[14] J. A. Waicukauski, E. Lindbloom, B. K. Rosen, and V. S. Iyengar, "Transition fault simulation," *IEEE Des. Test*, vol. 4, pp. 32–38, Apr. 1987.
[15] J. Richman and K. R. Bowden, "The modern fault dictionary," in *Proc. Int. Test Conf.*, Sept. 1985, pp. 696–702.
[16] P. G. Ryan, S. Rawat, and W. K. Fuchs, "Two-stage fault location," in *Proc. Int. Test Conf.*, Oct. 1991, pp. 963–968.

# On Extending Slicing Floorplan to Handle L/T-Shaped Modules and Abutment Constraints

F. Y. Young, M. D. F. Wong, and Hannah H. Yang

***Abstract*—In floorplanning, it is common that a designer wants to have certain modules abutting with one another in the final packing. The problem of controlling the relative positions of an arbitrary number of modules in floorplan design is nontrivial. Slicing floorplan has an advantageous feature in which the topological structure of the packing can be found without knowing the module dimensions. This feature is good for handling placement constraints in general. In this paper, we make use of it to solve the abutment problem in the presence of L- and T-shaped modules. This is done by a procedure which explores the topological structure of the packing and finds the neighborhood relationship between every pair of modules in linear time. Our main contribution is a method that can handle abutment constraints in the presence of L- or T-shaped modules in such a way that the shape flexibility of the soft modules can still be fully exploited to obtain a tight packing. We tested our floorplanner with some benchmark data and the results are promising.**

***Index Terms*—Abutment constraints, floorplanning, rectilinear-shaped modules, slicing floorplan.**

## I. Introduction

Floorplanning is an important step in physical design of VLSI circuits. It is the problem of placing a set of circuit modules on a chip to optimize the circuit performance. Besides optimizing the packing area and interconnect cost, it is common that the designers will want to impose some placement constraints on the final packing for different purposes. For example, a designer may want to have several logic modules in a circuit to abut one after another to favor the transmission of data between them. This abutment problem is common in practice, but few floorplanning algorithms can handle it. The problem of controlling the relative positions of an arbitrary number of modules in floorplan design is nontrivial.

One objective of our work is to handle abutment constraints in floorplanning. Our work is based on a particular type of floorplan called slicing floorplan. A slicing floorplan is one that can be obtained by recursively cutting a rectangle into two parts by either a vertical line or a horizontal line. There are several advantages of using slicing floorplan. First, focusing only on slicing floorplan significantly reduces the search space, which in turn leads to a faster runtime. Second, the shape flexibility of the soft modules can be fully exploited to give a tight packing based on an efficient shape curve computational technique [9], [11]. It has been shown mathematically that a tight packing is achievable [14] for slicing floorplan if the modules are flexible in shape. Slicing floorplan has another advantageous feature: we can find the topological structure of the packing efficiently without knowing the module dimensions. This feature is good for handling placement constraints in general because we can check and fix the constraints given the topological information.

We have also made used of this feature to handle rectilinear blocks. Because of the recent advance in the semiconductor manufacturing technology, new packaging schemes such as multichip modules (MCMs) and integrated circuit components often have their shapes more complex than a simple rectangle. A lot of works have been reported on placement of rectilinear blocks [1]–[8], [10], [12], [13], but none of them can handle rectilinear blocks with soft modules efficiently. Slicing floorplan is well known to be effective in handling soft modules. It is not obvious how L- or T-shaped modules can be handled because of the nature of slicing floorplan that the regions inside must be rectangular in shape. In our work, we treat an L- or T-shaped module like a rectangular one, but it will *be expanded* to its original shapes when being packed. Again, we will explore the topological structure of the packing and expand the modules accordingly.

Our main contribution is a method that can handle abutment constraints in the presence of L- or T-shaped modules in such a way that the shape flexibility of the soft modules can still be fully exploited to obtain a tight packing. We tested our floorplanner using some benchmark data. The experiments give very promising results. The rest of the paper is organized as follows. We will first define the problem formally in Section II. Section III describes our method to handle abutment constraints with L- and T-shaped modules. Experimental results will be given in Section IV.

## II. Problem Definition

We consider three kinds of modules $M = M_R \cup M_L \cup M_T$, where $M_R$ is a set of rectangular modules, $M_L$ is a set of L-shaped modules, and $M_T$ is a set of T-shaped modules. A rectangular module $A$ is a rectangle of *height* $h(A)$ and *width* $w(A)$. The *aspect ratio* of $A$ is defined as $h(A)/w(A)$. A rectangular module can either be *hard* or *soft*. The height and width of a hard module are fixed, but the module is free to rotate. The shape of a soft module can be changed as long as the area remains a constant and the aspect ratio is within a given range. An L-shaped module $B$ [see Fig. 1(a)] consists of two rectangular submodules $B_1$ and $B_2$, where $w(B_1)$ and $w(B_2)$ are aligned and $h(B_1) > h(B_2)$. A T-shaped module $C$ [see Fig. 1(b)] consists of three rectangular submodules $C_1$, $C_2$, and $C_3$, where $w(C_1)$, $w(C_2)$, and $w(C_3)$ are aligned and $h(C_1) > \max\{h(C_2), h(C_3)\}$. We assume that all the T- and L-shaped modules are hard modules.

In general, two modules $A$ and $B$ are said to be abutting horizontally (see Fig. 2), denoted by $H\mathrm{abut}(A, B)$, if a vertical boundary $L_A$ of module $A$ and a vertical boundary $L_B$ of module $B$ abut such that $L_A$ lays immediately on the left of $L_B$ and the length of the abutment is at least $\min\{\mathrm{len}(L_A), \mathrm{len}(L_B)\}$, where $\mathrm{len}(L_A)$ is the length of $L_A$ and $\mathrm{len}(L_B)$ is the length of $L_B$. The abutment in the vertical direction is defined similarly.

A floorplan for $n$ modules is a dissection of a rectangle by horizontal and vertical lines into $n$ nonoverlapping regions such that each region must be large enough to accommodate the module assigned to it. A *packing* is a nonoverlap placement of all the modules in $M$. A *feasible*

Manuscript received March 28, 2000; revised November 18, 2000. This paper was recommended by Associate Editor C.-K. Cheng.

F. Y. Young is with the Department of Computer Science and Engineering, the Chinese University of Hong Kong, New Territories, Hong Kong (e-mail: fyyoung@cse.cuhk.edu.hk).

M. D. F. Wong is with the Department of Computer Sciences, the University of Texas at Austin, Austin, TX 78712 USA (e-mail: wong@cs.utexas.edu).

H. H. Yang is with the Intel Corporation, Hillsboro, OR 97124 USA (e-mail: hyang@ichips.intel.com).

Publisher Item Identifier S 0278-0070(01)03538-2.

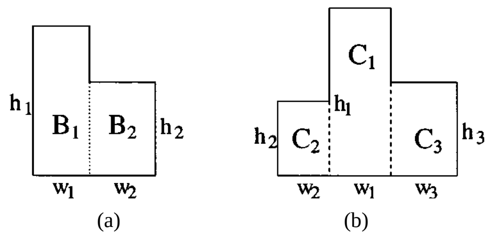
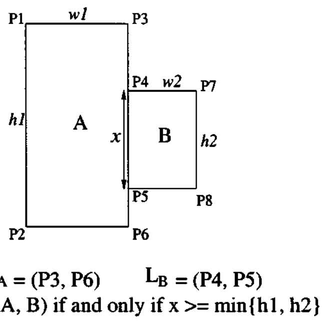


Fig. 1. (a) L-shaped module and (b) T-shaped module.



P1
w1
P3
P4
w2
P7
h1
A
x
B
h2
P5
P8
P2
P6
LA = (P3, P6) LB = (P4, P5)
Habut(A, B) if and only if x >= min{h1, h2}

Fig. 2. Abutment example.

*packing* is a packing such that all the abutment constraints are satisfied and the widths and heights of all the soft modules are consistent with their aspect ratio constraints and area constraints. Our objective is to construct a feasible packing $F$ to minimize $A + \lambda W$, where $A$ is the total area of the packing, $W$ is an estimation of the interconnect cost, and $\lambda$ is a user-specified constant that controls the relative importance of $A$ and $W$ in the cost function. We require that the aspect ratio of the final packing is between two given numbers: $r_{\min}$ and $r_{\max}$.

## III. Slicing Floorplan

A slicing floorplan can be represented by an oriented rooted binary tree called a slicing tree. Each internal node of the tree is labeled by a * or a + operator, corresponding to a vertical or a horizontal cut, respectively. Each leaf corresponds to a basic module and is labeled by a number from 1 to $n$. No dimensional information on the position of each cut is specified in the slicing tree. If we traverse a slicing tree in postorder, we obtain a *Polish expression.* A Polish expression is said to be *normalized* if there is no consecutive *'s or +'s in the sequence. It is proved in [12] that there is a 1:1 correspondence between the set of normalized Polish expressions of length $2n - 1$ and the set of slicing floorplans with $n$ modules. Our method is developed based on the simulated annealing algorithm in [12]. In [12], the set of all normalized Polish expressions is used as the solution space. In order to search the solution space efficiently, they defined three types of moves (M1, M2, and M3) to transform a Polish expression into another. They can make use of the flexibility of the soft modules to select the "best" floorplan among all those represented by the same slicing structures. This is done by carrying out an efficient shape curve computation whenever a Polish expression is examined. The cost function is $A + \lambda W$, where $A$ is the total packing area and $W$ is the interconnect cost.

## IV. Our Approach

### A. Overview

The algorithm *Main* below outlines our method. In each step of the annealing process, we consider a particular Polish expression. We will scan the expression once to find out the topological structure of the packing and, in particular, the neighborhood relationship between

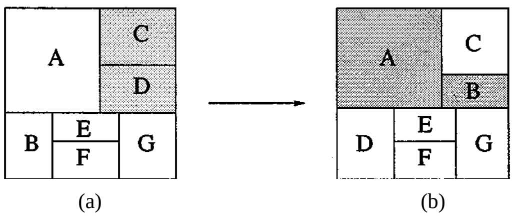


Fig. 3. Shuffling modules to obtain a feasible packing. (a) Both $C$ and $D$ are right neighbors of A. (b) A and B abut horizontally.

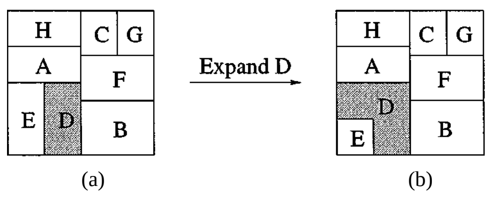
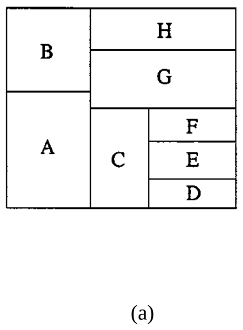


Fig. 4. Example of module expansion. $D$ is L-shaped. (a) Initial packing. (b) Packing obtained after expanding module $D$.



| Module | Left | Right | Top | Bottom |
|---|---|---|---|---|
| A | x | C | B | x |
| B | x | H | x | A |
| C | A | D,E,F | G | x |
| D | C | x | E | x |
| E | C | x | F | D |
| F | C | x | G | E |
| G | x | x | H | C,F |
| H | B | x | x | G |

x : no known neighbor in that direction

(b)

Fig. 5. Abutment between modules. (a) Packing corresponding to expression $AB + CDE + F +^{*} G + H+^{*}$. (b) Neighborhood information.

every pair of modules. This is possible because the operators + and * in a Polish expression have orientations, e.g., $AB+$, means that $A$ is below $B$ and $AB*$ means that $A$ is on the left of $B$. We will scan the expression once to mark the left, right, top, and bottom neighbors of every module. Fig. 5 shows a simple example in which the neighbors of every module are marked in a table after this step. We will then shuffle the modules to satisfy as many abutment constraints as possible. Please refer to Fig. 3 as an example. In this example, module $A$ is constrained to abut with module $B$ horizontally, i.e., $H\text{abut}(A, B)$, but this constraint is violated in the original packing [see Fig. 3(a)]. After finding the neighborhood information between all pairs of modules, we will shuffle $B$ with a closest right neighbor of $A$, i.e., module $D$ in this example, to obtain a similar packing [see Fig. 3(b)], which satisfies the constraint. After this shuffling step, the abutting modules will stay together unless some later moves break them apart.

After fixing the abutment constraints, we will *expand* the L- or T-shaped modules into their original shapes. This is done by modifying the Polish expression to embed the submodules of the rectilinear blocks in such a way that the relative positions between all the modules in the original Polish expression are preserved. Please refer to Fig. 4 as an example. In this example, module $D$ is L-shaped and the initial packing is shown in Fig. 4(a). We will expand $D$ to its original shape before computing the total area and interconnect cost. The packing after expansion is shown in Fig. 4(b). After expansion, we can do the shape curve computation as usual to obtain the total area of the final floorplan and the flexibility of the soft modules can be fully exploited. We will describe the steps in details in the following sections.

```
Algorithm Main
Input: The size, shape and interconnection of a set
       of modules $M = M_R \cup M_L \cup M_T$, where $M_R$ is
       a set of rectangular modules, $M_L$ is a set
       of L-shaped modules and $M_T$ is a set of
       T-shaped modules; a set of horizontal abut-
       ment constraints and a set of vertical abut-
       ment constraints.
Output: A feasible packing of the modules in $M$
1. Initialization.
2. Repeat:
3.   Transform the Polish expression $\alpha_{\mathrm{old}}$ to $\alpha$.
4.   Scan $\alpha$ to find the neighbors of every module.
5.   Modify $\alpha$ to $\alpha_{\mathrm{new}}$ by shuffling modules to fix the
     violated abutment constraints.
4.   Expand the L- or T-shaped modules in $\alpha_{\mathrm{new}}$
     to obtain a new Polish expression $\beta$.
5.   Calculate the total area and interconnect cost
     of the floorplan represented by $\beta$.
6.   Decide whether to accept $\alpha_{\mathrm{new}}$. If yes, $\alpha_{\mathrm{old}} = \alpha_{\mathrm{new}}$.
7. Until $\mathrm{Cost} < k$.
```

## B. Handling Abutment Constraints

*1) Finding the Neighbors of a Module:* We can find the neighbors of a module from the Polish expression because the operators in the expression have orientations, e.g., $AB+$ means that $A$ is below $B$ and $AB*$ means that $A$ is on the left of $B$. These topological relationships are independent of the dimensions of the modules. For example, Fig. 5 is a packing corresponding to the expression $AB + CDE + F + *G + H + *$. We can tell from the Polish expression the neighborhood relationship as shown in Fig. 5(b). This information can be obtained by scanning the expression once and update the table whenever an operator is seen, i.e., when two subfloorplans are combined by either a $+$ operator (vertical cut) or a * operator (horizontal cut). The algorithm *Neighbor* below outlines the step to find this neighborhood information. Notice that the variables $L[X], R[X], T[X]$, and $B[X]$ denote the set of modules lying along the left boundary, right boundary, top boundary, and bottom boundary of a subfloorplan $X$. Consider combining two subfloorplans $X$ and $Y$ horizontally as in $XY*$. If both $R[X]$ and $L[Y]$ have more than one modules, the top module in $R[X]$ will abut horizontally with the top module in $L[Y]$ and the bottom module in $R[X]$ will abut horizontally with the bottom module in $L[Y]$. We will explain using the example in Fig. 5. When we combine the subfloorplan containing $A$ and $B$ and the subfloorplan containing $C, D, E, F, G$, and $H$ by the * operator, we know that $B$ will abut with $H$ horizontally and $A$ will abut with $C$ horizontally. Notice that we do not know whether $G$ will abut with $A$, $B$, or both because this is dependent on the dimensions of the modules, so we will not indicate anything about the abutment of $G$ in Fig. 5(b). However, if either $R[X]$ or $L[Y]$ has only one module, every module in $R[X]$ will abut with every module in $L[Y]$ horizontally. For example, in Fig. 5, when we combine the subfloorplan containing $C$ and the subfloorplan containing $D, E$ and $F$ by the* operator, we know that $C$ will abut with $D, E$ and $F$ horizontally. Similarly, we can derive the vertical neighborhood relationship from the $+$ operator.

```
Algorithm Neighbor
Input:  A Polish expression $\alpha = \alpha_1\alpha_2\ldots\alpha_{2n-1}$
Output: For each module $A$, find the modules abut-
        ting with $A$ in all four directions.
1. For $i = 1$ to $2n - 1$:
2.   If $\alpha_i$ is a module name:
       $L[\alpha_i] = R[\alpha_i] = T[\alpha_i] = B[\alpha_i] = \alpha_i$
3.     Push $\alpha_i$.
4.   If $\alpha_i$ is a * operator:
5.     Pop Y. Pop X.
6.     If $R[X]$ or $L[Y]$ has only one module:
7.       $H\mathrm{abut}[A, B]$ is true for all $A \in R[X]$ and
         $B \in L[Y]$;
8.     Else:
9.       $H\mathrm{abut}[A_1, B_1]$ and $H\mathrm{abut}[A_2, B_2]$ are true
         where $A_1$ and $A_2$ are the top and bottom
         modules in $R[X]$ resp., and $B_1$ and $B_2$ are
         the top and bottom modules in $L[Y]$ resp.
10.    $R[\alpha_i] = R[Y],\ L[\alpha_i] = L[X]$,
       $T[\alpha_i] = T[X] + T[Y],\ B[\alpha_i] = B[X] + B[Y]$.
11.    Push $\alpha_i$.
12.  If $\alpha_i$ is a $+$ operator:
13.    Pop Y. Pop X.
14.    If $T[X]$ or $B[Y]$ has only one module:
15.      $V\mathrm{abut}[A, B]$ is true for all $A \in T[X]$ and
         $B \in B[Y]$;
16.    Else:
17.      $V\mathrm{abut}[A_1, B_1]$ and $V\mathrm{abut}[A_2, B_2]$ are true
         where $A_1$ and $A_2$ are the left and right
         modules in $T[X]$, resp., and $B_1$ and $B_2$
         are the left and right modules in $B[Y]$,
         resp.
18.    $T[\alpha_i] = T[Y],\ B[\alpha_i] = B[X]$,
       $R[\alpha_i] = R[X] + R[Y],\ L[\alpha_i] = L[X] + L[Y]$.
19.    Push $\alpha_i$.
```

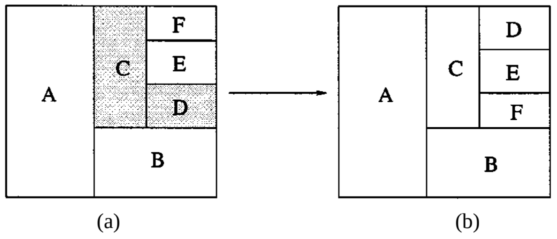


Fig. 6. Shuffling modules to fix violated abutment constraints. (a) Initial packing. (b) Packing obtained after shuffling module $D$ with module $F$.

*2) Shuffling Modules to Fix Violated Abutment Constraints:* If a Polish expression does not satisfy all the abutment constraints, we can fix it as much as possible by shuffling the modules. An example is shown in Fig. 6. In this example, assume that module $B$ is required to abut with $F$ vertically, i.e., $V\mathrm{abut}(B, F)$, but it is violated initially as shown in Fig. 6(a). We will then try to move $F$ to the top of $B$ or move $B$ to the bottom of $F$. In the first case, $B$ has two neighbors at the top: $C$ and $D$. Since $F$ is closer to $D$ than to $C$ in the Polish expression, we will shuffle $F$ and $D$ in order to fix this violated constraint. In general, if an abutment constraint $V\mathrm{abut}(X, Y)$ is violated, we will first try to move $Y$ to the top of $X$ by shuffling $Y$ with the closest top neighbor of $X$ in the Polish expression. If it is failed, e.g., all the top neighbors of $X$ are fixed in position, we will try to move $X$ to the bottom of $Y$ by shuffling $X$ with the closest bottom neighbor of $Y$. The procedure for abutment in the horizontal direction is defined similarly. Notice that we will not shuffle the modules back to their original positions if an expression is accepted, i.e., the constrained modules will stay together unless some later moves break them apart.

It is possible that some constraints are still violated after all the possible shufflings. We include a penalty term in the total cost to penalize the violated constraints. All violations will be eliminated as the annealing process proceeds in most of the cases.

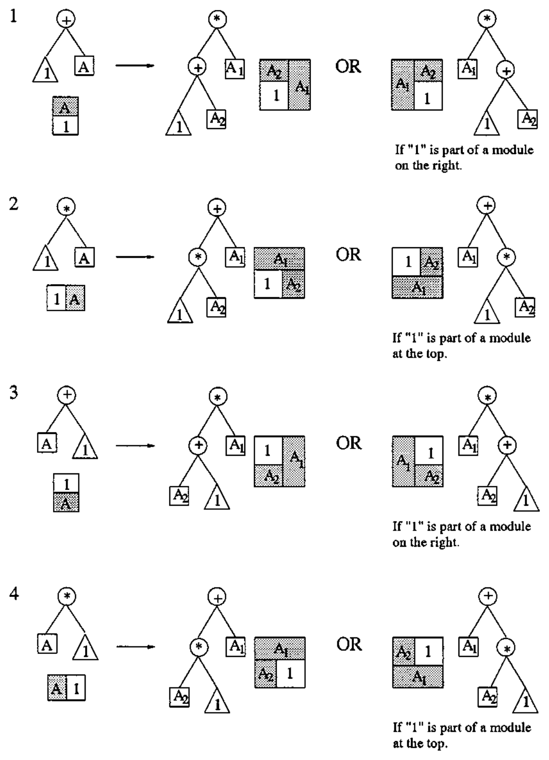

1
2
3
4
OR
If "1" is part of a module on the right.
If "1" is part of a module at the top.
If "1" is part of a module on the right.
If "1" is part of a module at the top.

Fig. 7. Expansion of an L-shaped module.

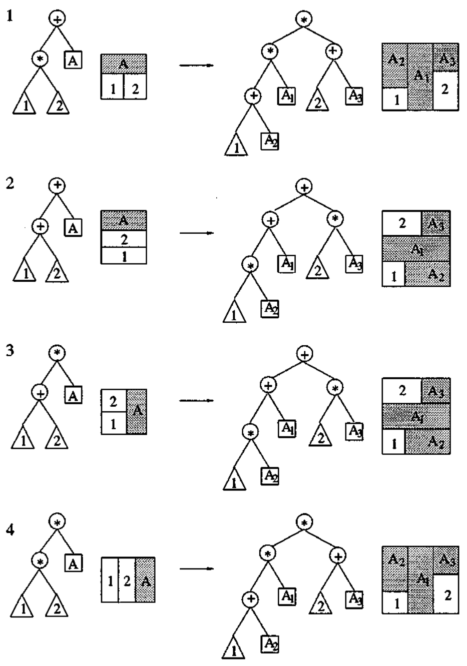


Fig. 8. Expansion of a T-shaped module, which is a right child.

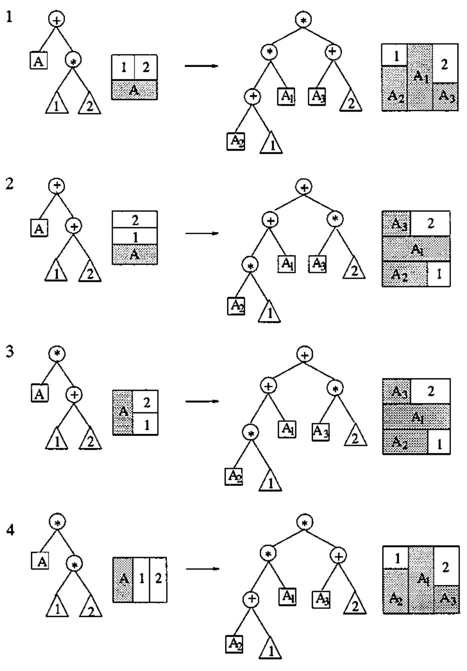


Fig. 9. Expansion of a T-shaped module, which is a left child.

## C. Handling L- and T-Shaped Modules

Instead of partitioning into rectangular submodules, L- and T-shaped modules are treated as single modules in the annealing process. They will be *expanded* to their original shapes when being packed and the expansions are dependent on their topological positions in the original Polish expression. After calculating the total area and interconnect cost, they are treated as single modules again in the floorplan transformation.

*1) Expansion of L-Shaped Modules:* Consider an L-shaped module $X$ in a Polish expression $\alpha$. We will expand it into its submodules $X_1$ and $X_2$ by modifying the expression according to the relative position of $X$ in $\alpha$. There are four different cases as shown in Fig. 7. The subtree labeled "1" can either be a basic module or a subtree of modules. We are trying to pack modules into the unoccupied area of the L-shaped modules. The L-shaped module is oriented differently in different cases so as to preserve as much as possible the relative position between all the other modules in the original Polish expression.

*2) Expansion of T-Shaped Modules:* Similar to an L-shaped module, we will expand a T-shaped module $X$ into its submodules $X_1$, $X_2$, and $X_3$ by modifying the Polish expression $\alpha$ according to the relative position of $X$ in $\alpha$. There are two different cases, depending on the sibling $u$ of $X$ in the slicing tree. If $u$ is an internal node and the two children subtrees of $u$ are not parts of the same module, we will pack the submodules of $X$ with the children subtrees of $u$ as shown in Figs. 8 and 9. The subtree labeled "1" or "2" can either be a basic module or a subtree of modules. Again, we are trying to pack modules into the two unoccupied areas of the T-shaped modules, which is oriented differently in different cases to preserve as much as possible the relative positions between all the other modules in the original Polish expression. If $u$ is a single basic module or the two children subtrees of $u$ belong to the same module (so we cannot pack them apart as shown in Figs. 8 and 9), we will label $C$ as a *degenerated* T-shaped module, which will be expanded into its submodules as described in Fig. 10.

*3) Expansion Order:* The result of the expansion will depend on the order in which the modules are expanded. An example is shown in Fig. 11. Assume that both module $A$ and $B$ in the figure are L-shaped. Expanding $B$ followed by $A$ will give us the packing in (a), while expanding in the reverse order will give us the packing in (b). If the order is not defined well, we may need to scan the Polish expression once for each L- or T-shaped module and this will be very time consuming. In our implementation, we will first expand the T-shaped modules. This requires scanning the expression twice. The first scan expands all the T-shaped modules that are right children, and the second scan expands all the T-shaped modules that are left children. The degenerated T-shaped modules are labeled on the way. After these two scans, any T-shaped module will either be expanded or labeled as degenerated.

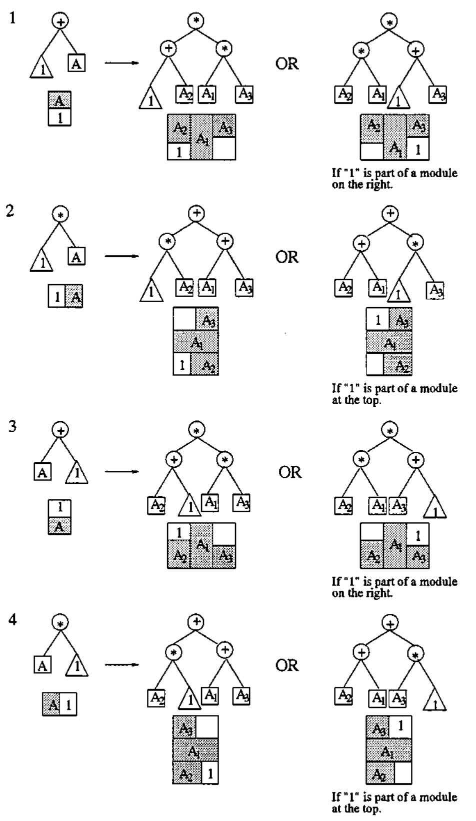

1
OR
If "1" is part of a module on the right.
2
OR
If "1" is part of a module at the top.
3
OR
If "1" is part of a module on the right.
4
OR
If "1" is part of a module at the top.

Fig. 10. Expansion of a degenerated T-shaped module.

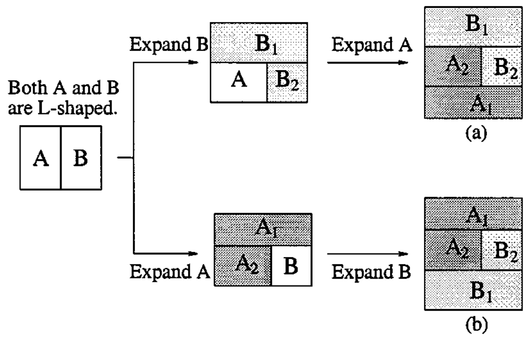


Fig. 11. Example demonstrating the effect of the expansion order. (a) Packing obtained by expanding $B$ followed by $A$. (b) Packing obtained by expanding $A$ followed by $B$.

We will then expand the remaining L-shaped modules and the degenerated T-shaped modules. This also requires scanning the expression twice. The first scan expands all the L-shaped modules or degenerated T-shaped modules that are right children and the second scan expands those that are left children. The algorithm is described by the algorithm *Expansion* below. We need to scan the expression four times in total. An example of expansion is shown in Fig. 12. In this example, module $A$ is T-shaped and module $B$ is L-shaped. $A$ is expanded first because it is a T-shaped module and a right child. After that, we should expand the T-shaped modules that are left children followed by the L-shaped modules that are right children, but there is none of them. Finally, we will expand $B$, which is an L-shaped module and a left child.

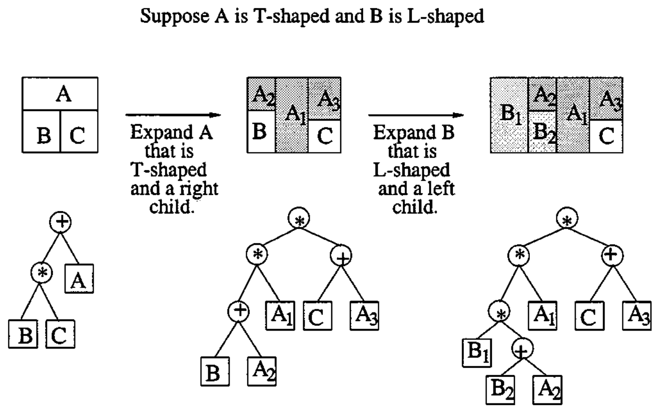

Suppose A is T-shaped and B is L-shaped
Expand A that is T-shaped and a right child.
Expand B that is L-shaped and a left child.

Fig. 12. Example of the expansion step.

```
Algorithm Expansion
Input: A Polish expression $\alpha$ of a set of modules
       $M = M_R \cup M_L \cup M_T$, where $M_R$ is a set of
       rectangular modules, $M_L$ is a set of L-shaped
       modules and $M_T$ is a set of T-shaped modules.
Output: A Polish expression $\beta$ with all the modules
       in $M_L$ and $M_T$ expanded to their corre-
       sponding submodules.
1. Scan $\alpha$ from left to right and generate a new
   Polish expression $\alpha_1$:
2.  For any T-shaped module $X$ that is a right
    child:
3.   If the sibling $u$ of $X$ is an internal node and
     the two children subtrees of $u$ are not parts
     of one module:
4.    Expand $X$ as described in Fig. 8
5.   Else:
6.    Label $X$ as a degenerated T-shaped module.
7. Scan $\alpha_1$ from left to right and generate a new
   Polish expression $\alpha_2$:
8.  For any T-shaped module $X$ that is a left child:
9.   If the sibling $u$ of $X$ is an internal node and
     the two children subtrees of $u$ are not parts
     of one module:
10.    Expand $X$ as described in Fig. 9;
11.   Else:
12.    Label $X$ as a degenerated T-shaped module.
13. Scan $\alpha_2$ from left to right and generate a new
    Polish expression $\alpha_3$:
14.  For any L-shaped module or degenerated
     T-shaped module $X$ that is a right child:
15.   If $X$ is an L-shaped module:
16.    Expand $X$ as in case 1–2 of Fig. 7;
17.   Else:
18.    Expand $X$ as in case 1–2 of Fig. 10.
19. Scan $\alpha_3$ from left to right and generate a new
    Polish expression $\beta$:
20.  For any L-shaped module or degenerated
     T-shaped module $X$ that is a left child:
21.   If $X$ is an L-shaped module:
22.    Expand $X$ as in case 3–4 of Fig. 7;
23.   Else:
24.    Expand $X$ as in case 3–4 of Fig. 10.
25. Output $\beta$.
```

## D. Time Complexity

We need to scan the Polish expression once to find the neighbors of every module. This takes $O(n)$ time, where $n$ is total number of modules. Then shuffling modules to fix violated abutment constraints takes another $O(nq)$ time, where $q$ is the total number of abutment

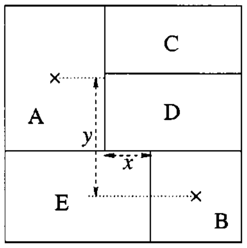


Fig. 13. Penalty for violation of an abutment constraint.

TABLE I
RESULTS OF TESTING ABUTMENT CONSTRAINTS WITH RECTANGULAR MODULES

| Data Set | n | #Constraints | %Deadspace | Time(s) | #Violation |
|---|---|---|---|---|---|
| ami33a | 33 | 12 | 3.62 (0.75) | 84.98 (79.45) | 0 (0) |
| ami49a | 49 | 12 | 2.02 (0.94) | 164.51 (136.36) | 0.2 (0) |
| playout | 62 | 12 | 2.93 (1.49) | 453.32 (432.69) | 1.4 (0) |

constraints. Notice that this is only a worst-case analysis. Usually, we do not need to scan all the modules once to find the closest module to shuffle with and the average time taken is just $O(q + n)$. To expand all the L- and T-shaped modules, we need to scan the expression four times and it takes $O(n)$ time. Therefore, the total time taken in each iteration of the annealing process to handle the abutment constraints and rectilinear blocks is $O(nq)$ in the worst case and $O(n + q)$ on the average.

## *E. Moves and Cost Function*

We use the same set of moves (M1, M2, and M3) as in [12]. The cost function is defined as $A + \lambda W + \gamma D$, where $A$ is the total packing area obtained from the shape curve at the root of the slicing tree, $W$ is a half perimeter estimation of the interconnect cost, and $D$ is a penalty term for the violated abutment constraints. If an abutment constraint between two modules are violated, the corresponding penalty term is computed as the manhattan distance that one of the two module centers needs to be moved in order to make them abut. An example is shown in Fig. 13. In this example, suppose $A$ and $B$ are constrained to abut horizontally, i.e., $H\text{abut}(A, B)$, but this constraint is violated and its corresponding penalty term $D$ will be $x + y$, where $x$ is the distance between the right boundary of $A$ and the left boundary of $B$ and $y$ is the vertical distance between the centers of $A$ and $B$. The penalty term is computed similarly in the case of L- or T-shaped modules by just considering the largest submodules in the rectilinear blocks. $\lambda$ and $\gamma$ are constants which control the relative importance of the three terms. $\lambda$ is usually set such that the area term and the interconnect term are approximately balanced. We usually set $\gamma$ large to ensure that all the abutment constraints can be satisfied.

## V. EXPERIMENTAL RESULTS

We tested our floorplanner with some benchmark data: ami33, ami49, and playout. In all the data, the rectangular modules are soft modules with aspect ratios lying between 0.25 and 4.0 and the L- or T-shaped modules are hard modules. For each experiment, the starting temperature is decided such that an acceptance ratio is 100% at the beginning. The temperature is lowered at a constant rate and the number of iterations in one temperature step is proportional to the number of modules. All the experiments were carried out on a 143-MHz UltraSPARC workstation.

We did two sets of experiments, one set with only rectangular modules and the other set with L- d or T-shaped modules. In the first set, we did five testings for each benchmark data, each testing with a different set of abutment constraints. The abutment constraints we imposed are usually that chains of four to five modules are required to abut horizontally or vertically. The averaged result for each benchmark data is shown in Table I. (The best values are shown in brackets.) We can compare these results with Table II, which shows the results of the control experiments in which there is no abutment constraint nor L/T-shaped module in the data sets. We can see from the tables that our method

TABLE II
RESULTS OF CONTROL EXPERIMENTS WITHOUT ANY CONSTRAINT NOR L/T-SHAPED MODULE

| Data Set | n | %Deadspace | Time(s) |
|---|---|---|---|
| ami33a | 33 | 0.59 | 77.89 |
| ami49a | 49 | 0.42 | 174.94 |
| playout | 62 | 1.72 | 413.61 |

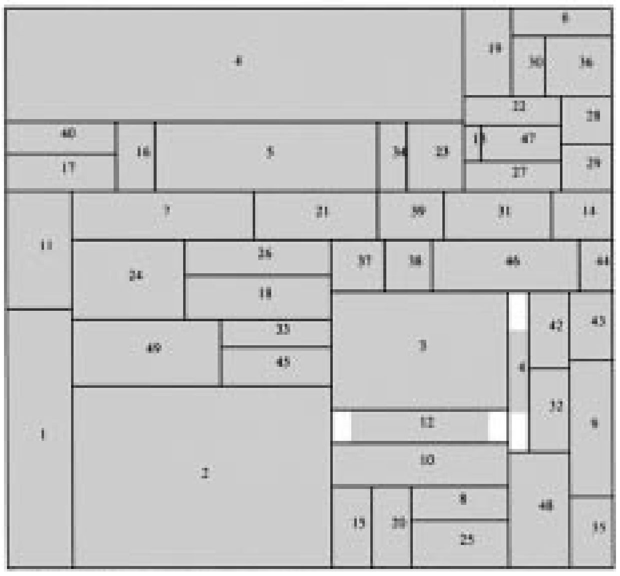


Fig. 14. Result packing of ami49. Modules 1, 2, 15, 20, and 25; 3, 41, 42, and 43 are required to abut horizontally. Modules 25, 8, 10, 12, and 3; 43 and 44 are required to abut vertically. All constraints are satisfied.

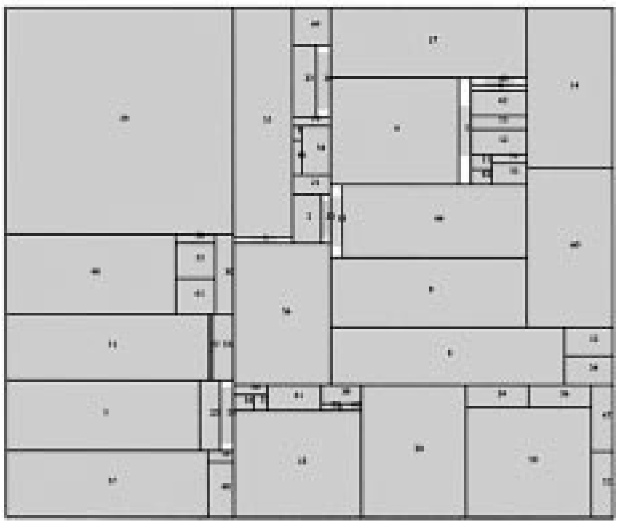


Fig. 15. Result packing of playout. Modules 1, 2, 50, 4, 5, 6, and 14 are required to abut horizontally and modules 8, 9, 10, 4, 11, 12, and 13 are required to abut vertically. Ten out of the 12 abutment constraints are satisfied.

TABLE III
RESULTS OF TESTING ABUTMENT CONSTRAINTS WITH L- AND T-SHAPED MODULES

| Data Set | n | #Constraints | #L-Blocks | #T-Blocks | %Deadspace | Time(s) | #Violation |
|---|---|---|---|---|---|---|---|
| lt-ami33a | 33 | 12 | 2 | 1 | 5.20 (3.79) | 78.49 (70.43) | 1 (0) |
| lt-ami49a | 49 | 12 | 3 | 2 | 5.38 (2.64) | 294.34 (276.26) | 1 (0) |
| lt-playout | 62 | 12 | 3 | 3 | 3.91 (2.90) | 807.89 (532.34) | 1.4 (1) |

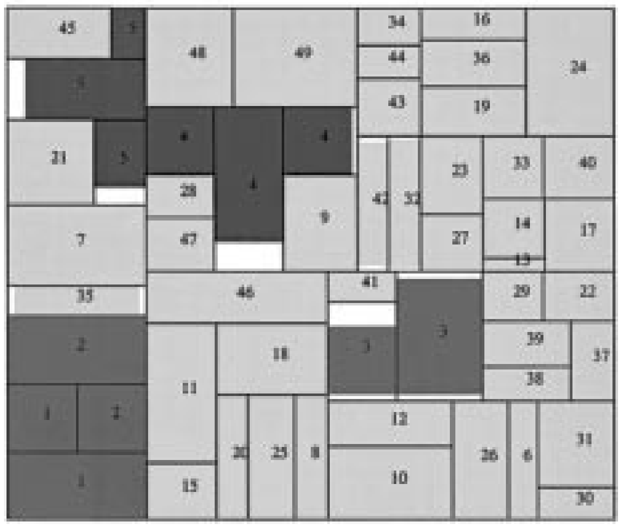


Fig. 16. Result packing of lt-ami49. Modules 1, 2, 15, 20, 25, 8, and 10 are required to abut horizontally and modules 10, 12, 3, 41, 42, 43, and 44 are required to abut vertically. 11 out of the 12 abutment constraints are satisfied.

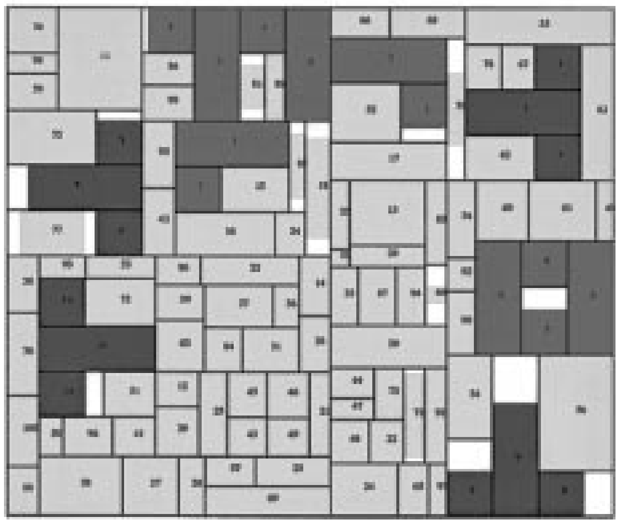


Fig. 18. Result packing of a randomly generated example without abutment constraint. There are 100 modules, six of which are L-shaped and four of which are T-shaped.

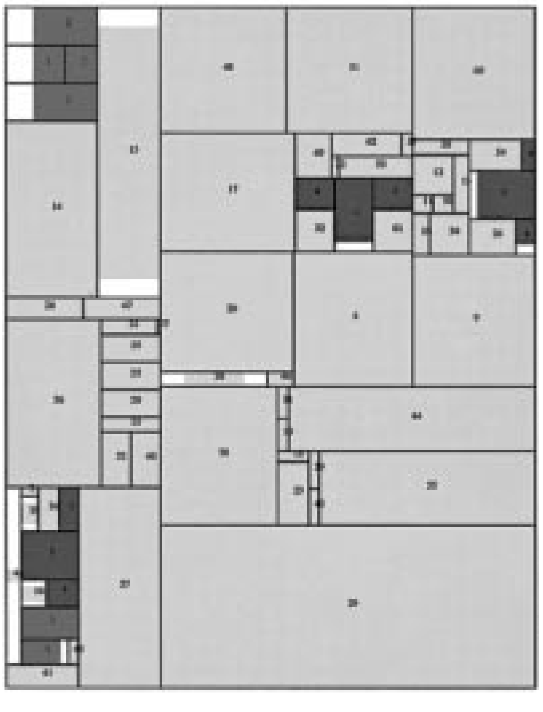


Fig. 17. Result packing of lt-playout. Modules 1, 2, and 15; 8 and 9; 12, 13, 14 and 15 are required to abut horizontally. Modules 9, 10, 11 and 12; 20, 17 and 48; 18 and 19 are required to abut vertically. 11 out of 12 of the abutment constraints are satisfied.

can handle abutment constraints efficiently. Figs. 14 and 15 show two result packings.

In the second set of experiments, we modified the benchmark data by changing some modules to L- or T-shaped. We called these data lt-ami33, lt-ami49, and lt-playout. Again, we did five testings for each data, imposing different abutment constraints on the modules for each testing. Table III summarizes the results. Figs. 16 and 17 show two result packings. The rectangular modules are white in color, the L-shaped modules are light gray, and the T-shaped modules are dark gray. Fig. 18 shows a result packing of a randomly generated example without any abutment constraint. It has 100 modules in total with four T-shaped modules and six L-shaped modules.

## VI. CONCLUDING REMARKS

We devised a method that can handle abutment constraints in the presence of L- or T-shaped modules in such a way that the shape flexibility of the soft modules can be fully exploited to obtain a tight packing. We made use of an advantageous feature in slicing floorplan to exploit the topological structure of a packing without knowing the module dimensions. We tested our floorplanner with some benchmark data and the results are good. However, our method can only handle L- and T-shaped modules. It is interesting to extend it to handle arbitrarily shaped rectilinear modules, but the expansion process of which will be quite complicated.

## REFERENCES

[1] J. Dufour, R. McBride, P. Zhang, and C. K. Cheng, "A building block placement tool," in *Proc. IEEE Asia South Pacific Design Automation Conf.*, Jan. 1997, pp. 271–276.

[2] K. Fujiyoshi and H. Murata, "Arbitrary convex and concave rectilinear block packing using sequence-pair," in *Proc. Int. Symp. Physical Design*, Apr. 1999, pp. 103–110.

[3] M. Kang and W. W. M. Dai, "General floorplanning with L-shaped, T-shaped and soft blocks based on bounded slicing grid structure," in *Proc. IEEE Asia South Pacific Design Automation Conf.*, Jan. 1997, pp. 265–270.

[4] M. Z. W. Kang and W. W. M. Dai, "Topology constrainted rectilinear block packing," in *Proc. Int. Symp. Physical Design*, Apr. 1998, pp. 179–186.

[5] M. Z. Kang and W. W.-M. Dai, "Arbitrary rectilinear block packing based on sequence pair," in *Proc. IEEE Int. Conf. Computer-Aided Design*, Nov. 1998, pp. 259–266.

[6] T. C. Lee, "A bounded 2D contour searching algorithm for floorplan design with arbitrarily shaped rectilinear and soft modules," in *Proc. 30th ACM/IEEE Design Automation Conf.*, June 1993, pp. 525–530.

[7] S. Nakatake, K. Fujiyoushi, H. Murata, and Y. Kajitani, "Module placement on BSG-structure and IC layout applications," in *Proc. IEEE Int. Conf. Computer-Aided Design*, Nov. 1996, pp. 484–491.

[8] S. Nakatake, M. Furuya, and Y. Kajitani, "Module placement on BSG-structure with preplaced modules and rectilinear modules," in *Proc. IEEE Asia South Pacific Design Automation Conf.*, Jan. 1998, pp. 571–576.

[9] R. H. J. M. Otten, "Efficient floorplan optimization," in *Proc. IEEE Int. Conf. Computer Design*, 1983, pp. 499–502.

[10] K. Sakanushi, S. Nakatake, and Y. Kajitani, "The Multi-BSG: Stochastic approach to an optimal packing of convex-rectilinear blocks," in *Proc. IEEE Int. Conf. Computer-Aided Design*, Nov. 1998, pp. 267–274.

[11] L. Stockmeyer, "Optimal orientations of cells in slicing floorplan designs," *Inform. Control*, vol. 59, pp. 91–101, 1983.

[12] D. F. Wong and C. L. Liu, "A new algorithm for floorplan design," in *Proc. 23rd ACM/IEEE Design Automation Conf.*, June 1986, pp. 101–107.

[13] J. Xu, P.-N. Guo, and C. -K. Cheng, "Rectilinear block placement using sequence-pair," in *Proc. Int. Symp. Physical Design*, Apr. 1998, pp. 173–178.

[14] F. Y. Young and D. F. Wong, "How good are slicing floorplans?," *Integr. VLSI J.*, vol. 23, no. 1, pp. 61–73, Oct. 1997.