Designing a complementary design rule checker based on a binary balanced quad list quad tree

P.-Y. Hsiao
J.-T. Yan

Indexing terms: Computer-aided design, Design rule checker, VLSI layout, Balanced quad tree

Abstract: An efficient real time VLSI-CAD tool, the complementary design rule checker (CDRC), composed of one interactive phase and one batch phase is presented. It is a general geometrical design rule model which checks some of the layout constraints in the interative phase and the other constraints in the batch phase. Those classified constraints are disjointed, and each one of them should be checked only once either during the interactive or batch phase. This system and the embedded layout editor are designed on the basis of the binary balanced quad list quad tree (BBQLQT) and its region query functions. The BBQLQT is more efficient than the most recently published spatial data structure, the Weyten's quad list quad tree. One day, provided that the BBQLQT has being further improved, the performance of our system will be promoted without giving more additional design effort. This system is therefore proven to be fully modularised and independent of any of the other spatial data structures.

1 Introduction

For VLSI circuit design automation, it is important to guarantee the correctness of the efficient design of the specified layout. From the viewpoint of layout correctness, the designer must avoid having constraint errors in the layout, however, the constraints of a VLSI layout are always so complex that human error cannot be avoided. As a result, the designer should use layout checking tools to guarantee the correctness of the current layout design. In considering the design of the checking tools, the key factor is to check automatically all the constraints of the layout, including many unavoidable constraints (such as all the physical constraints of the electrical components) and extra constraints (such as the constraints of the product requirement and the user specified requirements in the manufacturing process) between any pair of components. Those constraints of the VLSI layout are usually defined as the design rules [9] for the design of the attendant layout, and the checking process of those constraints is called design rule checking (DRC) [11–16, 20, 21] process. Since every component in the layout must be subjected to a design rule checking operation to verify whether the layout is correct, it is necessary to develop an efficient design rule checker for VLSI layout design automation.

As we know, the efficiency of a design rule checker depends heavily on the use of an efficient spatial data structure for the storage of graphical information of the layout. Some operations of region query and neighbouring search based on the efficient spatial data structure are considered as key functions used by the design rule checker. Therefore, many different storage methods ranging from the simple linear lists to the more sophisticated data structures, such as corner-stitching [23] and quad trees [1–6], have been presented to enhance the applications of VLSI–CAD tools design [11–16, 21].

A quad tree, as indicated by its name, means that it repeatedly divides space into quadrants. This subdivision generally will go ahead until the quadrants are so small that each contains only a few objects. Recently, besides Brown's multiple storage quad tree (MSQT) [3] presented in 1986 and Weyten's quad list quad tree (QLQT) [4], which is an improved MSQT, presented in 1989, a further improved quad tree called the binary balanced quad tree (BBQT) [6, 25] has been proposed in 1992. Both the MSQT and QLQT data structures are constructed by regularly dividing a quadrant recursively into four subquadrants with equal size. This kind of division method will be fair for a uniformly distributed layout. Unfortunately, the layout objects usually are not distributed in a completely uniform matter. Moreover, in the worst case, these two data structures, MSQT and QLQT, will lead to a linear time complexity but not an expected time order of $O(\log N)$ for the tree search operation. The BBQT data structure maintains a balance property of the tree construction and ensures that the tree search operation is in a time complexity of $O(\log N)$ for any randomly generated layout. From the balance property, we can develop many region query operations whose time complexity is in $O(\log N)$ for VLSI-CAD tools design.

Generally, a design rule checker (DRC) is one of the VLSI–CAD layout tools. Two of the most often used design rule checking methods are incremental design rule checking (IDRC) [8, 10] and global design rule checking (GDRC) [17, 18], respectively. IDRC is a dynamic and interactive approach. In this approach, whenever one component is added to the layout, all the related design rules between this component and the other neighbour-

Paper 8850E (C3), first received 30th January 1991 and in revised form 6th January 1992

The authors are with the Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan, ROC

This work was supported in part by the National Science Council of the Republic of China under contract NSC 81-0404-E-009-107.

ing components should be checked over. If all the relational design rules are error-free, then the component is allowed to be added into the layout. Otherwise, the component should be discarded from the layout. The checking time needed in this approach is hidden in the layout editing time, so that the total design time can be reduced. However, as the number of the components in the layout grows, the checking time for each added component increases rapidly. Hence, the checking time greatly delays the layout editing time so that the incremental checking becomes inefficient for a large layout.

In addition, the GDRC is an overall approach. It usually uses a plane-sweep method to enumerate all the components in the layout one by one to check every related design rule for each component and to exhaustively pick out all the errors occurring in the complete layout. However, whenever the layout is lightly modified after GDRC, the entire layout must be checked once more. As a result, the checking process may waste much time to do rechecking more than once while dynamically editing the layout.

This paper presents a complementary design rule checker, CDRC, which is a combination of the modified incremental design rule checker and the modified global design rule checker. The CDRC first processes a complementary incremental design rule checking (CIDRC) in order to check part of the simple related design rules (type C_1, C_2, and C_3) and to hide the check time during editing of the physical layout, and secondly, performs a complementary global design rule checking (CGDRC) to check those design rules (type C_4, C_5, and C_6) which were not checked by CIDRC. In the following Section, we shall discuss all of the six types of constraint used by CDRC and also will give an effective classification for those constraints (type C_1, C_2, and C_3) checked by CIDRC in the first interactive phase and the other constraints (type C_4, C_5, and C_6) checked by CGDRC in the second batch phase.

2 Constraints analysis

2.1 Constraints for interactive phase of CDRC

The interactive phrase of our CDRC is a complementary IDRC (CIDRC), which is considered as a dynamic verification process for design rule checking. In our system, whenever one new component is added to the layout, one must check whether the position of the component is correct by using the CIDRC, concerning part of the related design rules and other requirements. These design rules and requirements are grouped into three types, namely width constraints (C_1), overlap constraints (C_2), and clearance constraints (C_3).

Type 1, width constraints (C_1): For each rectangle, it is necessary to set up some distance constraints including the max/min constraints ($\leqslant$ or $\geqslant$) or the fixed constraints ($=$) between the left- and right-hand edges of the attentive rectangle. The distance constraints, shown in Fig. 1, are called the width constraints for the rectangles of a given layout.

Type 2, overlap constraints (C_2): For the overlap relation existing in each pair of rectangles which are partially overlapped, the width of the overlapped area is formed from the distance between the right-hand edge of the left-hand rectangle and the left-hand edge of the right-hand rectangle. These distance constraints also named as left-right constraints are dependent on the combination of rectangles in different layers. These constraints, illustrated in Fig. 2, are called overlap constraints.

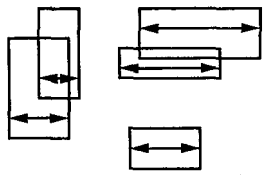

Fig. 1 *Width constraints* (C_1)

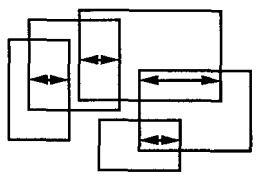

Fig. 2 *Overlap constraints* (C_2)

Type 3, Clearance constraints (C_3): For the clearance relation existing in each pair of rectangles which are partially or completely overlapped, the distance constraints between both of the left- (right-) hand edges of these two rectangles are called left-left (right-right) constraints. Those constraints, illustrated in Fig. 3, are also called the clearance constraints.

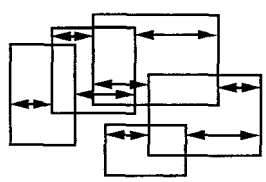

Fig. 3 *Clearance constraints* (C_3)

In order to explain the geometrical relations between rectangles for the checking algorithms of CIDRC presented in the following Section, some topological definitions are given as follows:

Definition 1 (*see Fig. 4*): π_{2-1} is a geometrical relation which presents one kind of the overlap relationship between two different rectangles, R_i and R_j, so that $R_i \pi_{2-1} R_j$: $Lx(R_i) < Lx(R_j)$ and $Rx(R_i) < Rx(R_j)$ and $R_i \cap R_j \neq \{\}$, where both $Lx(\cdot)$ and $Rx(\cdot)$ are functions for evaluating the x-coordinates of the left-hand right-hand edges of respectively, of the specified rectangle.

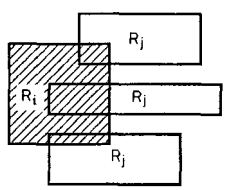

Fig. 4 $R_i \pi_{2-1} R_j$

Definition 2 (*see Fig. 5*): Similarly to π_{2-1}, π_{2-2} is a geometrical relation denoting the other kind of overlap relationship between two different rectangles R_i and R_j so that $R_i \pi_{2-2} R_j$: $By(R_i) < By(R_j)$ and $Ty(R_i) < Ty(R_j)$ and $R_i \cap R_j \neq \{\}$, where both $By(\cdot)$ and $Ty(\cdot)$ are functions

for evaluating y-coordinates of the bottom and top edges respectively, of the specified rectangle.

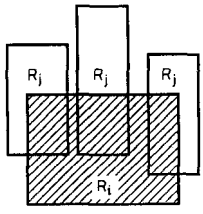

Fig. 5 $R_i \pi_{2-2} R_j$

Definition 3 (see Fig. 6): π_{3-1} is a geometrical relation which presents one kind of clearance relationship between two different rectangles R_i and R_j so that $R_i \pi_{3-1} R_j$: $Lx(R_i) \leqslant Lx(R_j)$ and $Rx(R_j) \leqslant Rx(R_i)$ and $By(R_j) \leqslant By(R_i)$ and $Ty(R_i) \leqslant Ty(R_j)$ and $R_i \cap R_j \neq \{\}$.

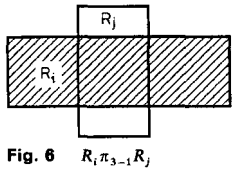

Fig. 6 $R_i \pi_{3-1} R_j$

Definition 4 (see Fig. 7): Similarly to π_{3-1}, π_{3-2} is a geometrical relation denoting the other kind of clearance relationship between two different rectangles, R_i and R_j so that $R_i \pi_{3-2} R_j$: $Lx(R_i) \leqslant Lx(R_j)$ and $Rx(R_j) \leqslant Rx(R_i)$ and $By(R_i) \leqslant By(R_j)$ and $Ty(R_j) \leqslant Ty(R_i)$ and $R_i \cap R_j \neq \{\}$.

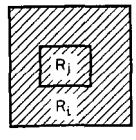

Fig. 7 $R_i \pi_{3-2} R_j$

2.2 Constraints for batch phase of CDRC

The batch phase of our CDRC is a complementary GDRC (CGDRC) which is considered as a final static verification process. It generally takes place at the end of the layout design and verifies the correctness of all of the constraints which were not checked over during the CIDRC process. In our system, those design rules which, in conjunction with the user specified requirements and the implicit electrical connectivities embedded in the layout and checked by CGDRC, should be the complementary constraints of those that have already been checked by the CIDRC. In other words, the set of constraints checked by the CGDRC and that checked by the CIDRC must be disjointed. Moreover, the union of these two sets of constraints must include all the layout constraints that need to be checked. Those constraints for the CGDRC can be grouped into three types, namely visible separation constraints (C_4), invisible separation constraints (C_5) and convex-vertex separation constraints (C_6).

Type 4, visible separation constraints (C_4): In this paper, 'separated rectangles' means that the pair of rectangles under consideration are never overlapped nor do their edges partially touch each other. For each pair of rectangles having a visible separation constraint, there must be no such a third rectangle blocking the concerned pair of rectangles. The visible relation implies that there are some distance constraints between the right-hand edge of the left-hand rectangle and the left-hand edge of the right-hand rectangle. Examples of those constraints are illustrated in Fig. 8.

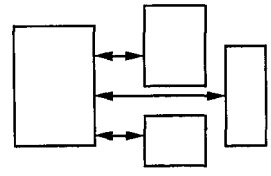

Fig. 8 *Visible separation constraints* (C_4)

Type 5, invisible separation constraints (C_5): The invisible separation constraints, as shown in Fig. 9, are similar to the visible separation constraints, except that there must be at least one rectangle blocking the concerned pair of rectangles.

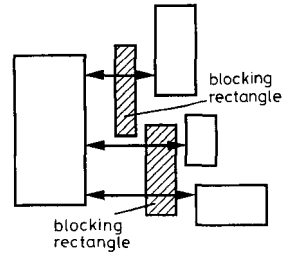

Fig. 9 *Invisible separation constraints* (C_5)

Type 6, convex-vertex separation constraints (C_6):For a pair of separated rectangles, as shown in Fig. 10, the distance constraints between the south-east corner of the left-hand rectangle and the north-west corner of the right-hand rectangle or between the north-east corner of the left-hand rectangle and the south-west corner of the right-hand rectangle are considered as the convex-vertex separation constraints.

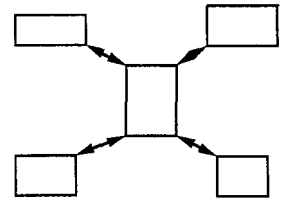

Fig. 10 *Convex-vertex separation constraints* (C_6)

In order to explain the CGDRC algorithm which will be presented in this paper, some extra topological definitions for the geometrical relation of the separated rectangles are given as follows:

Definition 5, parallel-edge separation relation (see Fig. 11): $\pi_{4,5}$ is a geometrical relation which presents one kind of the separation relationships between two different rectangles, R_i and R_j, so that $R_i \pi_{4,5} R_j$: $Lx(R_i) < Lx(R_j)$

and $R_i \cap R_j = \{\}$ and $Ty(R_i) \geq By(R_j)$ and $Ty(R_j) \geq By(R_i)$.

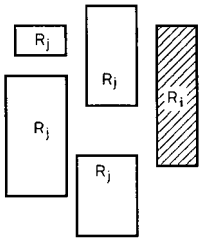

Fig. 11 *Parallel_Edge separation relation* $R_i \pi_{4,5} R_j$

Definition 6, convex-vertex separation relation (see Fig. 12): Similarly, π_6 is a geometrical relation denoting the other kind of the separation relationships between two different rectangles, R_i and R_j, so that $R_i \pi_5 R_j$: $Lx(R_i) < Lx(R_j)$ and $R_i \cap R_j = \{\}$ and $[Ty(R_i) < By(R_j)$ or $By(R_i) > Ty(R_j)]$.

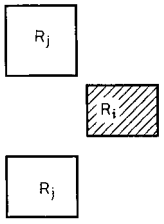

Fig. 12 *Convex_Vertex separation relation* $R_i \pi_6 R_j$

Definition 7, dynamic projection set (DPS): The dynamic projection set of rectangles, as shown in Fig. 13, is defined as a union interval of the vertical projection intervals of those rectangles.

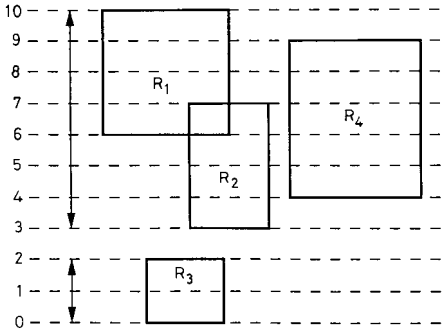

Fig. 13 *Dynamic projection set of* $\{R_1, R_2, R_3, R_4\}$ *in the union intervals of* $[0, 2]$ *and* $[3, 10]$

2.3 Basis for constraint classification

According to the above classification, type C_1, C_2 and C_3 constraints should be checked in CIDRC. As CIDRC is an interactive phase, all the constraints should be checked in real time during this phase. The general feature for type C_1, C_2 and C_3 constraints is that they are close together and near the current checking rectangle R_i. Hence it is quick and easy to apply simple region query operations to verify all of the type C_1, C_2 and C_3 constraints in an online real-time mode during the interactive phase of CDRC.

On the other hand, type C_4, C_5 and C_6 constraints apply between the current checking rectangle, R_i and the other concerned rectangle R_j, which may be located very far away form R_i. Therefore the search time for checking type C_4, C_5 and C_6 constraints is usually longer than that for checking type C_1, C_2 and C_3 constraints. In general, CGDRC requires the application of a global view to the whole layout to verity type C_4, C_5 and C_6 constraints. From the above discussion, the basis for the classification of C_1, C_2, C_3 and C_4, C_5, C_6 is easy to understand. It cannot be done in any other way without further reasonable consideration.

3 Region queries and BBQL quad tree

3.1 Painted quad tree

A VLSI quad tree is constructed by repeatedly dividing each of those quadrants satisfying some criteria into four equal subquadrants by associating a tree node with each quadrant and drawing four links from each tree node to its four children tree nodes [1–3]. A painted quad tree as shown in Fig. 14 was proposed by Nandy, Patnaik, and Ramakrishnan [19, 24] in 1986 and is one kind of VLSI quad tree, of which the dividing process does not stop until every leaf quadrant is completely painted or deliberately left blank.

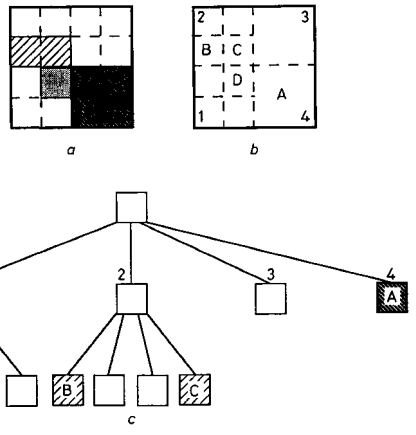

Fig. 14 *Example layout and its corresponding painted quad tree*

a Unit blocks covered by layout rectangles are shaded
b Block decomposition for the source layout shown in *a*
c Corresponding painted quad tree representation of layout shown in *a*

3.2 Multiple storage quad tree (MSQT)

In 1986, Brown [3] developed a new VLSI quad tree data structure as shown in Fig. 15 to give a solution for those objects intersecting more than one quadrant. The improvement is to multiply store such an object intersected with several leaf quadrants by one referenced pointer for each leaf quadrant. That means some objects will be stored in more than one leaf quadrant. This approach wastes a portion of space to store multiple pointers. Since some objects are located in more than one leaf quadrant, to avoid reporting some objects more than once, the further improvement [4–5] has been presented of first marking the object the first time it is reported, and then never reporting an object which has already been marked. This data structure requires the marking and

unmarking operations to maintain their validity whenever the objects are searched.

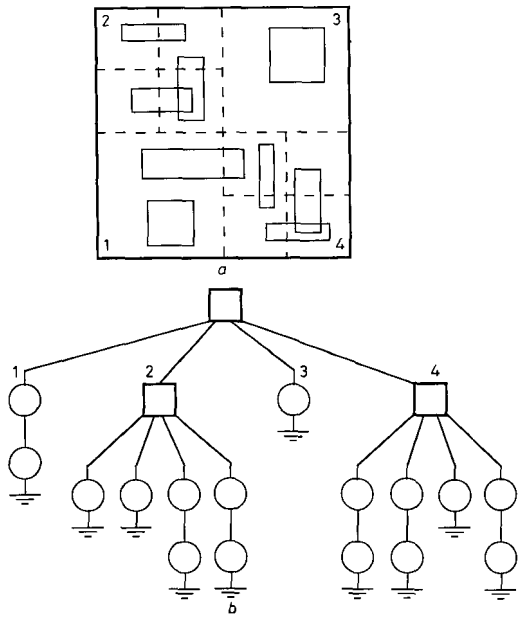

Fig. 15 *Example layout and its corresponding MSQT representation*

a The layout plane is divided into quadrants repeatedly until each of the leaf quadrants containing two or less than two rectangles

b Corresponding MSQT with a threshold value of two

3.3 Quad list quad tree (QLQT)

In considering the problem of enumerating the multiple objects in Brown's MSQT, the QLQT data structure was presented in 1989 [4] to split the single long linked list of object reference nodes into four distinct shorter lists. For any object intersecting the leaf quadrants, a reference to this object will be included in one of the four lists of the leaf quadrant according to the relative position of this object with respect to the presented region of the leaf quadrant. The assigning procedure is illustrated in Weyten and Pauw's paper [4]. While searching or finding objects in the QLQT, each object will be accessed only once from the four distinct lists instead of from the longer list in the MSQT, by using some critical and efficient rules proposed by Weyten and Pauw.

However, in the worst case, if the objects are unevenly distributed in the 2D plane, both the MSQT and the QLQT will become skewed and heavily unbalanced and will require a linear time complexity to do search operations.

3.4 Binary balanced QLQT (BBQLQT)

In considering Brown's MSQT and Weyten's QLQT, the division method in constructing the entire quad tree data structure is to recursively split the quadrant containing more than T_N objects into four subquadrants with equal size as shown in Fig. 15*a*, where T_N is a threshold number chosen by the designer. Each father quadrant represented by a tree node has four children subquadrants.

This kind of division method will be fair for a uniformly distributed layout. Unfortunately, the layout objects are usually not distributed in a completely uniform manner. In the worst case, these two data structures will lead to a linear time complexity but not in the expected time order of $O(\log N)$ for the tree search operations.

In order always to maintain the expected time order of $O(\log N)$ for the tree search operations, Hsiao and Jang [6, 25] have presented two binary balanced quad tree data structures to improve both the MSQT and the QLQT, respectively. This kind of balanced quad tree improves the division method as shown in Fig. 16 by

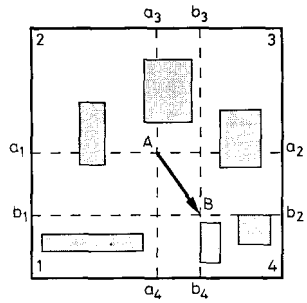

Fig. 16 *Dividing lines used for the MSQT and QLQT, $\overline{a_1a_2}$ and $\overline{a_3a_4}$ may split the quadrant into four equal-size subquadrants*

Note that the dividing lines of the binary balanced quad tree, $\overline{b_1b_2}$ and $\overline{b_3b_4}$, will split the quadrant into four distinct subquadrants, each of which intesects with almost the same number of objects

taking the distribution of layout objects into consideration to make the split four subquads contain almost the same amount of objects. From the experimental results shown by Hsiao and Jang [6, 25], the QLQT has an average improvement of 15–20% to the MSQT, and its BBQLQT also has a further average improvement of 15–20% to the QLQT. The time involved in the creation of the BBQLQT is given in Fig. 17. The source code associated with a low-cost layout editor, PC_UNION [26], is written in C based on an IBM compatible PC/AT 486 under MS DOS.

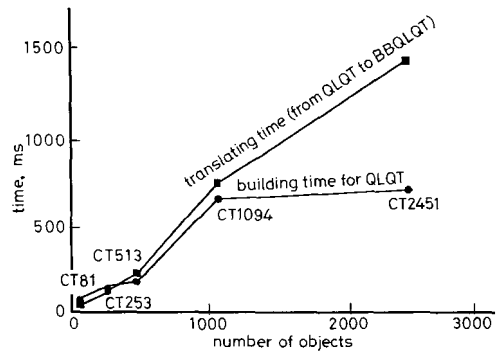

Fig. 17 *Illustration for the time involved in the creation of BBQLQT from QLQT and the time needed for building the QLQT from source data file*

There are five tested circuits for which the number of objects ranges from 81 to 2451

3.5 Region queries

Region queries always play an essential role in applications to VLSI layout design [12, 21]. The operation of region query means a search of all objects that intersect with a specified region, which is sometimes called a query window. As the speed of the query operation may depend

heavily on the number of objects in the layout plane, it is important to develop a highly efficient spatial data structure to manipulate those objects.

The most important advantage of the BBQLQT is that it offers several highly efficient operations for region queries. The following elementary operations, RQ1()–RQ5(), are outlined in term of their functions and time complexities, where N stands for the total number of rectangles in the source layout.

Moreover, it is assumed that m stands for the number of rectangles partially or entirely included in the query window, m_1 denotes the number of new rectangles which are compared to the rectangles encountered in the last query operation, m_2 denotes the number of rectangles entirely included in the query window, and m_3 represents the number of rectangles partially intersected by the query window. Finally, wx_1, wy_1, wx_2 and wy_2 represent the bottom-left and top-right co-ordinates of the query window and $Tm(\cdot)$ is a function of time complexity for various region queries. Some examples for the operations of region queries are shown in Fig. 18. The formal descriptions for the region queries are illustrated as follows:

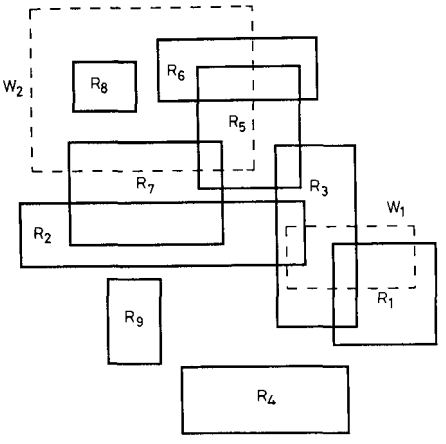

Fig. 18 *Example for $RQ_1() \sim RQ_5()$*
RQ_1 (BBQLQT, W_1) = {R_1, R_2, R_3}
RQ_2 (BBQLQT, W_2) = YES
RQ_3 (BBQLQT, W_2) = {R_5, R_6, R_7, R_8}
RQ_4 (BBQLQT, W_2) = {R_8}
RQ_5 (BBQLQT, W_2) = {R_5, R_6, R_7}

(i) **RQ1(BBQLQT, wx1, wy1, wx2, wy2)**
: This operation will find out all of the rectangles entirely or partially included in the query window.
Tm(RQ1) = O(mlogN)

(ii) **RQ2(BBQLQT, wx1, wy1, wx2, wy2)**
: Similar to RQ1(), this operation will reply YES or NO to show whether there are new rectangles which did not apear in the output of the last RQ1() operation.
Tm(RQ2) = O(logN)

(iii) **RQ3(BBQLQT, wx1, wy1, wx2, wy2)**
: Similar to RQ2(), this operation will return the new rectangles which did not appear in the output of the last RQ1() operation.
Tm(RQ3) = O($m_1$logN)

(iv) **RQ4(BBQLQT, wx1, wy1, wx2, wy2)**
: This operation will search output all rectangles entirely involved inside the query window.
Tm(RQ4) = O($m_2$logN)

(v) **RQ5(BBQLQT, wx1, wy1, wx2, wy2)**
: This operation will search output all rectangles partially intersected with the query window.
Tm(RQ5) = O($m_3$logN)

4 Complementary incremental design rule checking (CIDRC)

4.1 Complementary incremental design rule checking algorithm

In this Section, the presented CIDRC algorithm is a dynamic verification process and is regarded as an interactive phase of CDRC. Whenever a new component is added into the current layout, those neighboring components must be gradually checked according to the constraints of type C_1, C_2, and C_3. The CIDRC algorithm is shown as follows:

```
CIDRC_Algorithm(BBQLQT, Ri)
/* This CIDRC_Algorithm( ) completes a complementary incremental design rules checking for the added component,
Ri, in the current layout.*/
  {
      Buffer OR-SET;
      /* This buffer will be used for storing those components which overlapped Ri.*/
      justify the layer type of Ri;
      check C1 for Ri by table look-up;
      /* Compare the width of Ri with the width constraint in the design rule table.*/
      if (any violation in the width constraint)
          discard Ri and exit (C1 rule-error);
      Find_Overlap (BBQLQT, Ri, OR-SET);
      /* Find the overlapped components which overlap the added component and store them into the buffer OR-
      SET.*/
      if (OR-SET is not empty)
          Overlap_Checking (BBQLQT, Ri, OR-SET);
           /*Check the overlap constraints (Type C2) and the clearance constraints (Type C3) between Ri and the
           components of OR-SET.*/
      add Ri into BBQLQT;
      return (Ri error-free);
 }/* end of CIDRC_Algorithm( )*/
```

```
Find_Overlap(ABBQLQT, R_i, OR-SET)
 {
   OR-SET = { };
   obtain the left-bottom and right-top coordinates of R_1: rx1, ry1, rx2 and ry2.
   OR-SET = RQ1(BBQLQT, rx1, ry1, rx2, ry2);
   /* Run the region query operation, RQ1(), and store the returned rectangles into OR-SET.*/
 }

Overlap_Checking(BBQLQT, R_i, OR-SET);
 {
   for (R_j, j = 1, 2, ..., where R_j belongs to OR-SET and satisfies that R_i π_2-1 R_j or R_j π_2-1 R_i or R_i π_2-2 R_j or
   R_j π_2-2 R_i) /* see Figs. 4 and 5*/
      {
       check C_2 between R_i and R_j by table look-up;
       /*Compare the distance of the overlapped area between R_i and R_j with the constraints from the design rule
       table.*/
       if (any violation in the overlap constraint)
          discard R_i and exit (C_2 rule-error);
       check C_3 between R_i and R_j by table look-up;
       /*Compare the distance of the clearance relation between R_i and R_j with the constraints from the design rule
       table.*/
       if (any violation in the clearance constraint)
          discard R_i and exit (C_3 rule-error)
       }/* end of for */
    for (R_j, j = 1, 2, ..., where R_j belongs to OR-SET and satisfies that R_i π_3-1 R_j or R_j π_3-1 R_i or R_i π_3-2 R_j or
    R_j π_3-2 R_i) /* see Fig. 6-7 */
       {
        check C_3 between R_i and R_j by table look-up;
        if (any violation in the clearance constraint)
          discard R_i and exit (C_3 rule-error)
   }/* end of for */
  }/* end of Overlap_Checking()*/
```

4.2 Complexity analysis

According to the aforementioned algorithms of CIDRC_Algorithm(), Find_Overlap() and Overlap_Checking(), the computing time for the CIDRC algorithm is limited by Find_Overlap() and the number of components in OR-SET. The time complexity for Find_Overlap(), $O(m \log N)$, is the same for that required for applying a region query to the BBQLQT, RQ1(). For a normal VLSI layout, the number of the rectangles intersected with the query window m is much less than the total number N of layout rectangles. Moreover, from the FOR loops in Overlap_Checking(), the time complexity of Overlap_Checking() should be within $O(m)$. As a result, the overall time complexity for the CIDRC_Algorithm() will be maintained in $O(m + m \log N) \approx O(\log N)$, for a normal VLSI layout.

5 Complementary global design rule checking (CGDRC)

5.1 Complementary global design rule checking algorithm

As the batch phase of CDRC, the complementary global design rule checking algorithm contains two basic steps. The first step is to support a vertical sweeping line horizontally jumping from the left to the right boundary of the layout to check all the horizontal type C_4, C_5, and C_6 constraints. Similarly, in the second step, all the vertical type C_4, C_5, and C_6 constraints will be checked over by maintaining a horizontal sweeping line jumping from bottom to top.

Considering a great number of overlapped rectangles in a two dimensional plane, the sweeping line algorithm will keep up a vertical (horizontal) sweeping line to sweep from the left (bottom) boundary to the right (top) boundary of the layout plane by visiting the left (bottom) and right (top) edges of all of the rectangles in $x(y)$-directional order. This approach is quite useful in global design rule checking. Therefore, to create an efficient sweeping line algorithm based on BBQLQT will very effectively promote the performance of our CGDRC algorithm. Fortunately, in 1990, Hsiao & Tsai [7] presented a generalised sweeping line algorithm based on common region query operations provided by such as the MSQT, QLQT, and BBQLQT. It can successfully be used in the following CGDRC algorithm:

```
CGDRC_Algorithm(BBQLQT)
/* This CGDRC_Algorithm completes a complementary global design rule checking for the entire layout. It uses the
sweeping line algorithm to check the layout from left to right, and then from bottom to top by the similar checking
method.*/
  {
      Horizontal_Checking(BBQLQT);
      /* Complete a horizontal checking for the given layout.*/
      Vertical-Checking(BBQLQT);
      /* Similar to Horizontal_Checking(.)*/
       if (no violation)
         return(The given layout is error-free);
  }/* end of CGDRC_Algorithm() */
```

```
Horizontal_Checking(BBQLQT)
 {
      Buffer E-SET;
        while (the sweeping line algorithm [7] does not finish yet)
            {
            run the sweeping line algorithm to get the x-coordinate of the next event(E);
            E-SET = RQ1(BBQLQT, E, Ly1, E, Ly2);
            /* Base on event E to do region query operation, RQ1(), and to produce E-SET = {Ri | i = 1, 2, ..., m,
            Lx(Ri) = E or Rx(Ri) = E}, where Lx(Ri) and Rx(Ri) are the x-coordinates of the left and right edge of Ri
            respectively.*/
            for (Ri, i = 1, 2, .., m)
              if (Lx(Ri) = = E)
                Separation-Checking(Ri);
                /* Apply Separation_Checking( ) to check the constraints of type C4, C5, and C6 associated with
              Ri.*/
            }/* end of while */
 }/* end of Horizontal_Checking( )*/

Separation_Checking(Ri).
/* This algorithm completes the separation design rule checking in respect of the left edge of Ri */
 {
      Buffer V-SET, C-SET, I-SET, Obuf;
      V-SET = C-SET = I-SET = {};
      Obuf = RQ1(BBQLQT, Lx(Ri), By(Ri), Rx(Ri), Ty(Ri));
      /* The buffer Obuf stores the rectangles that overlapped with Ri. */
      determine the horizontal width of the whole layout, W(layout);
      get the maximum visible/invisible constraint values of Ri, Vi/Ii, from the design rule table;
      /* Vi, Ii ≪ W(layout)*/
      get the x-coordinate of the left boundary of the whole layout, Lx1;
      W = MIN{Lx(Ri) − Lx1, MAX{Vi, Ii}};
      /* Determine the checking length of the maximum parallel-edge separation constraint window */
      P = Lx(Ri) − W;
      /* Define the x-coordinate of the left boundary of the processing window*/
      Find_Visible_Invisible(Ri, W, P, I-SET, V-SET, Obuf);
      /* Apply Find_Visible_Invisible() to search the layout and then obtain the visible separation rectangle set,
      V_SET, and the invisible separation rectangle set, I_SET, for Ri. */
      obtain the maximum convex-vertex separation constraint value of Ri, Ci ;
      Add RQ1(BBQLQT, Lx(Ri)-Ci, Ty(Ri), Lx(Ri), Ty(Ri) + Ci) into C-SET;
      Add RQ1(BBQLQT, Lx(Ri)-Ci, By(Ri)-Ci, Lx(Ri), By(Ri)) into C-SET;
      /* See Fig. 19*/
```

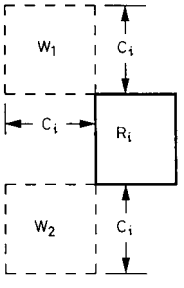

Fig. 19 *Example illustrating the maximum convex-vertex separation window W_1, W_2*

```
      for (Rj, j = 1, 2, ..., such that Rj in V-SET)
        {
          check C4 between Ri and Rj ;
          /* Compare the distance of the visible separation relation between Ri and Rj with the constraint from the
          design rule table.*/
          if (any violation in the visible separation constraint)
            print(C4 rule-error between Ri and Rj)
        }/* end of for */
      for (Rj, j = 1, 2, ..., such that Rj in I-SET)
        {
          check C5 between Ri and Rj ;
          /* Compare the distance of the invisible separation relation between Ri and Rj with the constraint from the
          design rule table.*/
```

```
        if (any violation in the invisible separation constraint)
          print(C5 rule-error between Ri and Rj)
        }/* end of for */
      for (Rj, j = 1, 2, ..., such that Rj in C-SET and Ri π6 Rj)
        {
        check C6 between Ri and Rj;
        /* Compare the distance of the convex-vertex separation relation between Ri and Rj with the constraint from
        the design rule table. */
        if (any violation in the convex-vertex separation constraint)
          print(C6 rule-error between Ri and Rj)
        }/* end of for */
  }/* end of Separation_Checking()*/

Find_Visible_Invisible(Ri, W, P, I-SET, V-SET, Obuf)
  {
      Buffer Obuf, I_SET, V_SET, Buf1, Buf2, Buf3, TempBuf;
      DPS I1, I2,I3; /* Dynamic Projection Set, refer to Definition 7*/
      float W, P, LP;

      get the minimum width of the rectangles of the whole layout, Wmin;
      Buf1 = RQ1(BBQLQT, P, By(Ri), Rx(Ri), Ty(Ri));
      /* The buffer Buf1 stores the rectangles (Rj) that satisfy the relation Ri π4, 5 Rj .*/
      Buf1 = Buf1-Obuf;
      I1 = Create_DPS(Buf1);
      if (I1 = = { }) return();
      W = W/2;
      P = P + W;
      Buf2 = RQ1(BBQLQT, P, By(Ri), Rx(Ri), Ty(Ri));
      /* The buffer Buf2 stores the rectangles that are intersected with the right half part of the processing window.*/
      /* See Fig. 20 */
      Buf2 = Buf2 − Obuf;
      I2 = Create_DPS(Buf2);
      if (I1 = = I2)/* See Fig. 21 */
        { Add Buf1-Buf2 into I-SET;
          Find_Visible_Invisible (Ri, W, P, I-SET, V-SET, Obuf);
        }/* end of if */
      else /* See Fig. 22 */
      {
        while (I1 > I2)
          {  LP = P;
             Buf3 = Buf2;
```

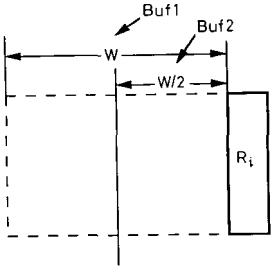

Fig. 20 *Two windows for the buffers Buf_1 and Buf_2*

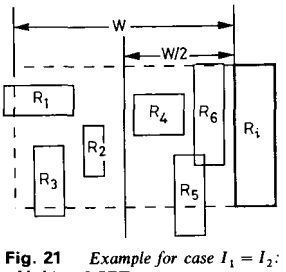

Fig. 21 *Example for case $I_1 = I_2$: the rectangles R_1, R_2 and R_3 are added into I-SET*

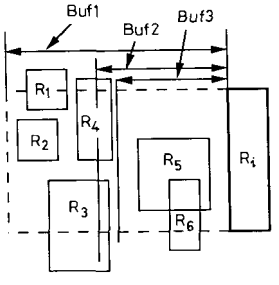

Fig. 22 *Example for case $I_1 > I_2$, the rectangles R_1, R_2 and R_3 are added into I-SET, and the rectangle R_4 is added into V-SET*

```
        W = W/2; P = P − W;
        Buf2 = RQ1(BBQLQT, P, By(Ri), Rx(Ri), Ty(Ri));
        Buf2 = Buf2 − Obuf;
        I2 = Create_DPS(Buf2);
      }/* end of while */
    /* Search for the rectangles of the invisible separation area in case I1 > I2. */
    while (W ≥ Wmin)
      { W = W/2;
       TempBuf = RQ1(BBQLQT, (P + LP)/2, By(Ri), Rx(Ri), Ty(Ri));
        TempBuf = TempBuf − Obuf;
        I2 = Create_DPS(TempBuf);
        if (I1 = = I2)
              { P = (P + LP)/2;
                Buf2 = TempBuf;
              }/* end of if */
            else /*I1 > I2 */
             { P = (P + LP)/2;
                Buf3 = TempBuf;
              }/* end of else */
       }/* end of while */
      Add (Buf1-Buf2) into I-SET;
       /* Use binary search method to search for the rectangles of the visible separation area */
      I3 = Create_DPS(Buf3);
      for (Rj, j = 1, 2, ..., such that Rj in (Buf2-Buf3))
        if ([By(Rj), Ty(Rj)] ∉ I3)
            Add Rj into V-SET;
          else
            Add Rj into I-SET;
        P = LP;
        W = Lx(Ri)-LP;
        I1 = I3;
        Find_Visible_Invisible (Ri, W, P, I-SET, V-SET, Obuf);
    }/* end of else */
}/* end of Find_Visible_Invisible()*/
```

5.2 Complexity analysis

According to CGDRC_Algorithm( ), the computing time for the final batch phase of the presented CDRC will be limited by the function of Horizontal_Checking( ) or similarly by the function of Vertical_Checking( ). In considering the function of Horizontal_Checking( ), its while-loop takes at most $2N$ events to scan all the source layout over by the horizontal sweeping line algorithm presented in Reference 7. Moreover, the function of Separation_Checking( ) should be repeatedly involked once for each while-loop. The time complexity of Separation_Checking( ) depends on the region query operations of the BBLQT and the function of Find_Visible_Invisible( ). The time complexity for the region query operations of the BBQLQT will be in $O(\log N)$ provided that, for a large layout, the number of rectangles found from the query window m is assumed to be much less than the total number N of layout rectangles. For the same reason, the amount of calling to region query operation in the function of Find_Visible_Invisible( ) is also much less than N. Hence, the time complexity of Find_Visible_Invisible( ) still is limited in $O(\log N)$, and the number of rectangles in V-SET, I-SET, and C-SET which will be checked in the function of Separation_Checking( ) is much less than N. Consequently, the time complexity of Separation_Checking( ) is in $O(\log N)$, and the overall time complexity for the CGDRC_Algorithm( ) has been proven to be in $O(N \log N)$.

6 Illustrative example and experimental results

To understand the details of the presented CDRC, Fig. 23 shows that the CIDRC processes the width checking, the overlap checking and the clearance checking for any rectangle currently being added into the source layout in the first interactive checking phase. After finishing the editing process of the given layout, CGDRC then checks the visible separation constraints, the invisible separation constraints and the convex-vertex separation constraints between each pair of rectangles in the batch checking phase by the sweeping line approach [7], as shown in Fig. 24. Therefore, CDRC completes the checking implementation for all the layout constraints by the above two disjointed checking phases.

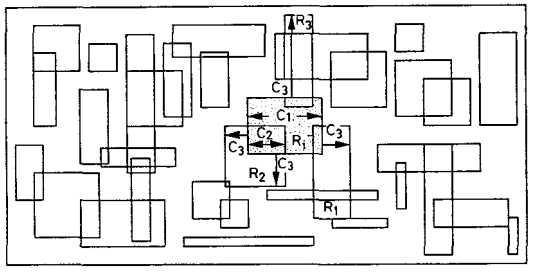

Fig. 23 *Interactive phase of CDRC*

After the rectangle R_i is added into the current layout, the CIDRC will immediately check the width constraint for R_i and the overlap and clearance constraints between R_i and the intesected rectangles R_1, R_2 and R_3

Besides the illustrative example shown in Figs. 23 and 24 for the checking process of CIDRC and CGDRC, the time complexity of the algorithm has to be demonstrated through implementation. Table 1 and Fig. 25 show the experimental results that demonstrate the time complex-

ity of CIDRC and CGDRC. Five circuit layouts are used as test examples for which the number of objects range from 81 to 2451. From Fig. 25, the time complexity of

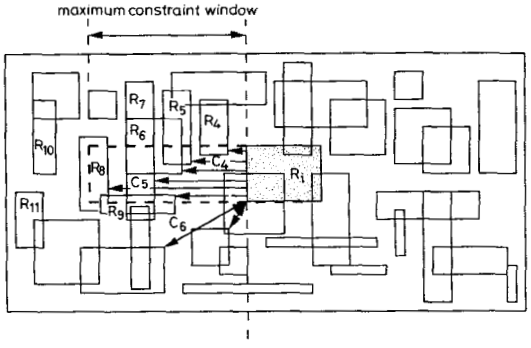

Fig. 24 *Batch phase of CDRC*

While the sweeping line jumps from left to right and the rectangle R_i is being enumerated, the CGDRC will check those constraints, type C_4, C_5, and C_6, that are associated with R_i

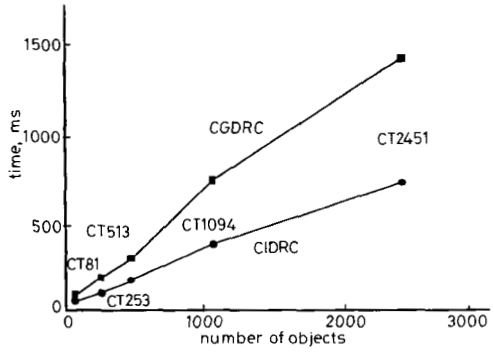

Fig. 25 *Experimental results demonstrating the time complexity of CIDRC and CGDRC*

CIDRC is in $O(\log N)$ for each interactively added object or in $O(N \log N)$ for the sum of all of the added layout objects, and the time complexity of CGDRC is in $O(N \log N)$. All the experimental time complexities match with the theoretical results. The program associated with a low-cost layout editor, PC_UNION [26], is written in C based on an IBM compatible PC/AT 486 under MS DOS.

7 Conclusions

As we know, for a large layout, IDRC may be too slow to become as a real time checking tool, and GDRC is usually inefficient for rechecking the whole layout more than once. If the given layout is interactively and gradually modified step by step, neither conventional IDRC nor GDRC can achieve a bettter performance. In this paper, therefore, we have established a general geometrical design rule checking model, complementary DRC, to consider some of the layout constraints by the complementary incremental design rule checker (CIDRC) and the other constraints by the complementary global design rule checker (CGDRC). Every layout constraint should be checked only once, either by the function of CIDRC_Algorithm() or of CGDRC_Algorithm(). The time complexity of the presented CDRC, $O(N \log N)$, causes from the combination of the time complexity of the CIDRC, $O(\log N)$ for each object, and the time complexity of the CGDRC, $O(N \log N)$.

This system and the embedded layout editor [26], PC_UNION, are designed on the basis of BBQLQT and its region query functions. The BBQLQT [6, 25] is more efficient both in memory usage and query speed than the most recently published QLQT [4]. One day, if the BBQLQT is further improved, the performance of our system will be promoted without the need for additional design effort. Hence, our system is fully modularised and sufficiently independent of any of the other spatial data structures. Those constraints checked by the interactive phase (CIDRC) and the batch phase (CGDRC) are separated, independent, and disjointed, so that this system is perceived as an excellent real time CAD tool.

8 References

1 SAMET, H.: 'The quad tree and related hierarchical data structure', *ACM Computing Surveys*, 1984, **16**, June, pp. 187–260

2 FINKEL, R., and BENTLEY, J.: 'Quad trees — A data structure for retrievals on composite keys', *Acta Informatica*, 1974, **4**, pp. 1–9

3 BROWN, R.L.: 'Multiple storage quad tree: a simpler faster alternative to bisector list quad trees', *IEEE Trans.*, 1986, **CAD-5**, (3), pp. 413–419

Table 1: Experimental results obtained from CDRC combined with a low-cost layout editor, PC_UNION [26]

CDRC Circuits	CIDRC		CGDRC	
	Number of checked constraints	Checking time ms	Number of checked constraints	Checking time ms
CT81	$C_1 = 81$ $C_2 = 182$ $C_3 = 607$	33.12	$C_4 = 185$ $C_5 = 37$ $C_6 = 324$	57.92
CT253	$C_1 = 253$ $C_2 = 553$ $C_3 = 1557$	89.34	$C_4 = 617$ $C_5 = 104$ $C_6 = 784$	185.86
CT513	$C_1 = 513$ $C_2 = 1135$ $C_3 = 3146$	166.18	$C_4 = 1053$ $C_5 = 274$ $C_6 = 1609$	307.34
CT1094	$C_1 = 1094$ $C_2 = 3006$ $C_3 = 9847$	440.63	$C_4 = 2639$ $C_5 = 757$ $C_6 = 3472$	837.34
CT2451	$C_1 = 2451$ $C_2 = 5352$ $C_3 = 15\,784$	735.67	$C_4 = 5746$ $C_5 = 1576$ $C_6 = 9528$	1405.89

4 WEYTEN, L., and DE PAUW, W.: 'Quad list quad trees: a geometrical data structure with improved performance for large region queries', *IEEE Trans.*, 1989, **CAD–8,** (3), pp. 229–233
5 HSIAO, P.Y., and FENG, W.S.: 'Using a multiple storage quad tree on a hierarchical VLSI compaction scheme', *IEEE Trans.*, 1990, **CAD–9,** (5), pp. 522–536
6 HSIAO, P.Y., and JANG, L.D.: 'On VLSI layout systems spatial data structure: the binary balanced quad list quad trees', *IEEE Trans. CAD*, 1992 (to be published)
7 HSIAO, P.Y., and TSAI, C.C.: 'A new plane-sweep algorithm based on spatial data structure for overlapped rectangles in 2-D plane', Proceedings of COMPSAC, IEEE, 1990, pp. 347–352
8 HSIAO, P.Y., and FENG, W.S.: 'An incremental design rule checking based on quad tree representations'. ISMM International Symposium on *Mini and microcomputers*, Miami Beach, Florida, 1988.
9 OHTSUKI, T.: 'Layout design and verification' (North-Holland, 1985)
10 TAYLOR, G.S., and OUSTERHOUT, J.K.: 'Magic's incremental design-rule checker'. Proceedings of 21st Design Automation Conference, ACM/IEEE, 1984, pp. 160–165
11 SATO, M., KIM, J.B., AWASHIMA, T., and OHTSUKI, T.: 'A theoretically optimal and practically fast algorithm for VLSI geometrical design rule verification'. Proceedings of ISCAS, IEEE, 1988, pp. 1445–1448
12 JEONG, J.C., SHIN, S.Y., LEE, C.D., and YU, Y.U.: 'An efficient sequential range query model for minimum width/space verification'. Proceedings of ISCAS, IEEE, 1988, pp. 330–333
13 BERGER, J., and MAZARE, G.: 'A range searching sub-system used to perform efficient VLSI design checks'. Proceedings of ISCAS, IEEE, 1986, pp. 408–411
14 MODARRES, H., and LOMAX, R.J.: 'A formal approach to design-rule checking', *IEEE Trans.*, 1987, **CAD-6,** (4), pp. 561–573
15 BONAPACE, C.R., and LO, C.Y.: 'LARC2: a space-efficient design rule checker'. Proceedings of ISCAS, IEEE, 1987, pp. 298–301
16 BONAPACE, C.R., and LO, C.Y.: 'An $O(n \log m)$ algorithm for VLSI design rule checking'. Proceedings of 26th Design Automation Conference, ACM/IEEE, 1989, pp. 503–507
17 CHAPMAN, P.T., and CLARK, K. Jr.: 'The scan line approach to design rules checking: computational experiences'. Proceedings of 21st Design Automation Conference, ACM/IEEE, 1984, pp. 235–241
18 CARLSON, E.C., and RUTENBAR, R.A.: 'A scanline data structure processor for VLSI geometry checking', *IEEE Trans.*, 1987, **CAD-6,** (5), pp. 780–794
19 NANDY, S.K., and PATNAIK, L.M.: 'Linear time geometrical design rule checker based on quadtree representation of VLSI mask layout', *Computer Aided Design*, 1986, **18,** (7), pp. 380–388
20 HEDENSTIERNA, N., and JEPPSON, K.O.: 'New algorithms for increased efficiency in hierarchical design rule checking', *Integration, the VLSI Journal*, 1987, **5,** pp. 319–336
21 SATO, M., and OHTSUKI, T.: 'Applications of computational geometry to VLSI layout pattern design', *Integration, the VLSI Journal*, 1987, **5,** pp. 303–317
22 KEDEM, G.: 'The Quad-CIF: a data structure for hierarchical online algorithm'. Proceedings of 19th Design Automation Conference, ACM/IEEE, 1982, pp. 325–357
23 OUSTERHOUT, J.K.: 'Corner stitching: a data-structuring technique for VLSI layout tools', *IEEE Trans.*, 1984, **CAD-3,** pp. 87–100
24 NANDY, S.K., and RAMARKRISHNAN, I.V.: 'Dual quadtree representation for VLSI designs'. Proceedings of 23rd Design Automation Conference, ACM/IEEE, 1986, pp. 663–666
25 HSIAO, P.Y., and JANG, L.D.: 'Using a balanced quad tree to speed up a hierarchical VLSI compaction scheme'. 5th International Conference on *VLSI design*, Bangalore, India, January 1992, pp. 370–371
26 HSIAO, P.Y., and YAN, J.T.: 'PC_UNION: a low cost layout tool for consumer IC design'. International Symposium on *IC design, manufacture and applications*, Singapore, 1991, pp. 452–457