부분적인 데이타에 대해 출력 레이어를 지정하여 새로운 데이타를 발생할 수 있으며 대화식으로 수행이 가능하다.

4. 레이아웃 에디팅

하나의 셀을 수정하고 있는 도중에 reference되어 있는 다른 셀 (instantiation)의 수정이 가능하며, 또한 “edit-in-place”시에도 incremental DRC의 수행이 가능하여 계층적인 레이아웃 설계에 사용이 매우 용이하다. User interface로는 다양한 에디팅 명령어와 bind-key, 사용자 정의 마크로 키 기능, 빠른 interrupt, 편리한 데이타의 복구등의 다양한 기능들을 가지고 있다. 또한, 레이아웃 에디터의 입력 및 출력으로 사용하는 그래픽 데이타 포맷으로 CIF, Calma GDSII stream,[12] Silvar-Lisco 사의 FDR 포맷[13]이 가능하다.

V. DRC 알고리듬

1. Single Layer Check

(1) 선폭

선폭의 검사는 그림14(a)에서와 같이 다각형의 선폭을 검사하는 것으로서, 평행한 에지들에 대해서만 검사하는 경우와 이웃한 에지 이외의 모든 에지들에 대해 검사하는 경우가 있다. 선폭 검사의 pseudo C 코드는 다음과 같다.

```
Check_WIDTH (polygon, min_value)
POLYGON *polygon;
float min_value;
{
  Pre_Resize (min_value);     /*for oversize */
  if (is_clockwise (polygon))
    return (ERROR);
  Reorder_Vertex (polygon);
  if (self_intersection exists)
    return (ERROR);
  else
    return (NO_ERROR);
}
```

(2) 선간격 (Notch)

그림14(b)에서와 같이 하나의 다각형 자체의 선간격을 검사하는 것으로서, 이의 알고리듬은 다음과 같다.

```
Check_SPACE1 (polygon,  min_value)
{
  Pre_Resize (polygon,  min_value/2) ;   /*for oversize */
  Reorder_Vertex (polygon) ;
  if (self_intersection exists)
    return (ERROR) ;
  else
    ruturn (NO_ERROR) ;
}
```

(3) 면적

한 다각형의 면적이 주어진 범위내에 속하는지를 검사하는 것으로서, 다각형의 버텍스가 n개라고 가정하면 ($x[0]=x[n-1]=x[n]$, $y[0]=y[n-1]=y[n]$) 다음과 같이 간단히 다각형의 면적을 구할 수 있다.

```
Check_AREA (polygon,  range)
{
  for (i=0;i<n;i++)  {
    add_term +=x[i]*y[i+1];
    sub_term + =y[i]*x[i+1];
  }
  area= (add_term+ sub_term) /2. 0;
  if (area is belong to specified range)
    return (NO_ERROR)
  else
    return (ERROR) ;
}
```

(4) 경로 길이

그림15에서와 같이 다각형을 trapezoid로 분할하면서 각 trapezoid의 center line의 길이를 더함으로써 경로 길이를 구한다.

PATH_LENGTH = pl_1+pl_2+pl_3

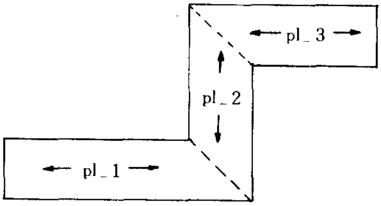


그림15. 경로 길이의 계산
Fig. 15. Calculation of path length.

2. Two Layer Check

(1) 선간격

그림14(d)에서와 같이 동일한 또는 서로 다른 레이어의 두 다각형간의 선간격을 검사하는 것으로서, 두 다각형간의 교차와는 무관하다. 이를 수행하는 알고리듬을 다음의 pseudo C 언어로 나타내면 다음과 같다.

```
Check_SPACE2 (polygon1,  polygon2,  min_value)
POLYGON *polygon1,  *polygon2;
float min_value;
{
  oversize (polygon1,  min_value) ;
  if (AND (polygon1,  polygon2) = =NULL)
    return (NO-ERROR) ;
  else {
    for (each ANDed polygon) {
      if (accurate_recheck (polygon) = =OK)
        return (NO ERROR)
      else
        return (ERROR) ;
    }
  }
}
```

(2) 포함 관계 (Enclosure)

그림14(e)에서와 같이 임의의 레이어의 다각형이 다른 레이어의 다각형 내에 완전히 속하는지를 검사하는 것으로서, 최소 거리가 지정되는 경우는 MARGIN 검사를 다시 수행하게 된다. 이 검사의 알고리듬은 다음과 같다.

```
Check_ENCLOSURE (polygon1, polygon2, min_value)
{
  if (overlapped (polygon1,  polygon2) ) {
    if (MINUS (polygon1,  polygon2) = =NULL)
      if (min_value is specified)
        return (Check_MARGIN (polygon1, polygon2, min_value) ) ;
      else
        return (NO_ERROR) ;
    else
      return (ERROR) ;
  }
  else
    return (ERROR) ;
}
```

(3) 여분 (Margin, Extension)

여분 검사는 그림14(f)에서와 같이 두 다각형이 교차할 때 하나의 다각형 (polygon1) 의 외부 에지 (outside edge) 로부터 교차하는 다각형 (polygon2) 의 내부 에지 (inside edge) 의 거리를 검사하는 것과, 그림14(e)

에서와 같이 하나의 다각형이 다른 다각형의 내부에 존재할 때 내부 다각형의 외부 에지로부터 외부 다각형의 내부 에지까지의 거리를 검사하는 것으로서 이를 수행하는 알고리듬은 다음과 같다.

```
Check_MARGIN(polygon1, polygon2, min_value)
{
  search_intersection(polygon1, polygon2);
  if(! intersecting point exists){
    oversize(polygon1, min_value);
    if(MINUS(polygon1, polygon2)==NULL)
      return(NO_ERROR);
    else
      return(ERROR);
  }
  else {
    AND(polygon1, polygon2);
    for(each ANDed polygon){
      edge_expand(polygon, min_value,
        edge_mask=polygon1);
      /* expand edges which came from polygon1 by min_value */
      if(MINUS(polygon, polygon2)==NULL)
        return(NO_ERROR);
      else
        return(ERROR);
    }
  }
}
```

(4) 상호 여분(Co-margin)

여분 검사와 비슷하나 그림14(g)에서와 같이 상호의 내부 에지로부터 외부 에지까지의 거리를 검사하는 것이다. 개념적으로는 다음과 같이 두번의 여분 검사를 수행하나 실질적으로는 동시에 하나의 검사로 수행한다.

```
Check_COMARGIN(polygon1, polygon2, min_value)
{
  Check_MARGIN(polygon1, polygon2 , min_valve);
  Check_MARGIN(polygon2, polygon1 , min_value);
}
```

(5) 교차(Intersect)

그림14(h)에서와 같이 서로 다른 레이어의 두 다각형이 교차할 때 하나의 다각형의 내부 에지로부터 다른 다각형의 내부 에지까지의 거리를 검사하는 것으로서, 이 검사의 알고리듬은 다음과 같다.

```
Check_INTERSECT(polygon1, polygon2, min_value)
{
  make_clockwise(polygon2);
  AND(polygon1, polygon2);
  for(each ANDed polygon) {
    edge_expand(polygon, min_value, polygon2);
    if(MINUS(polygon, polygon1)==NULL)
      return(NO_ERROR);
    else
      return(ERROR);
  }
}
```

3. DRC 수행의 최적화

하나의 다각형에 대해 여러가지의 설계 규칙들을 검사하는 경우, 각 설계 규칙들을 독립적으로 수행하는 것이 아니라 검사하여야 할 설계 규칙들의 조합을 미리 살펴봄으로써 각 검사들간의 중복되는 작업을 최소화 할 수 있도록 각 설계 규칙 검사의 순서를 재배치함은 물론 한번의 작업내용을 여러 검사과정에서 같이 사용할 수 있도록 한다.

Ⅵ. 계층적(Hierarchical) DRC

본 DRC 시스템은 계층적 구조의 레이아웃 데이타에 대해서도 대화식으로 즉시 DRC를 수행하기 위하여 각 셀간의 계층 정보를 트리 구조로 유지하게 되며, 각 셀마다 instantiation시의 기준점과 좌표 변환 메트릭스(transformation matrix)를 가지고 있다. 이 계층 정보를 이용하여, 하나의 다각형에 대한 DRC의 수행시는 이 다각형의 bounding box로부터 DRC 규칙의 최소 거리들 중 최대의 거리만큼 확장시킨 검사영역이 임의의 셀의 bounding box와 영역 교차가 있는 경우, 이 다각형의 좌표들의 교차된 셀의 좌표 변환 메트릭스에 적용하여 이 셀내의 새로운 다각형으로 만든 후 셀내의 다른 다각형들과 DRC를 수행하게 되며, 이 과정은 recursive하게 수행된다. 한 셀의 instantiation시의 DRC 수행은 마찬가지로 셀의 bounding box로부터 최대의 DRC 규칙만큼 확장시킨 검사 영역과 교차되는 다각형들을 먼저 선택한 후 이 다각형들에 대해 위와 같은 방법으로 DRC를 수행하게 된다.

또한, 이 계층적 DRC를 배취 모드에서 수행하는 경우, 하나의 셀이 여러번 instantiation된 경우 이 셀에 대한 한번의 DRC 수행만으로 반복적인 동일 셀의 DRC 수행과정을 피할 수 있게 됨으로써 DRC 수행에 필요한 시간을 단축시킬 수 있다.[8] 계층적 DRC를 수행하는 알고리듬을 pseudo C 언어로 나타내면 다음과 같다.

```
Hierarchical_DRC(polygon)
POLYGON *polygon;
{
```

```
Find_Objects_ToBeChecked (polygon) ;
  /* finds objects in the window expanded from the
     bounding box of the polygon by maximum DRC value */
  for (each object) {
    if (object is a cell)
      make_transform_to_lower_cell (polygon) ;
        /* changes coordinates of vertices of polygon
           to the coordinates of object cell */
      Hierarchical_DRC (polygon) ;
        /* in this step, active_opened-cell is chnaged
           to the object cell */
      make_transform_to_higher_cell (polygon) ;
        /* changes coordinates to come back */
    }
    else
      check_DRC (object) ;
  }
}
```

계층적 DRC의 수행은 위의 알고리듬에서와 같이 검사하고자 하는 다각형의 bounding box로부터 최대의 DRC 규칙 거리만큼 확장시킨 새로운 영역내에 존재하는 object들을 찾아낸다. 각각의 이 object들에 대해 object가 다각형이 아닌 reference되어 있는 셀인 경우 검사할 다각형의 버텍스 좌표들은 object의 좌표 변화 메트릭스에 적용하여 이 셀상의 좌표로 변환한다. 이 과정을 recursive하게 수행하면서 object가 다각형일 때 DRC를 수행하게 된다. 검사가 수행된 후 원래의 다각형 좌표로 귀환하기 위해 역변환 메트릭스에 적용하게 된다.

Ⅶ. 실험결과

실시간으로 수행하는 incremental DRC 시스템의 성능을 평가하는 기준으로써는 수행속도와 동시에 기능을 고려하여야 한다. 개발된 incremental DRC 시스템(HINT)의 성능을 평가하기 위하여 Berkeley 대학에서 개발된 Magic과 Cadence사의 Edge[11]와의 기능 및 수행 속도의 비교를 표 2와 표 3에 제시하였다.

표 2에서는 incremental DRC 시스템들의 기능을 비교하였다. Magic은 임의의 각도를 가지는 다각형을 취급하지 못하며, Edge는 실시간 DRC를 수행할 수 없으며 선택적인 영역내의 다각형들에 대한 DRC를 수행할 수 없는 단점을 가지고 있으나, HINT 시스템은 임의의 각도를 취급함은 물론 실시간 DRC의 수행 및 선택적인 영역내의 다각형들에 대한 DRC의 수행도 가능하다.

표　2. Magic, Edge와 HINT와의 기능 비교

Table 2. Functionality comparison among Magic, Edge and HINT.

	angle	real-time DRC	selective window DRC
Edge	arbitrary	no	no
Magic	rectangle	yes	yes
HINT	arbitrary	yes	yes

표　3. Magic, Edge와 HINT와의 속도 비교

Table 3. Speed comparison among Magic, Edge and HINT.

cell	n805_datadec (630 polygons)	n805_sysgen (949 polygons)	timer (1239 polygons)
Edge	3.5 sec	4.0 sec	6.8 sec
Magic	2.1 sec	2.4 sec	4.2 sec
HINT	3.0 sec	2.9 sec	5.7 sec

(* elapsed time on SUN 4/40)

표 3에서는 실제의 설계에 사용된 셀에 대하여 배취 모드의 DRC 수행 속도를 비교하였다. Incremental DRC의 정확한 성능을 비교하기 위해서는 실시간 DRC의 수행속도를 비교하여야 되지만 실제의 정확한 측정이 어렵기 때문에 배취 모드로 측정하였다. DRC 수행의 속도 비교에는 선폭, 선간격, 면적, 여분, 상호 여분, 그리고 교차의 설계 규칙이 사용되었다. 한편, Magic이 임의의 각도를 가지는 다각형을 처리하지 못하는 문제점을 가지고 있어 테스트 데이타를 모두 rectangle의 다각형만 사용하였다. 표 3에 나타난 결과상으로는 Magic의 DRC 수행 속도가 가장 빠른 것으로 나타났다. 그 이유는 Magic 에서는 모든 다각형을 직사각형의 타일로 분해하여 사용하기 때문에 DRC의 수행이 간단한 연산에 의해 수행되기 때문이다. HINT 시스템은 임의의 각도를 가지는 다각형을 처리하는 알고리듬을 사용하여 다각형을 분해하지않은 상태로도 Magic의 속도와 거의 비슷한 성능을 가지며, 다각형을 분해하지 않음으로써 메모리의 사용을 줄일 수 있다는 장점을 갖게 된다.

표 2와 표 3에 나타난 바와 같이 sliced-edge trace 알고리듬을 사용한 HINT 시스템은 기능면에서 기존의 incremental DRC 시스템의 성능보다 우수하며, 속도면에서 Magic에는 조금 뒤지나 Edge보다는 우수하다는 것을 확인하였다. 현재, HINT 시스템은 DRC 수행의 속도 측면에서는 최적화가 되어있지 않은 상태이어서 최적화 과정이 구현된 후 Magic에 비교할 만한 수행 속도를 얻을 수 있을 것으로 기대된다.

Ⅷ. 결 론

본 논문에서는 incremental DRC의 수행에 적합한 sliced-edge trace 알고리듬을 제안하였으며, 이를 이용하여 incremental DRC의 기능을 가지는 매우 사용이 편리한 레이아웃 에디터를 SUN 워크스테이션상에서 개발하였다.[14]

Sliced-edge trace 알고리듬은 모든 형태의 임의의 각도를 가진 다각형들을 취급할 수 있어서 full-custom 방식의 VLSI 데이타(MOS 및 bipolar 소자)의 DRC, resizing, layer generation 등의 pattern operation에 매우 적합하다. Incremental DRC의 경우 선폭 및 선간격 검사는 물론 여분, 상호 여분, 포함, 교차등의 모든 설계 규칙의 계층적 검사가 대화식은 물론 실시간으로 수행되며, 오류 수정을 위한 편리한 환경을 제공한다. 또한 DRC 수행의 기능 및 속도 측면에서도 기존의 incremental DRC 시스템들에 비해 우수한 성능을 보였다. 이에 바탕을 둔 레이아웃 에디터는 다양한 에디팅 명령들과 편리한 user interface를 제공하며 "edit-in-place"를 실현함으로써 계층적인 설계 방식에 사용될 수 있다.

參 考 文 献

[1] T. Ohtshki, *Layout Design and Verification,* North-Holland, 1986.

[2] S.M. Trimberger, *An Introduction to CAD for VLSI,* Kluwer Academic Publisher, 1987.

[3] J. Berger and G. Mazare, "A Range Searching Algorithm Sub-system Used to Perform Efficient VLSI Design Checks," *in Proc. ICCAD,* pp. 408-411, Nov. 1986.

[4] Y.D. Fontayne and R.J. Bowman, "The Multiple Storage Radix Hash Tree: An Improved Region Query Data Structure," *in Proc. ICCAD,* pp. 302-305, Nov. 1987. .

[5] Y.S. Huang and S.P. Chan, "A Graph-Theoretic Approach to the IC Layout Resizing Problem," *IEEE Trans. on Circuits and Systems,* vol. CAS-27, no. 5, pp. 380-391, May 1980.

[6] G. Kedem, "The Quad-CIF Tree: A Data Structure for Hierarchical On-line Algorithm," *in Proc. 19th DAC,* pp. 352-357, June 1982.

[7] S. Nahar and S. Sahni, "Fast Algorithm for Polygon Decomposition," *IEEE Trans. on Computer-Aided Design,* vol. 7, no. 4, pp. 473-483, Apr. 1988.

[8] M.E. Newell and D.T. Fitzpatrick, "Exploitation of Hierarchy in Analysis of Integrated Circuit Artwork," *IEEE Trans. on Computer-Aided Design,* vol. CAD-1, no. 4, pp. 192-200, Oct. 1982.

[9] L.M. Patnaik, R.S. Shenoy, and D. Krishnan, "Set Theoretic Operation on Polygons using the Scan-grid Approach," *Computer-Aided Design,* vol. 18, no. 5, pp. 275-279, June 1986.

[10] S. Perry, S. Kalman, and D. Pilling, "Edge based layout Verification," *VLSI Systems Design,* vol. 6, no. 9, pp. 106-114, Sep. 1985.

[11] *Physical Design and Verification Reference Manual,* Cadence, 1988.

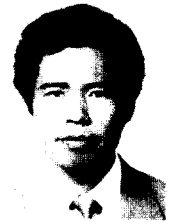
[12] *GDSII Reference Guide Release Manual,* CALMA Co., 1988.

[13] *NCA File Reference Manual,* NCA Corp., Aug. 1984.

[14] 오성환, 문인호, 이상훈, "ILED: 집적회로 레이아웃의 계층적 설계를 위한 대화식 그래픽 에디터," 대한전자공학회 추계종합학술대회 논문집, vol. 10, no. 1, 1987년 11월, pp. 729-733.

[15] 문인호, 김현정, 오성환, 황선영, "Incremental DRC를 위한 Sliced-Edge Trace 알고리듬," 1990년도 전자계산, 반도체 재료 및 부품, 씨에이디 합동학술발표회 논문집, 제 8 권, 제1호 1990년 5월, pp. 80-83.

著者紹介



文寅鎬 (正會員)
1962年 11月 15日生. 1985年 2月 한양대학교 전자공학과 졸업. 1990年 2月～현재 서강대학교 전자공학과 대학원 석사과정. 1984年12月～현재 삼성전자 반도체 연구소 CAD실 주임연구원. 주관심분야는 Design Verification, Layout Synthesys, CAD 시스템, Computer Architecture 등임.

◆

金賢晶 (正會員)
1964年 5月 10日生. 1986年 2月 성균관대학교 전자공학과 졸업. 1986年 1月～현재 삼성전자 반도체 연구소 CAD실 주임연구원. 주관심분야는 집적회로 설계 자동화 분야 등임.

吳晟煥 (正會員)
1959年 3月 25日生. 1982年 2月 서강대학교 전자공학과 졸업. 1985年 2月 연세대학원 전자공학과 졸업 공학석사 취득. 1985年 1月～현재 삼성전자 반도체 연구소 CAD실 선임연구원. 주관심분야는 집적회로 설계 자동화 분야 등임.

◆

黃善泳 (正會員) 第28卷 第1號 參照
현재 서강대학교 전자공학과 교수