Recognition of Regular Layout Structures

Yu-Cheng Chiang, Shr-Cheng Tsai and Rung-Bin Lin[1]
Yuan Ze University, Taoyuan, Taiwan
[1]E-mail: csrlin@cs.yzu.edu.tw

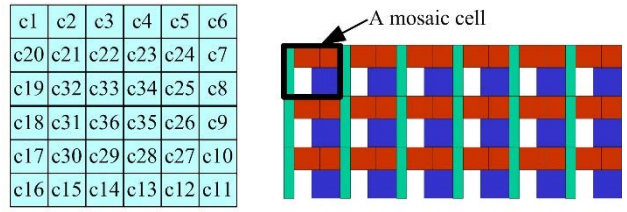
Abstract
This paper presents an algorithm for finding array structures in a layout design. The algorithm can find all the regular layout structures from a flattened layout design without knowing its building blocks beforehand. A potential application of this work is to reduce layout DRC and lithography check time. Experimental results show that our algorithm is efficient and robust.

Keywords
Regular layout, design rule checking, lithography

1. Introduction
A regular layout structure in a VLSI design comprises the same kind of geometrical objects disposed regularly in x and y directions. The structure is often found in a design with memory blocks and customized data paths or in a Structured ASIC [1]. We call such a layout structure array and the problem of finding such a layout structure array recognition problem. Arrays can be used to speed up design rule check (DRC) [2-6] and lithography check [7] for design manufacturability by verifying only a subset of cells in an array [5]. For example, given a 36-cell array in Figure 1(a) for DRC check, we normally perform DRC for all the cells in it. However, checking c22~c36 is not needed because these cells each have the same surrounding as that of c21 [5]. Clearly, we are not interested in an array whose dimensions are not larger than 3-by-3.

There are abundant works about DRC; however, to the best of our knowledge, no algorithm for array recognition problem has been found in the open literature. Some commercial tools do provide array recognition capability. For example, Cadence's Dracula [5] provides a command ARRAY-ENABLED to find only simple array automatically, but no details about how this is done are revealed. Cadence's Assura Physical Verification provides some commands to recognize memory arrays such as DRAM, ROM, SRAM arrays. Details about how this is done is not found in the literature. Because it is known a priori which memory device will be disposed regularly to form an array, memory array recognition is simpler than the problem addressed in our work. The work in [8] is somewhat related to ours, but it only finds out all the repetitive instances. No algorithm is presented there to unearth any array structures. In this paper we propose a systematic approach to array recognition problem. Our work addresses complex array structure recognition as shown in Figure 1(b). Test cases solicited from a major EDA vendor are used to evaluate our approach. Experimental data show that our algorithm is time efficient and very robust, i.e., able to find all the arrays and almost always the largest ones.

The rest of this paper is organized as follows. Section 2 presents a problem definition. Section 3 describes our array recognition algorithm. Section 4 presents some experimental results. Section 5 draws a conclusion.



(a) A 6-by-6 simple array (b) A 3-by-6 mosaic array

Figure 1: Simple array and mosaic array.

2. Preliminaries
The layout objects referred in this work can be in any shape, on a single layer, or on multiple layers. If a layout object cannot be decomposed, it is called primitive. Otherwise, it is called composite. The objects inside a composite one can be in different layers and may overlap. However, the primitive objects of the same type employed to form an array should be on the same layer. Without loss of generality, we assume a layout design consists of non-overlapped cell instances drawn from a set of rectangular primitive cell templates on the same layer. Each cell (instance) is stamped on its template name.

A mosaic array consists of mosaic cells, each of which contains more than one primitive cell as shown in Figure 1(b). An array's shape can be rectangular or rectilinear.

Definition 1: A *simple array* (ex. Figure 1(a)) consists of only primitive cells. A *mosaic cell template* is a cell template that consists of more than one primitive cell drawn from Ω which is formed by non-decomposable objects in a layout design. A mosaic array (ex. Figure 1(b)) consists of mosaic cells based on a certain mosaic cell template.

The shape of a mosaic cell template is the smallest bounding box that contains all of its primitive cells. Let $K(S)$ denote the cell templates of array S.

Definition 2: A *rectangular array* S of dimensions $D(S) = D(S)_x \times D(S)_y$ has $D(S)_x$ columns and $D(S)_y$ rows of the same kind of cells. The cells in a column (row) have the same $x(y)$ coordinate. The region $R(S)$ is the smallest bounding box that contains all the cells in S. The distance between any two adjacent columns (rows), called the *leap* of S in x (y) direction and denoted by $\ell(S)_x$ ($\ell(S)_y$), must be the same. Herein, $\ell(S)$ denotes the two leaps of array S. ℓ_x and ℓ_y denote the two leaps, respectively, if S itself is not of interest.

Definition 3: A *rabbeted array* S is an array whose region $R(S)$ may not be rectangular but rectilinear. Its dimensions are set equal to the dimensions of the largest rectangular array embedded in S. We will treat a rectangular array as a degenerated rabbeted array.

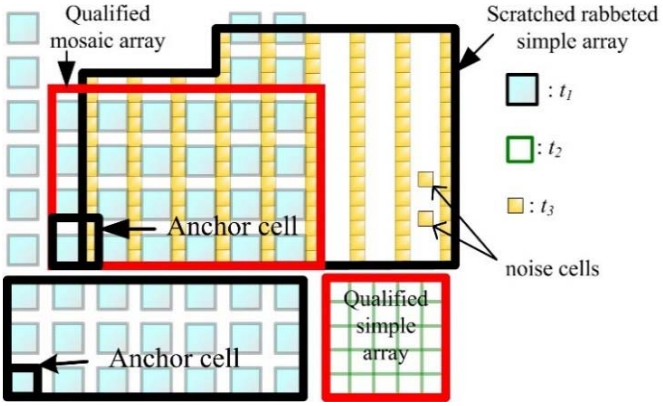


Figure 2: Various kinds of arrays.

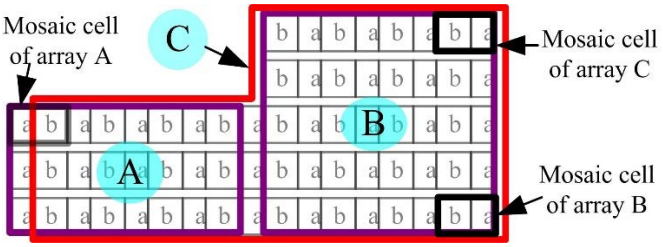


Figure 3: Problem of forming a rectangular mosaic array: C ≠ A∪B.

Definition 4: An anchor cell $\varphi(S)$ is the left-bottom most cell in array S.

Definition 5: A *noise cell* is a cell not belonging to any array.

Definition 6: A *scratched array* $S=(K(S),\varphi(S),\ell(S),R(S))$ is an array where noise cells or cells belonging to other arrays may be present in $R(S)$.

Definition 7: A *qualified array* $Q=(K(Q),\varphi(Q),\ell(Q),R(Q),E(Q))$ is an array where no cells belonging to other arrays are contained in $R(Q)$ and the region $E(Q)$ containing noise cells inside Q has been identified.

Here, two scratched arrays may overlap each other, but any two qualified arrays should not. The above definitions are also applied to simple arrays and mosaic arrays. We will use "array" to indicate a scratched, qualified, or rabbeted array if its meaning is clear from the discourse. Figure 2 shows different types of arrays where t_1, t_2, and t_3 are primitive cell templates. Each mosaic cell in the qualified mosaic array has one t_1 and three t_3's.

Array recognition problem:

Given a layout design containing rectangular objects, each of which is stamped on its template name, find all the qualified arrays Q_i in the design such that $D(Q_i)_x \geq 3$ and $D(Q_i)_y \geq 3$ for all i.

The challenge of this problem is due to not knowing mosaic cell templates a priori. It is further complicated by having to handle non-rectangular arrays. One way to deal with non-rectangular arrays is first to find rectangular arrays, then use them independently to form some mosaic arrays, and finally combine the mosaic arrays into a rectangular one. However, this approach does not work, as shown in Figure 3 where arrays A and B are not adjacent and do not have the same mosaic cell template so that they cannot be combined.

3. Array recognition algorithm

Our algorithm has three phases. The first phase identifies all scratched rectangular arrays (SRTAs) and combines adjacent SRTAs into scratched rabbeted arrays (SRBAs). The second phase finds the scratched mosaic arrays (SMAs) by intersecting SRBAs found in the first phase. The third phase qualifies an SMA by identifying noise cells. Figure 4 shows an example of our approach. We first find the SRTAs S_1, S_2, $S_{3'}$ and $S_{3''}$. We form SRBA B_3 by uniting $S_{3'}$ and $S_{3''}$ (Figure 4(b)). In the second phase, we find the region $R(B_1)\cap R(B_3)$ (Figure 4(c)). We tentatively assume that the intersected region will form SMA B_4. We use the intersected region to find out the anchor cell $\varphi(B_4)$ (Figure 4(d)). Adjusting the region boundaries based on $\varphi(B_4)$, we create SMA B_4 (Figure 4(e)). We then remove the cells now belonging to B_4. Finally, we qualify these scratched arrays to obtain Q_1, Q_2, Q_3 and Q_4 (Figure 4(f)).

Our algorithm relies on storing cells on R-B trees [9], creating an R-B tree T_t for all the cells referring to the same primitive cell template t as shown in Figure 5. In the following, we will elaborate on the three phases of the algorithm.

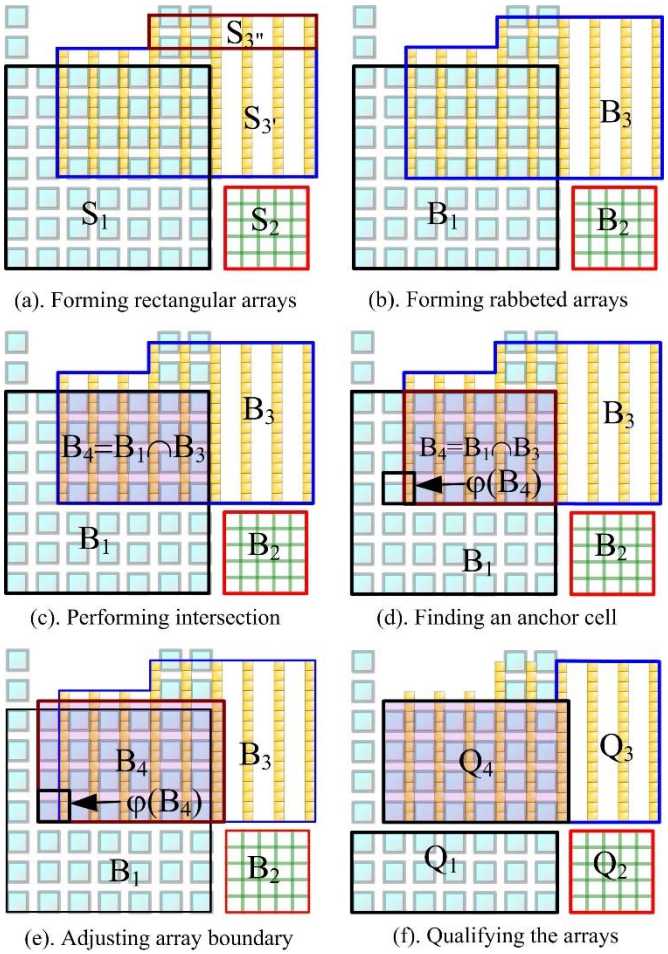


Figure 4: An example of array recognition.

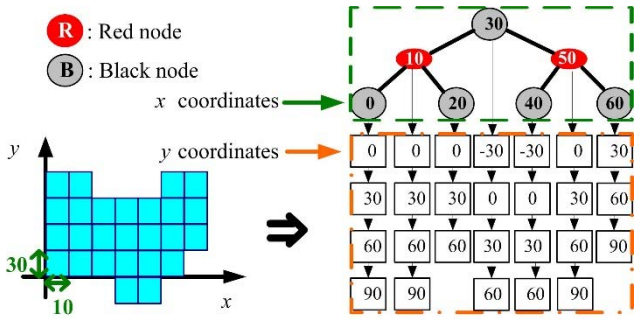


Figure 5: An R-B tree for cells using the same template.

```
//Input: An R-B tree T_t for the cells using template t.
//Output: Scratched rectangular arrays and noise cells
void SRTA_recognition( T_t ) {
(1)  while( T_t is not empty){
(2)      Find a tentative anchor cell φ(S) from T_t ;
     // S is the array yet to be formed with respect to (w.r.t.) φ(S) ;
(3)      Find the leap ℓ(S)_x w.r.t. φ(S) in T_t ;
(4)      Find the leap ℓ(S)_y w.r.t. φ(S) in T_t ;
(5)   D(S) = Dimension_Determination( T_t , ℓ(S) , φ(S) );
(6)   if( D(S)_x ≥ 3 or D(S)_y ≥ 3 ){
(7)   Set φ(S) to be the anchor cell of S;
(8)   Use φ(S) , ℓ(S) , and D(S) to determine R(S) ;
(9)   Create an R-B tree for S based on (K(S), φ(S), ℓ(S), R(S)) ;
(10)  Remove the cells of S from T_t ;}
(11)  else Remove φ(S) from the tree and treat it as a noise cell;
   } // end of while
   }
```

Figure 6: Scratched rectangular array recognition.

3.1. Scratched simple array recognition

The first phase, scratched simple array recognition process, has two steps. The first step finds out scratched rectangular arrays (SRTAs). The second step combines adjacent SRTAs into scratched rabbeted arrays (SRBAs).

3.1.1. Scratched rectangular array recognition

We process each R-B tree T_t to find out SRTAs formed by cells referring to cell template *t*. Let *S* be an SRTA yet to be found. We start with selecting a tentative anchor cell $\varphi(S)$, then find the leaps $\ell(S)$, and finally determine the dimensions $D(S)$. If *S* has dimensions greater than 3-by-3, we create an R-B-tree for it. The cells included in *S* are then deleted from T_t. Otherwise, $\varphi(S)$ is removed from T_t and treated as a noise cell. The above process is repeated until T_t is empty. Figure 6 shows such an algorithm which is elaborated below.

- *Determining tentative anchor cell*

The SRTAs embedded in an R-B tree can be in various forms, depending on how a tentative anchor cell is selected during a recognition process. A tentative anchor cell is tentatively served as the anchor cell of a yet-to-be recognized array. Figure 7 shows two different ways of forming SRTAs using c1 and c2 as tentative anchor cells. Although we obtain two different sets of rectangular arrays, we will get the same rabbeted array after rectangular arrays are combined. Here, we adopt the left-bottom most selection rule.

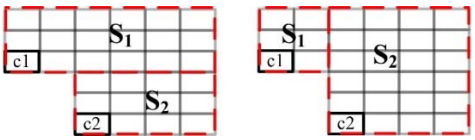


Figure 7: Tentative anchor cells.

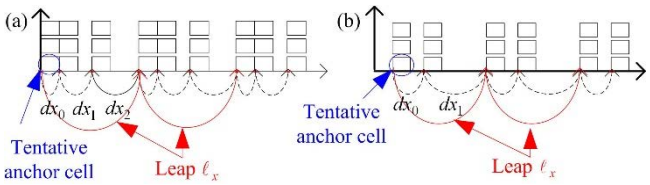


Figure 8: Determining leaps.

- *Determining leaps*

We use *jumps* to determine the leaps ℓ_x and ℓ_y of an array. A *jump* in *x* direction between two cells with the same *y* coordinate is the difference between their *x* coordinates. Starting from the tentative anchor cell, we can find jumps $dx_0, dx_1, dx_2, \ldots$, and dx_k as shown in Figure 8(a) where *k* is the smallest value that makes $dx_0 = dx_k$. We then define $\ell_x = \sum_{i=0}^{k-1} dx_i$, which is $dx_0 + dx_1 + dx_2$ for the example in Figure 8(a). ℓ_y can be determined similarly. Based on the leaps, the cells in columns 1, 4, and 7 of the example will form an SRTA with respect to the tentative anchor cell. Figure 8(b) shows the leaps for another array contained in the same layout. Given the tentative anchor cell with coordinates (0,0) in Figure 5, we have $\ell_x = 10$ and $\ell_y = 30$.

- *Determining dimensions of scratched rectangular array*

The algorithm in Figure 9 determines the dimensions $D(S)$ of an SRTA *S*. Let (x_a, y_a) be the coordinates of $\varphi(S)$. We start with the tree node from which $\varphi(S)$ can be reached (line 2) and search the linked-list pointed by the tree node to determine whether the cells in the linked-list can form a column of *S* (lines 7~10). If a column can be found (i.e., $temp\_Dy \neq 0$, line 11), we increase $D(S)_x$ by one and then move to the next adjacent tree node based on an in-order-traversal (the outer *while* loop). A column is found if we have a sequence of nodes with coordinates y_i, $i = 1, 2, \ldots, k$, so that $i \times \ell(S)_y = y_i - y_0$ and $y_0 = y_a$ hold where *k* is the smallest value that makes $k \times \ell(S)_y \neq y_k - y_0$ and y_a is the *y* coordinate of the tentative anchor cell.

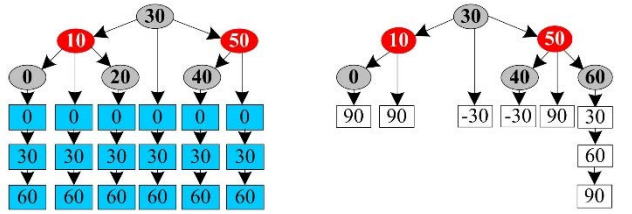
Given the R-B tree in Figure 5 as the input to *Dimension_Determination*(*), a 6-by-3 SRTA as shown in Figure 10(a) will be obtained. Figure 10(b) shows the R-B tree after the nodes that form the array are removed.

3.1.2. Forming scratched rabbeted arrays (SRBA)

After SRTAs have been identified, we combine any two adjacent SRTAs S_1 and S_2 into an SRBA if $K(S_1) = K(S_2)$, $\ell(S_1) = \ell(S_2)$, $(L(\varphi(S_1)) - L(\varphi(S_2))) \bmod \ell(S_1) = 0$ where $L(\varphi(S_1))$ and $L(\varphi(S_2))$ denotes the coordinates of $\varphi(S_1)$ and $\varphi(S_2)$, respectively. Array $B_3 = S_{3'} \cup S_{3''}$ in Figure 4(b) shows such an example.

```
//Input: T_t , ℓ(S) , φ(S)
//Output: D(S)
void Dimension_Determination ( T_t , ℓ(S) , φ(S) ) {
 // ( x_a , y_a ): the coordinates of the tentative anchor cell;
(1)  D(S)_{x=0} ; D(S)_y = ∞ ;
(2)  X_node = φ(S) ;
(3)  X_temp = x_a ; temp_Dy = 1 ; // y-dimension of an array
(4)  while( X_node ≠ NULL && temp_Dy ≠ 0 ){
(5)    Y_node = node pointed by X_node with y coord. equal to y_a ;
(6)    Y_temp=y_a ; temp_Dy = 0;
(7)    while( Y_node ≠ NULL ){ // till not repeated in every ℓ(S)_y .
(8)      temp_Dy = temp_Dy + 1 ;
(9)      Y_temp = Y_temp + ℓ(S)_y ;
(10)     Y_node = the node with its y coord. equal to Y_temp ; }
(11)   if ( temp_Dy ≠ 0 ){
(12)     if ( D(S)_y > temp_Dy ) D(S)_y = temp_Dy ;
(13)     X_temp = X_temp+ℓ(S)_x ;
(14)     X_node = the tree node with its x coord. equal to X_temp ;
(15)     D(S)_x = D(S)_x + 1 ; }
      }
(16)  return( D(S) );
    }
```

Figure 9: Determining array dimensions.



(a.) R-B tree for the recognized array (b). R-B tree for the remaining nodes

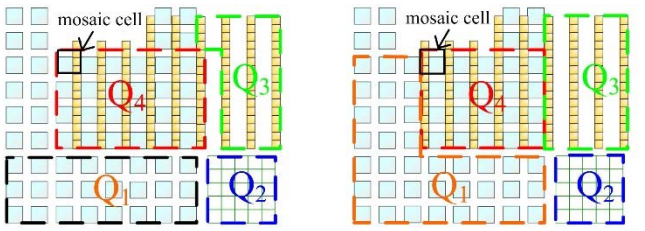
Figure 10: R-B trees generated after forming an array.

3.2. Scratched mosaic array (SMA) recognition

The second phase is to form a scratched mosaic array (SMA). This phase takes scratched rabbeted arrays (SRBAs) as input and perform geometric intersections among the SRBAs to find out SMAs. Performing such a geometric intersection is simple, but forming a mosaic cell template and determining the boundaries of the SMA are not trivial. A mosaic cell template may not be simply formed by combining the cell template of one SRBA with the cell template of the other SRBA. For example, the mosaic cell template of array Q_4 in Figure 4(f) is formed by one cell template from array B_1 plus three cell templates from B_3. The formation of a mosaic cell template also influences the size of a mosaic array as the two different Q_4's shown in Figure 11. Here we seek to find as a larger mosaic array as possible.

Figure 12 presents our algorithm for finding out SMAs. For each SRBA B_i, we start with putting the indices of all other SRBAs B_j into a linked list *LS* if $R(B_i) \cap R(B_j) \neq \varnothing$ for all $j > i$. The recursive calls to *Forming_SMAs*(*) (Figure 13) is performed for each index stored in *LS* to find out all possible SMAs. Basically, each call to *Forming_SMAs*(*) will return an SMA. All SMAs will be stored and then checked after finishing the *for* loop in Figure 12.

Figure 13 shows our algorithm for forming an SMA from any two SRBAs B_i and B_j. Clearly, most of the mosaic cells



(a). Forming a larger mosaic array (b). Forming a smaller mosaic array

Figure 11: Forming a larger or smaller mosaic array.

```
//Input: SRBAs B_i , i = 1,2,..., p
//Output: Modified SRBAs B_j , j = 1,2,..., q and SMAs M_k , k = 1,2,..., r
    void Finding_all_SMAs ( ) {
(1)   for( i = 1; i < p; i++ ){
(2)     Put the index j > i for each B_j into LS if R(B_i) ∩ R(B_j) != φ ;
      //LS is a linked list for storing indices of scratched rabbeted arrays.
(3)   ptr = head of LS;
(4)   while(ptr != NULL ){
(5)     Forming_SMAs ( B_i , B_{ptr->value} , LS , ptr );
(6)     ptr = ptr → next ; } // ptr → value is an array index
      } // B_i will be modified during the recognition process
(7)   Delete any B_i or M_k and treat the cells in B_i or M_k as noise cells if
      its dimensions are smaller than 3-by-3;
    }
```

Figure 12: Finding scratched mosaic arrays.

will lie in $R(M)$, but some mosaic cells may lie on the boundaries of $R(M)$. This requires some B_i's and B_j's cells, lying outside $R(M)$ but just next to $R(M)$, to be included in the mosaic cells. Therefore, in line 3 we perform *Find_reference_cells*(*) to find two reference cells in B_i and B_j respectively. These two reference cells will be employed to determine whether B_i'sand B_j's rows(columns) lying outside $R(M)$ should be included in an SMA. The reference cells are determined based on whether the two cells, one from B_i and the other from B_j,, closest to the left-bottom most corner $L(x_l, y_l)$ of $R(M)$ align in x or y directions. The details of this procedure is presented in Figure 14. Figures 16(e) and 16(f) show an example of how to determine the two reference cells when the two cells closest to $L(x_l, y_l)$ align in x direction.

Once the two reference cells are found, we call the function *Seed_of_φ*(*) twice (lines 4-5 in Figure 13). The first time is to find out from B_i a subarray A_i with its x and y dimensions not larger than $\ell(M)_x / \ell(B_i)_x$ and $\ell(M)_y / \ell(B_i)_y$, respectively. The second time is to find out a subarray A_j from B_j. Figure 15 shows the details of this procedure.

Once the two subarrays A_i and A_j are found, we use them to create $\varphi(M)$ (line 6 in Figure 13). Since the two subarrays are found independently, the smallest bounding box containing $\varphi(M)$, denoted by $BB(\varphi(M))$, may be larger than the region defined by the leaps $\ell(M)_x$ and $\ell(M)_y$. In this situation, we update either P_i or P_j to make P_i and P_j closer to each other to make $BB(\varphi(M))$ smaller. Lines 7-14 in Figure 13 carry out the above task. If the anchor cell $\varphi(M)$, also the mosaic cell template of M, cannot be established, remove the cells in the region $R(M)$ from B_i and B_j and treat them as noise cells (lines 15-17 in Figure 13). Conversely, we will create SMA M based on $\varphi(M)$ and remove the cells included in M

from B_i and B_j. We then pick an SRBA from *LS* and call *Forming_SMAs*(*) recursively to form a more complicate SMA using *M* and the SRBA picked from *LS*. The details are described in lines 18-25 in Figure 13.

Figure 16 shows an example of forming an SMA. We must point out that although we have drawn the traversing course of P_1 in Figure 16(f), this traversal is not actually performed for this example because the condition in line 7 of Figure 13 is false. The traversing course drawn there is intended to show how pointer P_1 should be moved if the condition in line 7 were true.

The time complexity of our array recognition algorithm is $O(2^p n^2)$, where n is the maximum number of cells in an SRBA and p is the maximum number of SRBAs that interweave each other. In practice, p is quite small.

3.3. Qualifying scratched arrays

The third phase is to exclude noise cells from a scratched (mosaic) array to create a qualified array $Q = (K(Q), \varphi(Q), \ell(Q), R(Q), E(Q))$. The identification process is easy because we need only check whether a non-array cell is located in the region $R(Q)$ and use this information to create a noise cell region $E(Q)$.

3.4. Limitations and extensions

Our algorithm has some limitations. First is about the choice of anchor cells. Although different choices normally end up with the same result, in some rare situations, the result may be different and thus finding maximum-sized qualified arrays is not guaranteed. Figure 17 shows such an example where anchor cells are determined using left-bottom most selection rule. Due to such a choice of anchor cells, simple array *A*, at its largest size, also consists of cells within the dotted region shown in Figure 17(a). As a result, we end up with simple array *B* also shown in Figure 17(a) and thus obtain a non-maximum-sized qualified array shown on the right of Figure 17 (b). Conversely, if we adopt the right-bottom most selection rule, the cells in the column where the dotted region resides will belong to simple array *B* and thus we will end up with two maximum-sized qualified arrays (not shown there). Second limitation is about the ordering problem of intersecting several SRBAs during forming an SMA. Different orderings may generate different solutions. The above two problems can be solved by running our algorithm with different anchor cell selection rules and different orderings, respectively.

Currently, our algorithm only recognizes arrays whose mosaic cells are rectangular and non-overlapped. However, as shown in Figure 18, a solution can only be found if mosaic cells are allowed to have a non-rectangular shape (Figure 18(a)) or be overlapped each other (Figure 18(b)). Although these two issues look differently, they are intrinsically the same. Our algorithm can be easily extended to deal with these two issues by simply allowing to have overlapped mosaic cells. For the current implementation, we only deal with non-overlapped layout objects, but extension of our algorithm to handle overlapped layout objects is straightforward.

```
//Input: SRBAs B_i and B_j, linked-list LS, pointer to LS ptr.
//Output: modified SRBAs B_i and B_j and SMAs M_k, k = 1,2,...,r
    void Forming_SMAs ( B_i, B_j, LS, ptr ){
          // Let M be the SMA yet to be recognized.
(1)   if( (R(M) = R(B_i) ∩ R(B_j)) = ∅ ) return;
(2)   ℓ(M) =Least_common_multiple( ℓ(B_i), ℓ(B_j) );
(3)   Find_reference_cells( B_i, B_j, R(M), P_i, P_j );
(4)   Seed_of_φ( B_i, ℓ(M), A_i, P_i, ℓ(M)_x / ℓ(B_i)_x, ℓ(M)_y / ℓ(B_i)_y );
(5)   Seed_of_φ( B_j, ℓ(M), A_j, P_j, ℓ(M)_x / ℓ(B_j)_x, ℓ(M)_y / ℓ(B_j)_y );
(6)   φ(M) = A_i ∪ A_j ;
    // create a tentative anchor cell (also a mosaic cell template) of M.
(7)   while( BB(φ(M)) > ℓ(M) ){
      // BB(φ(M)) : the smallest bounding box of φ(M)
(8)     if( A_i ==NULL || A_j ==NULL) break;
(9)     if ( P_i > P_j ){
(10)      Move P_j to point to the cell above the current one;
        // move to the next column if the top is hit.
(11)        Seed_of_φ( B_j , ℓ(M) , A_j , P_j , ℓ(M)_x / ℓ(B_j)_x ,
            ℓ(M)_y / ℓ(B_j)_y );
        } // Return A_j
(12)    else { Move P_i to point to the cell above the current one;
          // move to the next column if the top is hit.
(13)                Seed_of_φ( B_i , ℓ(M) , A_i , P_i , ℓ(M)_x / ℓ(B_i)_x ,
                    ℓ(M)_y / ℓ(B_i)_y ); } // Return A_i
(14)    φ(M) = A_i ∪ A_j ;
      } // end of while
(15)  if( A_i ==NULL || A_j ==NULL){
(16)  Remove cells in R(M) from B_i ( B_j ) and treat them as noise cells.
(17)  return; }
(18)  else{ Create M in terms of φ(M) ;
(19)    Remove all the cells being included in M from B_i and B_j ;
(20)    R(B_i) = R(B_i) − R(M) ; R(B_j) = R(B_j) − R(M) ;
(21)    ptr = ptr → next ; // Move to the next element in LS
(22)    while( ptr != NULL ){ // Forming a more complicate SMA.
(23)    Forming_SMAs ( B_{ptr→value}, M, LS, ptr );
(24)    ptr = ptr → next ; }
(25)    return; // Some SMAs M_k will be returned if they exist.
        } // end of else
    }
```

Figure 13: Forming scratched mosaic arrays using two SRBAs.

4. Experimental results

The experiments are run on a PC with 2.4 GHz Intel Core 2 Duo E6600 and 2GB main memory. Some results are shown in Table 1. Test cases *tc1*, *tc2*, *tc3*, *aa*, and *tlf* are obtained from a major EDA vendor. *t1f-d1*, *t1f-d2*, and *t1f-d3* are obtained from deleting an arbitrary number of cells from *t1f*, respectively. The rest of them are manually made to test the robustness of our method. *tlf* and *aa* are memory blocks. These test cases contain only flat layout objects described in CIF. The (mosaic) cell templates are not provided. The number of cells in *tlf* is about 9 times that of *ibma*, but it takes much less time for processing *tlf*. The reason for this is that *tlf* contains only a few large mosaic arrays whereas *ibma* contains a large number of small mosaic arrays. *t1f-d1*, *t1f-d2*, and *t1f-d3* takes more time than *t1f* because they are more

```
//Input: SRBAs B_i , B_j , and R(M)
//Output: Two reference cells P_i and P_j
//Let L(x_l, y_l) be the left-bottom most corner of R(M)
   void Find_reference_cells( B_i , B_j , R(M), P_i, P_j ) {
(1) Find a cell P_i(x_i, y_i) in B_i most closest to L(x_l, y_l) ;
(2) Find a cell P_j(x_j, y_j) in B_j most closest to L(x_l, y_l) ;
(3) if( x_i == x_j ){ // aligning in y direction
(4)   if(there is a cell of B_i at position( x_i , y_i − ℓ(B_i)_y ))
(5)   make P_i point to this cell;
(6)   if(there is a cell of B_j at position( x_j , y_j − ℓ(B_j)_y ))
(7)   make P_j point to this cell;
   }
(8) else if ( y_i == y_j ){ // aligning in x direction
(9)   if(there is a cell of B_i at position( x_i − ℓ(B_i)_x , y_i ))
(10)  make P_i point to this cell;
(11)  if(there is a cell of B_j at position( x_j − ℓ(B_j)_x , y_j ))
(12)  make P_j point to this cell;
   }
(13) else{ // not aligning in x and y directions
(14)  if(there is a cell of B_i at position( x_i − ℓ(B_i)_x , y_i − ℓ(B_i)_y ))
(15)  make P_i point to this cell;
(16) if(there is a cell of B_j at position( x_j − ℓ(B_j)_x , y_j − ℓ(B_j)_y ))
(17) make P_j point to this cell;
   }
   }
```

Figure 14: Finding reference cells.

```
//Input: B_k, ℓ(M), A_k, P_k, D_x, D_y .
//Output: A_k, P_k
     void Seed_of_φ ( B_k, ℓ(M), A_k, P_k, D_x, D_y ){
(1)    Starting from P_k and traversing the R-B tree of B_k , just like what
       has been done for finding a SRTA, to find out a D_x x D_y
       subarray A_k from B_k with BB(A_k) ≤ ℓ(M) .
(2)    if ( A_k is not found) {
(3)     A_k =NULL;
(4)     P_k =NULL;
(5)    }
(6)    else set P_k pointing to the anchor cell of A_k ;
   }
```

Figure 15: Finding a subarray A_k from B_k.

irregular. The last column in Table 1 gives the number of cell instances that are not included in any qualified array. A zero means that all cells are contained in the arrays.

For small test cases, we visually inspect their layout designs and find that our algorithm finds all the qualified arrays in each of the test cases. Our algorithm also finds all the qualified arrays in *ibme*. Even though *tlf* is large, we can also visually verify that our algorithm find all the qualified arrays of largest dimensions. We believe that our algorithm also does a good job for *ibma* even though it is too complicate to be visually inspected. Figure 19 shows some qualified arrays for some test cases.

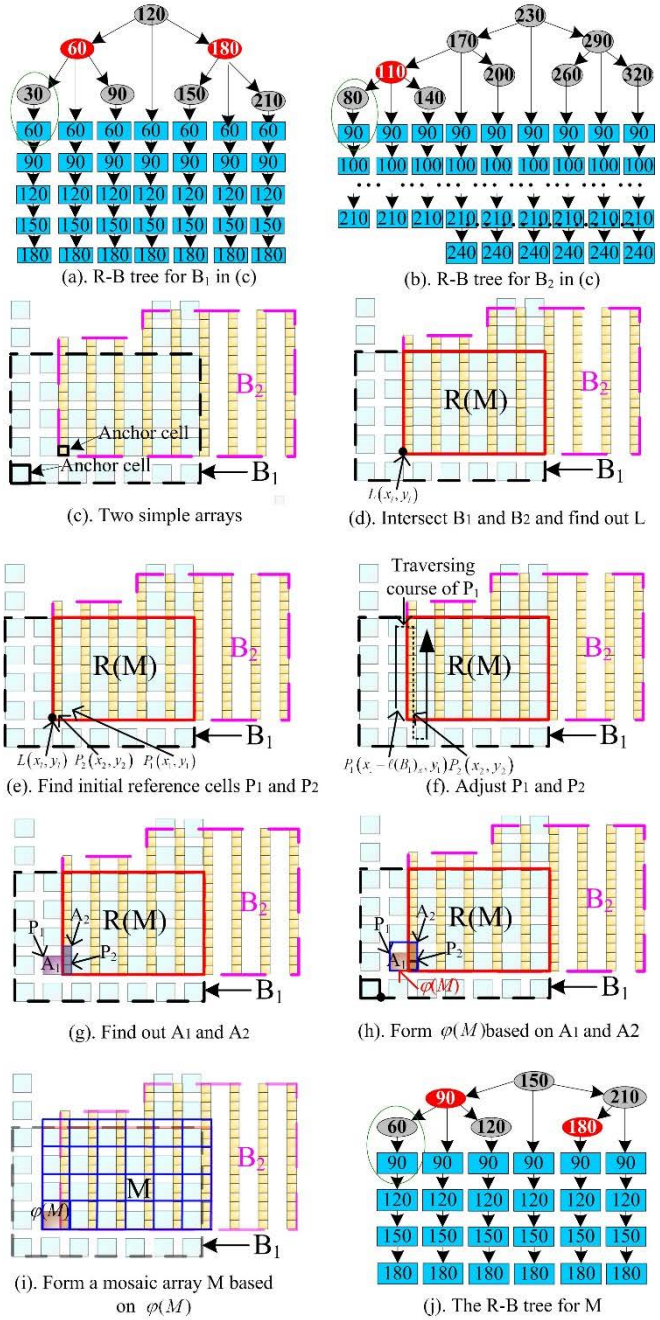


Figure 16: An example of forming an SMA.

A comparison to the previous works is not possible due to no related works found in the open literature. A comparison to Cadence Darcula with ARRAY-ENABLED is possible, but this cannot be done because the layout designs given by the EDA vendor consist of only empty cells, i.e., the layout objects inside the cells have been removed. ARRAY-ENABLED command is not executed if cells contain no layout objects. Also note that Dracula recognizes only simple arrays. By the same reason, we are not able to run our tests using array recognition capability provided by Assura.

5. Conclusions

In this paper, we present an efficient approach to finding regular layout structures in a design. Our implementation finds all the qualified arrays if mosaic cells are rectangular and non-overlapped. It can be easily extended to find out

arrays from a design where mosaic cells are overlapped or non-rectangular.

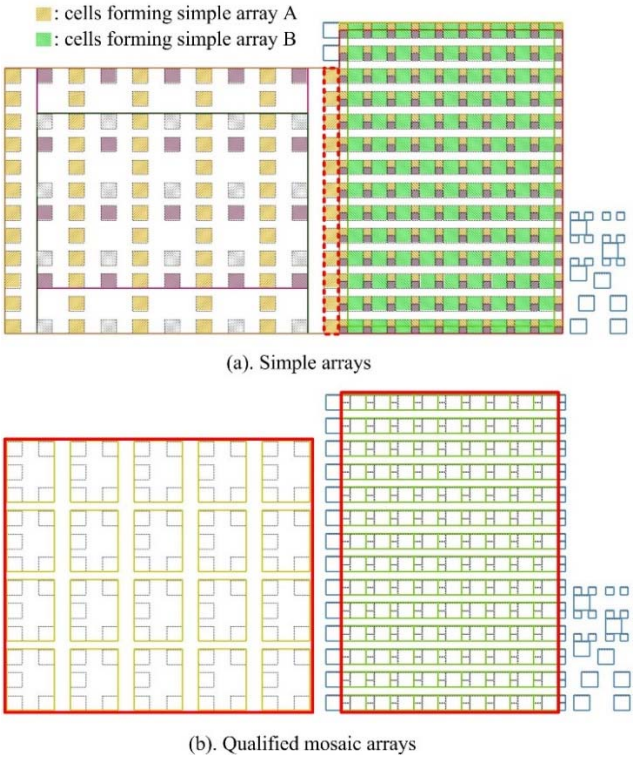


Figure 17: Non-maximum qualified arrays due to choice of anchor cells.

Table 1: Results and run time.

Name	# cells	#PC	# SRBAs	# SMAs	# QAs	Run time (sec.)	# non-array cells
tlf	614144	3	7	7	8	66	128
aa	2000	1	1	0	1	1	0
tc1	170	2	3	1	2	1	20
tc2	147	2	4	1	2	0	32
tc3	121	2	3	2	2	0	1
t1f-d1	550824	3	7	7	9	162	552
t1f-d2	445874	3	7	6	10	102	7730
t1f-d3	395854	3	6	6	11	85	548
tc4	540	2	6	2	2	1	62
tc6	135	1	4	1	2	0	20
tc7	384	2	2	1	1	0	0
A1	200	2	2	1	3	0	0
A2	1700	2	2	1	2	0	0
A3	800	3	3	2	3	0	30
A4	450	2	2	1	1	0	30
a2	64000	1	1	0	1	1	0
fl	736	3	3	1	2	0	0
ffffl	1186	4	4	2	4	0	0
fff	57	1	1	0	1	0	18
flr	144	3	4	1	1	0	0
ibme	68478	95	8	2	5	8	66196
ibma	69000	1	775	713	713	492	6900
sp	39400	2	3	2	4	5	0

#PC: number of primitive cell templates, found out by the algorithm.

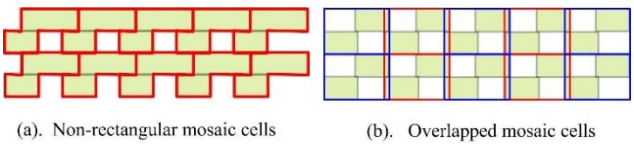


Figure 18: Extensions of the proposed algorithm.

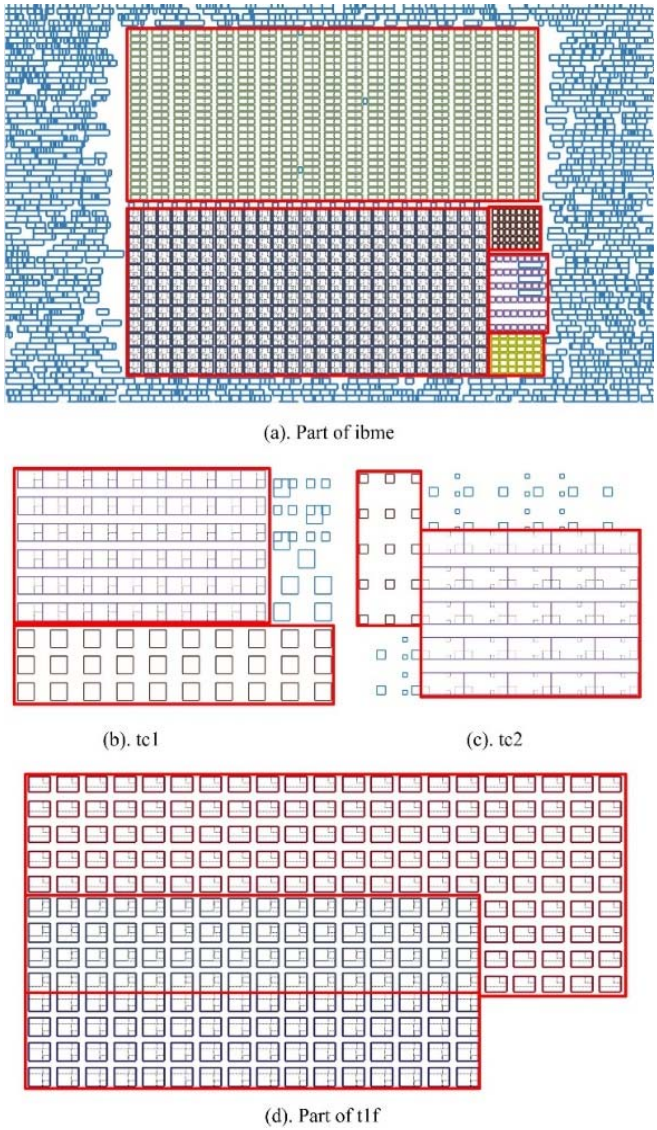


Figure 19: Qualified arrays found by our algorithm.

6. References

[1] K. C. Wu, Y. W. Tsai, "Structured ASIC, Evolution or Revolution?" ISPD, 2004, pp.103-106.

[2] G. S. Taylor and J. K. Ousterhout, "Magic's Incremental Design-Rule Checker," Proc. of the 21st DAC, 1984, pp. 160 -165.

[3] M. H. Arnold and J. K. Ousterhout, "Lyra: A New Approach to Geometric Layout Rule Checking," Proc. 19th DAC, 1982, pp. 530-536.

[4] T. Whitney, "A Hierarchical Design Checker," Caltech CSTR: 4320-tr-81, 1981.

[5] Cadence Inc. Dracula Reference Manual, Product Version 4.9.1205, Dec. 2005.

[6] Cadence Inc. Assura Physical Verification Command Reference, Product Version 3.1.7, June 21, 2007.

[7] C. A. Mack, "30 Years of Lithography Simulation," in Proc. SPIE, B. W. Smith, ed., vol. 5754, 2005, pp. 1-12.

[8] M. Niewczas, W. Maly, and A. Strojwas, "An Algorithm for Determining Repetitive Patterns in Very Large IC Layouts," IEEE Trans. on CAD, Vol. 18, No. 4, 1999, pp. 494-501.

[9]. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms. 2nd Edition, McGraw-Hill, New York, 2001.