AN EFFICIENT DATASTRUCTURE AND ALGORITHM FOR VLSI ARTWORK VERIFICATION

J. T. Fokkema*

Department of Mining Engineering

T. G. R. van Leuken

Department of Electrical Engineering

Delft University of Technology

Delft, the Netherlands

Abstract

A polygon can be represented by a set of vertical line segments. A new algorithm is presented which produces these line segments using a mechanism called the stateruler. Line segments belonging to the same polygon obtain the same polygon-number. The input of the algorithm is a collection of rectangles. Experience with an implementation of this algorithm shows fast execution on a large number of rectangles. The algorithm is of O(n) complexity, where n is the number of rectangles.

1. Introduction

The basic element in a layout data structure for VLSI circuits is often a box(rectangle) and the description of a mask layer consists of a large collection of unordered boxes. Then verification is a difficult task because many boxes overlap. The input for a design-rule checker or a circuit extractor should be a representation that is efficient and without redundancy. The best way to do this is with a polygon representation.

A polygon is uniquely described by a set of connected line segments. There are two approaches to accomplish this goal : the bit-map approach [5] and the method that confronts the edges of the boxes with other edges[2,3]. Both methods have their disadvantages, the first requires a large memory allocation and the second has an algorithmic complexity of O(n**m) where m>1.

The method proposed in this paper is also based on edge confrontation, however the area of confrontation is restricted. The implementation of the method results in an one-pass algorithm. The new aspect of this algorithm is that it does not search for information, all required information is present in the area of interaction.

* The basic ideas of this research were formed during the stay of J.T. Fokkema at Stanford University under the support of the Netherlands Organization for the advancement of Pure Research (Z.W.O.). The organization is gratefully acknowledged.

In our case a polygon is described by start and stop occurrences. We use the term start occurrence for a line segment when the interior of the polygon is situated on the right side. A stop occurrence denotes a line segment where the interior is located on the left side.

2. Computational model

We consider a particular mask area as a two-dimensional integer space D2, where the horizontal and vertical values are denoted by the integers x and y, respectively. Then a mask box is uniquely described by the following integers: xl, the left horizontal value; xr the right horizontal value; yb, the bottom vertical value and yt, the top vertical value.

The mathematical model underlying the computational model for the generation of occurrences is the set-theoretic relation, which is a subset of the Cartesian product of a list of domains. A domain is a simple set of values. Then the Cartesian product of domains Dom(1)*Dom(2)* ... *Dom(n) is the set of all n tuples[6]:

$$< a[p](1), a[p](2), \ldots a[p](n) >,$$

such that a[p](1) is in Dom(1), a[p](2) is in Dom(2) and so on. The integer n is called the arity of the tuple, p denotes the element number of the tuple in the set A and a[p](2) denotes the second component of a[p].

The collection of mask boxes can be represented by the set B of tuples of arity four such that

$$B = \{b[p]: < xl, yb, yt, xr >\},$$

and Dom(i) = D1.

where D1 is the one-dimensional integer space. We call a tuple set A ordered if there is at least one component a[p](q) such that for all p:

$$a[p](q) < a[p+1](q).$$

If K and L are both subsets of the ordered set A, then we use the notation

K <- L,

for the insert of L in K, such that K maintains its order. In the following section, an algorithm for occurrence generation will be discussed based on interactions between tuple sets.

3. The algorithm for occurrence generation

To explain our algorithm, we introduce the following ordered tuple sets.

The set of events E.

E = { e[n] : <b[q](1), b[q](2), b[q](3), b[q](4)>,
e[n](1) < e[n+1](1) or
e[n](1) = e[n+1](1) -> e[n](2) <= e[n+1](2)},

and Dom(i) = D1.

It is noted that E has the same elements as B, however ordered with respect to xl and yb.

The set of states S.

S={s[j]:<bottom, top, duration, group>,
s[j](1) < s[j+1](1) and s[j](2) < s[j+1](2) },

with Dom(i) = D1.

The duration and the group are the state variables of the set S.

The set of occurrences OC.

OC = { oc[m] : <X value , bottom value ,
top value , occurrence type, group>,
oc[m](1) < oc[m+1](1) or oc[m](1) = oc[m+1](1) ->
oc[m](2) < oc[m+1](2) and oc[m](3) < oc[m+1](3)},

with Dom(1), Dom(2), Dom(3), Dom(6) = D1,
and Dom(4) = { start occurrence, stop occurrence}.

The set of groups G.

G = { g[r] : <r, number>,
g[r](1) < g[r+1](1)},

with Dom(i) = D1.

The set of numbers N.

N = { n[t] : <t, G_connect, group number>,
n[t](1) <= n[t+1](1) },

with Dom(i) = D1.

The sets S and OC are linked with the set G via the component group, which refers to a element number of G. The set G is linked with the set N via the component number, which refers to a element number in N. The component G_connect in the element definition of N denotes the number of elements of the set G that are connected to a particular element in N.

The algorithm we propose for the generation of occurrences strongly resembles the basic algorithm of a logic simulator. A particular box of E, say e[n], has to be confronted with the state of the mask at Xpresent where Xpresent equals e[n](1). The state of the mask for a particular Xpresent is given by the stateruler { Xpresent, S}. The stateruler is the vertical cross-section of the mask at Xpresent and the states s[j](3) indicate the x value where the fields s[j](2) - s[j](1) will terminate. Xpresent divides the mask area in a past and a future with respect to the x value. We notice that no information of the past is available, only information of the future is present in the stateruler. The stateruler allows a *two-dimensional extension in the future of the* one-dimensional stateruler. This is just the information needed for a correct insert of the event in the stateruler.

Let e[n] denote the event that has to be merged with the stateruler. e[n] partitions the set S of the stateruler in three subsets :

(i) S1, such that S1 = { S1[j] is in S and S1[j](2) < e[n](2) }

(ii) S2, such that S2 = { s2[j] is in S and (S2[j](1) or S2[j](2)) is in [e[n](2), e[n](3)]}, where [... , ...] denotes the closed interval.

(iii) S3, such that S3 = { S3[j] is in S and S3[j](1) > e[n](3)}

To determine the merge position of e[n] in S all elements in S1 have to be visited. Because the imposed order of E every member of S1, S1[j], for which S1[j](3) = Xpresent is a stop occurrence. By generating the stop occurrences S1 is updated. The subset S2 indicates the area in the stateruler where the start occurrences are generated. The resulting modified set S2 merged with the set S3 determines the reduced stateruler, needed for the evaluation of the next event e[n+1] if e[n+1](1) = Xpresent. If e[n+1](1) does not equal Xpresent, each element of the reduced stateruler is tested for stop occurrences and updated, in the same way as S1. Then the next Xpresent is determined by the minimum value of the X of the next event and the smallest state of the set S. If the next value of Xpresent equals the minimum state, only stop occurrences are generated for that particular Xpresent. In the other case the procedure described above is repeated.

Program I shows an implementation of the main algorithm. In the first block the input boxes are ordered and the initialization takes place. The next block realizes the stop- and start-occurrence generation by the subroutines STOP and START respectively. The first call of STOP evaluates the subset S1, the call of START performs the confrontation of the event e[n] with the set S2 and the second call of STOP produces the stop occurrences of the reduced set S or the set S itself in the case that Xpresent equals the minimum state. The insert of the dummy event <max, 0, 0, max> flushes the stateruler, hence generates the last stop occurrences. In the last block the occurrences are counted.

```
ProgramI. : Occurrence Determination.

Input:  The set of unordered boxes .
Output: The set of ordered occurrences .

Procedure OCCURRENCE(B) :
begin
 E <- Quicksort(B); Xpresent <- e[1](1);
 min <- e[1](4);n <- 1; j <- 1;max <- 0;
 key <- 1;
 while n<= last element number of E do
 begin
  max <- maximum(e[n](4),max);
  if n= last element number of E and key =1
  then begin
   E <- <max,0,0,max>;key <- 0
  end
  if e[n](1) = Xpresent
  then begin
   if n < last element number of E
   then begin
    STOP(j,{Xpresent,S},min, flag <- e[n](2),OC);
    START(j,{Xpresent,S},e[n], OC,G,N)
   end
   n <- n+1
  end
  else begin
   STOP(j,{Xpresent,S},min,flag<-S[last](2)+1,OC);
   Xpresent <- minimum(e[n](1), min);
   min <- e[n](4);j <- 1
  end
 end;
 COUNT(OC,G,N)
end

ProgramII. : Stop Occurrence Generation

Input: j,the actual element number of S;
{Xpresent,S}, the stateruler; min,the
smallest element of the set {S[1](3),
and OC the set of occurrences.
Result: S and OC are updated by generating
the stop occurrences; j and min are
updated by stepping through S.

Procedure STOP(j,<Xpresent,S>,min,flag,OC)
begin
 if S is empty
 then return;
 while S[j](2) < flag do
 begin
  if S[j](3) = Xpresent
  then begin
   OC <- <Xpresent,S[j](1),
   S[j](2), STOP OCCURRENCE,S[j](4)>;
   delete S[j] from S
  end
  else begin
   min <- minimum(min,S[j](3)); j <- j+1
  end
 end
end
```

Program II shows the implementation of the subroutine STOP. When S[j](3)=Xpresent a stop occurrence is scheduled. The group of the stop occurrence is the same as the group of the deleted element S[j]. If there is no stop occurrence the minimum state is adapted.

```
ProgramIII. : Start Occurrence Generation

Input: j,the actual element number of S;
{Xpresent,S}, the stateruler; e[n], the event n;
OC, the set of occurrences;G, the set of groups
and N the set of group numbers.
Result: S,OC,G and N are updated by generating
the start occurrences.

Procedure START(j,{Xpresent,S},e[n],OC,G,N)
begin
 if S is empty
 then begin
  OC <- <e[n](1),e[n](2),e[n](3), START OCCURRENCE,
  last element of G+1>;
  S <- <e[n](2),e[n](3),e[n](4), last element of G+1>
  G <- <last element of G+1, last element of N+1>;
  N <- <last element of N+1,1,0>;
  return
 end
 k <- j; empty the sets F, O and O';
 if S[k](1) < e[n](2)
 then begin
  F <- <S[k](1),e[n](2),S[k](3), S[k](4)>;
  if S[k](2) > e[n](2)
  then
    S[k](1) <- e[n](2);
  else
    delete S[k] from S
 end
 while S[k](2) <= e[n](3) do
 begin
  O <- <S[k](1),S[k](2),S[k](3) <- maximum(e[n](4),
  S[k](3)), S[k](4)>; delete S[k] from S; k <- k+1
 end
 if( S[k](1) <= e[n](3)
 then begin
  F <- <e[n](3),S[k](2),S[k](3), S[k](4)>
  if (S[k](1) != e[n](3) )
  then
   O <- <S[k](1),e[n](3),S[k](3)<-maximum(e[n](4),
     S[k](3)), S[k](4)>; delete S[k] from S
 end;
 O' <- Construct the complement set O' of O with
 respect to the interval [e[n](2),e[n](3)]; F<-O;
 for each o'[l] in O' do
 begin
  F <- o'[l] resulting in the element f[p];
  f[p](3) <- e[n](4); determine f[p](4) ;
  if OC[last-1](4) = START OCCURRENCE and
     OC[last-1](2) = f[p](1)
  then
   merge START OCCURRENCES ;
  else
   OC <- <Xpresent,f[p](1), f[p](2),
   START OCCURRENCE, f[p](4)>
 end
 for each f[p] in F such that f[p](1) = f[p-1](2)
 and f[p](3) = f[p-1](3) do
 begin
  f[p](1) <- f[p-1](1); delete f[p-1] from F
 end
 S <- F
end
```

In program III the implementation of the start occurrence generation is described. The subset S2 is determined and deleted from S. S2 is further partioned in the boundary set F and the

overlap set O (the true projection domain of the event on S). The states of the elements of the set O are modified according the overlap rule that the new element state is the maximum value of the old element state and the event state(the xr value). Next the complement set O' with respect to the event field e[n](3) - e[n](2) is determined. The elements in this set form the start occurrences and the state of all elements equal the xr value of the event. O is inserted in F and successively all elements of O' are inserted in F. Now the environment of the start occurrence is known and the group can be determined. Finally the set F is reduced so that no connected fields have the same state. The set F is inserted back into the set S. Figure 2 shows an event confrontation with the stateruler and the corresponding sets.

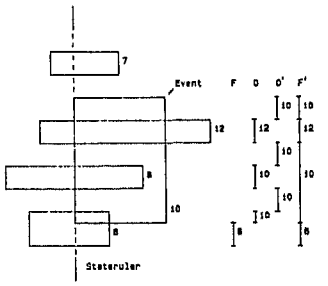

Figure 1. Event confrontation for a particular case with the corresponding sets F, O, O'and F. The numbers shown refer to the states of the fields.

ProgramIV. : Group Counting

```
Input:  OC,the set of occurrences; G ,the set of
        groups and N,the set of group numbers.
Result: the set OC is updated with respect to the
        group numbers.
procedure COUNT(OC,G,N)
begin
 n <- 1;
 for each OC[k] in OC do
 begin
  if N[G[OC[k](6)](2)](3) = 0
  then begin
   N[G[OC[k](6)](2)](3) <- n;
   OC[k](6) <- n; n <- n+1
  end
  else OC[k](6) <-N[G[OC[k](6)](2)](3)
 end
end
```

In program IV the counting of the occurrences is performed. Since the group structure is already embedded via the links of OC, G and N, the counting is a trivial matter.

4. Results

The algorithm described above is implemented in the program language C and is running under the UNIX * operating system on a VAX-11/750.

We investigated the linear-time complexity of the presented algorithm. We chose a random set of boxes as input to guarantee a constant design style and mask property. Table I shows the results for an increasing number of boxes and clearly illustrates the linearity of the time complexity.

TABLE 1. Computer runtime for a random set of boxes.

Number of random boxes	CPU time in sec.
100	1.8
1000	16.9
10000	188.8
50000	963
100000	2063

5. Conclusion

The new algorithm presented in this paper is efficient both in runtime and space. It generates a database suitable as input for the plotter, the pattern generator, the design rule checker and the circuit extractor.

We already have implemented a design rule checker, a circuit extractor and a program that performs Boolean operations on masks. Both programs use the occurrence sets as input. The working is also based on the stateruler formalism. Hence they also have a linear-time complexity. The programs only differ from the described algorithm in the nature and number of state variables of the fields in the stateruler. This implies that the confrontation of the events (now recruited from the set of occurrences) with the stateruler changes in every application.

References

[1] H.S. Baird: "Design of a Family of Algorithms for Large Scale Integrated Circuits Mask Artwork Analysis", M.S. Thesis, Dept. of Computer Science, Rutgers University, New Brunswick, New Jersey, May 1976.

[2] H.S. Baird: "Fast Algorithms for LSI Artwork Analysis", Proc. 14th DA Conf., New Orleans, June 1977, pp. 303-311.

[3] U. Lauther: "An O(nlogn) Algorithm for Boolean Mask Operations", Proc. 18th DA Conf., Nashville, Tenn., June 1981, pp. 555-562.

[4] P. Losleben, K. Thompson: "Topological Analysis for VLSI Circuits", Proc. 16th DA Conf., June 1979, San Diego, pp. 461-473.

[5] J.A. Wilmore: "A Hierarchical Bit Map Format for the Representation of IC Mask Data", Proc. 17th DA Conf., June 1980, Minneapolis, pp. 585-590.

[6] A.V. Aho, J.E. Hopcroft, S.D. Ullman: "The Design and Analysis of Computer Algorithms", Reading, MA: Addison-Wesley, 1974.

* Unix is a Trademark of Bell laboratories.