

**Phoenix:**  
An  
Interactive Hierarchical Topological Floorplanning Placer

by Chee-Seng Chow

Submitted to the

Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements

for the Degrees of

Electrical Engineer,

Master of Science in Electrical Engineering and Computer Science,

Bachelor of Science in Physics,

Bachelor of Science in Computer Science,

and

Bachelor of Science in Electrical Engineering

at the


Massachusetts Institute of Technology

June 1985

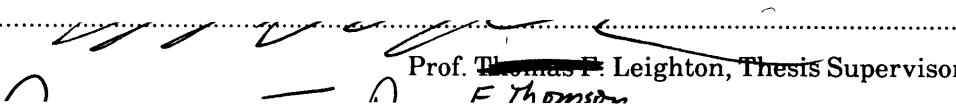
© Chee-Seng Chow 1985

The author hereby grants to MIT permission to reproduce and to distribute copies of this thesis document in whole or in part.

Signature of Author.....

  
Department of Electrical Engineering and Computer Science, 10 May 1985

Certified by.....

  
Prof. ~~Thomas F.~~ Leighton, Thesis Supervisor

Certified by.....

  
Dr. Bryan T. Preas, Company Supervisor

Accepted by.....

Prof. Arthur C. Smith

Chairman, Departmental Committee on Graduate Students



**Phoenix:**  
**An**  
**Interactive Hierarchical Topological Floorplanning Placer**

by Chee-Seng Chow

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the Degrees of

Electrical Engineer,

Master of Science in Electrical Engineering and Computer Science,

Bachelor of Science in Physics,

Bachelor of Science in Computer Science,

and

Bachelor of Science in Electrical Engineering

at the

Massachusetts Institute of Technology

June 1985

**Abstract**

Phoenix is an interactive, hierarchical, custom integrated circuits placement program used for designing very large scale integrated circuits. It has the following novel features:

- Ability to handle arbitrary-sized rectangular blocks with some of the corners removed.
- A complete and extensible set of operations on the model.
- Efficient tracking and backtracking support for these operations.

Underlying Phoenix is a topological layout model. Based on this layout model, powerful search heuristics, using the tracking and backtracking capabilities, have been developed to address both top-down (floorplanning and shape determination) and bottom-up (initial placement and placement improvement) placement problems. When combined with a topological router, this topological approach ensures routing completion of all interconnection nets.

This thesis describes the topological model and the layout heuristics developed, shows the results obtained, and discusses future research based on Phoenix.

Thesis Supervisor: Prof. Frank T. Leighton

Company Supervisor: Dr. Bryan T. Preas

## Preface

This thesis describes the topological model and the layout heuristics developed so that interested readers may duplicate the work and experiment with the ideas. The work was part of my MIT VI-A Internship program with the Xerox Palo Alto Research Center (PARC). My mandate was to develop a topological model for the VLSI placement problem that can handle arbitrarily sized rectangular blocks with some corners removed.

### Acknowledgments

I acknowledge the contribution of many people and the influence of three earlier works [4, 7, and 10] on the results reported in this thesis.

I thank the following researchers at the Computer System Laboratory (CSL) of PARC for their help and suggestions: Bryan Preas, my supervisor, for introducing me to the problem and helping me solve it; without his guidance, encouragement, and support this work would never have been successful. Russ Atkinson, for helping me through a number of major implementation hurdles and contributing many elegant implementation ideas. Christian Jacobi for his advice on building large systems and help in interfacing to Chipndale, the circuit design system. Also my thanks to CSL for their excellent programming and research environment and to PARC for providing the funds and resources.

I also thank the following individuals at MIT without whose kindness and support this thesis may not have been completed on time: Prof. Leighton, my thesis advisor, for giving me a theoretical perspective of circuit layout. Prof. Malone for providing me with support and facilities while I was writing my thesis. Prof. Rivest for introducing me to the placement problem, four years ago, when I worked for him on the *PI (Placement and Interconnect) System* [9] project. Marie Rountree, my technical writing instructor, whose meticulous readings and

critical comments significantly improve the style and presentation of this thesis. Mr. Tucker and the MIT VI-A Office for running the program so efficiently. And my friends, Whay Chiou Lee, William Tiger Lee, and Pui Ng, for reading initial drafts of this thesis and contributing many invaluable improvements.

Lastly, but not the least, I thank my family for the sacrifices they made to ensure that I get the best education available.

*C.S.C*

*May 1985*

## List of Figures

<b>Figure 2-1:</b> <i>Placing a block</i>	8
<b>Figure 2-2:</b> <i>Partial ordering of CQ1</i>	10
<b>Figure 2-3:</b> <i>Blocks and channels</i>	12
<b>Figure 2-4:</b> <i>Topological holes</i>	13
<b>Figure 2-5:</b> <i>Different geometries with the same topology</i>	14
<b>Figure 2-6:</b> <i>Three possible types of channel intersections</i>	15
<b>Figure 2-7:</b> <i>A conceptual view of Topologize</i>	18
<b>Figure 2-8:</b> <i>Different ways of forming outlines</i>	19
<b>Figure 2-9:</b> <i>Different ways of forming channels</i>	20
<b>Figure 2-10:</b> <i>MakeOutlines routine</i>	21
<b>Figure 2-11:</b> <i>MakeChannels routine</i>	22
<b>Figure 2-12:</b> <i>Horizontal channel position graph</i>	24
<b>Figure 2-13:</b> <i>MakeBot2TopCPG routine</i>	26
<b>Figure 2-14:</b> <i>Block constraints</i>	27
<b>Figure 2-15:</b> <i>Topological constraints</i>	28
<b>Figure 2-16:</b> <i>PositionChannelsBot2Top routine</i>	28
<b>Figure 2-17:</b> <i>Simple operations</i>	32
<b>Figure 2-18:</b> <i>Grow and shrink operations</i>	36
<b>Figure 2-19:</b> <i>ClearInteriorKnee operation</i>	37
<b>Figure 2-20:</b> <i>ClearExteriorKnee operation</i>	37
<b>Figure 2-21:</b> <i>Twenty possible corner intersections for CQ1</i>	40
<b>Figure 2-22:</b> <i>Generalized slack</i>	45
<b>Figure 2-23:</b> <i>Some restricted grow operations using pseudo-channels</i>	46
<b>Figure 2-24:</b> <i>A topological hole formed from pseudo-channels only</i>	46
<b>Figure 2-25:</b> <i>Using fake blocks to shape corners of an assembly</i>	49
<b>Figure 2-26:</b> <i>Using fake block to augment Topologize</i>	50
<b>Figure 3-1:</b> <i>Paths between two points in a search space</i>	56
<b>Figure 3-2:</b> <i>Local search algorithm</i>	57
<b>Figure 3-3:</b> <i>Improve routine</i>	57
<b>Figure 3-4:</b> <i>Conceptual view of local search</i>	58

<b>Figure 3-5:</b> <i>BestImprove routine</i>	58
<b>Figure 3-6:</b> <i>LookaheadImprove routine</i>	59
<b>Figure 3-7:</b> <i>Look-ahead cost function</i>	60
<b>Figure 3-8:</b> <i>Bounded look-ahead cost function</i>	61
<b>Figure 3-9:</b> <i>Multi-path search (depth-first)</i>	62
<b>Figure 3-10:</b> <i>Conceptual view of multi-path search (depth-first)</i>	63
<b>Figure 3-11:</b> <i>Five types of shape constraints</i>	66
<b>Figure 3-12:</b> <i>Using generalized slack as hints for shape determination</i>	67
<b>Figure 3-13:</b> <i>Min-cut algorithm</i>	69
<b>Figure 3-14:</b> <i>Generating a floorplan from a min-cut tree</i>	70
<b>Figure 3-15:</b> <i>Four different ways of splitting a topological hole</i>	70
<b>Figure 3-16:</b> <i>Using BreakCross and FormCross for placement improvement</i>	72
<b>Figure 3-17:</b> <i>Estimating net length</i>	74
<b>Figure 4-1:</b> <i>Two different loose routes of a two-pin net</i>	80
<b>Figure 4-2:</b> <i>Loose routes of a three-pin net</i>	81
<b>Figure 4-3:</b> <i>Four possible topologies of a three-pin net</i>	82
<b>Figure 4-4:</b> <i>T-order constraint</i>	84
<b>Figure 4-5:</b> <i>Channel routing order conflict loop</i>	85
<b>Figure 4-6:</b> <i>Generalized T-order constraint</i>	86
<b>Figure 4-7:</b> <i>L-order constraint</i>	87
<b>Figure 4-8:</b> <i>Generalized L-order constraint</i>	88
<b>Figure 4-9:</b> <i>Resolving channel routing order conflict loop example</i>	89
<b>Figure A-1:</b> <i>Three solutions for Block-packing Example One</i>	101
<b>Figure A-2:</b> <i>Original solution to Block-packing Example One</i>	102
<b>Figure A-3:</b> <i>Two solutions for Block-packing Example Two</i>	103
<b>Figure A-4:</b> <i>Block-packing Example Two: Derivation of Solution #1 (part 1)</i>	104
<b>Figure A-5:</b> <i>Block-packing Example Two: Derivation of Solution #1 (part 2)</i>	105
<b>Figure A-6:</b> <i>Original solution to Block-packing Example Two</i>	106
<b>Figure A-7:</b> <i>Four solutions for Block-packing Example Three</i>	108
<b>Figure A-8:</b> <i>Block-packing Example Three: Solution #7 (channels only)</i>	109
<b>Figure A-9:</b> <i>Block-packing Example Three: Solution #7 (initial position)</i>	110
<b>Figure A-10:</b> <i>Block-packing Example Three: Solution #7 (1 step back)</i>	111
<b>Figure A-11:</b> <i>Block-packing Example Three: Solution #7 (2 steps back)</i>	112
<b>Figure A-12:</b> <i>Block-packing Example Three: Solution #7 (3 steps back)</i>	113
<b>Figure A-13:</b> <i>Block-packing Example Three: Solution #7 (4 steps back)</i>	114
<b>Figure A-14:</b> <i>Block-packing Example Three: Solution #7 (5 steps back)</i>	115

<b>Figure A-15:</b> <i>Block-packing Example Three: Solution #7 (6 steps back)</i>	116
<b>Figure A-16:</b> <i>Block-packing Example Three: Solution #7 (7 steps back)</i>	117
<b>Figure A-17:</b> <i>Block-packing Example Three: Solution #7 (8 steps back)</i>	118
<b>Figure A-18:</b> <i>Block-packing Example Three: Solution #7 (9 steps back)</i>	119
<b>Figure A-19:</b> <i>Block-packing Example Three: Solution #7 (10 steps back)</i>	120
<b>Figure A-20:</b> <i>Block-packing Example Three: Solution #7 (11 steps back)</i>	121
<b>Figure A-21:</b> <i>Block-packing Example Three: Solution #7 (12 steps back)</i>	122
<b>Figure A-22:</b> <i>Original solution to Block-packing Example Three</i>	123
<b>Figure A-23:</b> <i>Floorplanning Example One</i>	126
<b>Figure A-24:</b> <i>Floorplanning Example Two</i>	127
<b>Figure B-1:</b> <i>View of four blocks</i>	131
<b>Figure B-2:</b> <i>View of the four blocks after Topologize</i>	132
<b>Figure B-3:</b> <i>View of the four blocks after Geometrize</i>	133
<b>Figure B-4:</b> <i>View of the four blocks after reshaping block c2</i>	134
<b>Figure B-5:</b> <i>View of channels only</i>	135

**This page is intentionally blank**



# Contents

Preface	p-3
List of Figures	p-5
<b>1. Introduction</b>	<b>1</b>
1.1 The IC Layout Problem	1
1.1.1 VLSI Layout Systems	1
1.1.2 Automatic Layout Systems	2
1.2 Phoenix Overview	2
1.2.1 Topological Model	2
1.2.2 Topological Operators	3
1.2.3 Modes of Operations	3
1.2.4. Hierarchical Decomposition	3
1.2.5 Bottom-up and Top-down Design	4
1.2.6 Problem Size	4
1.3 Complexity of IC Layout	4
1.4 Thesis Outline	5
<b>2. The Layout Model</b>	<b>7</b>
2.1 The Modeling Problem	7
2.2 Convex Quadrics	8
2.3 Layout Model Overview	9
2.3.1 A Formal Description	9
2.3.1.1 Topological Invariants	9
2.3.1.2 Geometrical Invariant	11
2.3.2 The Topological Model	11
2.3.2.1 Blocks	11
2.3.2.2 Channels	12
2.3.2.3 Intersections	13
2.3.2.4 Channel Intersection Graph	15
2.3.2.5 Advantages of the Topological Model	15
2.3.3 Layout Model Summary	16
2.4 Topologize Operation	17

2.4.1 Topologize Overview	17
2.4.2 Topologize Algorithm	19
2.4.2.1 MakeOutlines Routine	19
2.4.2.2 MakeChannels Routine	21
2.4.3 Topologize Summary	23
2.5 Geometrize Operation	23
2.5.1 Geometrize Overview	23
2.5.1.1 Channel Position Graph	24
2.5.2 Geometrize Algorithm	26
2.5.2.1 Channel Positioning	26
2.5.2.2 Block Positioning	29
2.5.3 Geometrize Summary	29
2.6 Topological Operators	30
2.6.1 Topological Operators Overview	30
2.6.1.1 Tracking and Backtracking	30
2.6.1.2 First Class Operations	30
2.6.1.3 Simplifications	31
2.6.2 Simple Operations	32
2.6.3 Complex Operations	35
2.6.4 Primitives	38
2.6.5 Shrink	39
2.6.5.1 Complexity of Shrink	39
2.6.5.2 Overview of Shrink	41
2.6.5.3 Algorithm for Shrink	41
2.6.5.4 Proof of Correctness	41
2.6.5.5 Shrink Summary	42
2.6.5.6 Completeness	42
2.6.6 Grow	43
2.6.6.1 Generalized Slack	43
2.6.6.2 Restricted Grow	44
2.6.7 Miscellaneous Operations	47
2.6.7.1 Stack Operations	47
2.6.7.2 Other Operations	47
2.6.8 Topological Operators Summary	48
2.7 Layout Model Summary	49
2.7.1 Other Layout Model Issues	49

2.7.1.1 Fake Blocks	49
2.7.1.2 External Constraints	50
2.7.1.3 A More Efficient Geometrize	51
2.7.2 Layout Model Conclusion	51
<b>3. Placement Heuristics</b>	<b>53</b>
3.1 Problem Statement for Placement	53
3.2 Placement Overview	53
3.3 Review of Search Techniques	55
3.3.1 Basic Search Concepts	55
3.3.2 Local Search	57
3.3.3 Steepest Descent	58
3.3.4 Look-ahead Search	59
3.3.5 Multi-path Search	61
3.3.6 Summary on Search Heuristics	64
3.4 Placement	64
3.4.1 Shape Determination	64
3.4.2 Floorplanning	68
3.4.3 Initial Placement	71
3.4.4 Placement Improvement	72
3.4.5 Other Placement Issues	72
3.5 Cost Function	73
3.5.1 Net Length Estimation	74
3.5.2 Channel-width Estimation	74
3.6 Placement Summary	77
<b>4. Routing Heuristics</b>	<b>79</b>
4.1 Problem Statement for Routing	79
4.2 Routing Overview	79
4.3 Global Routing	80
4.4 Detailed Routing	82
4.4.1 Channel Order Constraint Graph	83
4.4.1.1 T-order Constraint	84
4.4.1.2 Generalized T-order Constraint	85
4.4.1.3 L-order Constraint	85
4.4.1.4 Generalized L-order Constraints	86

4.4.1.5 Routing Order Constraint Summary	87
4.4.2 Resolving Routing Order Conflicts	88
4.5 Routing Summary	90
<b>5. Conclusions</b>	<b>91</b>
5.1 Conclusions	91
5.2 Contributions	91
5.3 Suggestions for Future Work	92
5.3.1 Comparing Search Heuristics	92
5.3.2 Combining Layout Heuristics	92
5.3.3 Implementing Hierarchical Decomposition	93
5.3.4 Missing Corners	93
5.3.5 More Complex Shapes	93
5.3.6 Topological Routing Operations	94
5.3.7 Solving Channel Routing Order Conflicts	94
<b>A. Placement Results</b>	<b>97</b>
A.1 Results Overview	97
A.2 Overview of Block-packing Examples	97
A.2.1 Block-packing Problem Statement	97
A.2.2 Block-packing Heuristics	97
A.2.3 Discussion of Block-packing Examples	98
A.3 Overview of Floorplanning Examples	99
A.3.1 Floorplanning Problem Statement	99
A.3.2 Floorplanning Heuristics	99
A.3.3 Discussion of Floorplanning Examples	99
A.4 Discussion of Results	100
A.5 Block-packing Results	101
A.5.1 Block-packing Example One	101
A.5.2 Block-packing Example Two	103
A.5.3 Block-packing Example Three	107
A.6 Floorplanning Results	125
<b>B. Design and Implementation of Phoenix</b>	<b>129</b>
B.1 Background and Programming Environment	129
B.2 Interactive Interface Overview	129

B.3 Implementation Discussions	136
B.3.1 Design Principles	136
B.3.2 Data Structure	137
B.3.2.1 Channel	137
B.3.2.2 Intersection	137
B.3.2.3 Block	137
B.3.2.4 Data Structure Summary	138
B.3.3 Implementing Tracking and Backtracking	138
B.3.4 Implementing First Class Operations	138
B.3.5 Debugging Aids	139
B.3.6 Implementation Summary	140
<b>References</b>	<b>141</b>

**This page is intentionally blank**

# Chapter 1

## Introduction

This chapter gives a brief introduction to the Integrated Circuit (IC) layout problem. It is intended to put the work in a proper perspective. See [12] for a general overview of circuit layout.

### 1.1 The IC Layout Problem

Rapid advances in IC fabrication technology, especially in Very Large Scale Integration (VLSI) technology, have made possible the fabrication of single chips with enormous numbers of transistors. As the number of transistors integrated onto a chip increases, so does the difficulty in designing the chip. This makes the task of designing a state-of-the-art chip extremely difficult.

#### 1.1.1 VLSI Layout Systems

Designing a large VLSI circuit is a costly, time consuming, and tedious process. Part of the complexity is in the physical layout of the components (circuit elements) in the circuit. Various computer-aided design tools are needed, and have been developed, to help the circuit designers in this phase of the design. The kinds of VLSI design tools range from general-purpose drawing systems (systems with little or no knowledge of the VLSI layout problem) and special-purpose drawing systems (systems specifically built for VLSI layout) to silicon compilers that compile the high-level functional descriptions of circuits into physical layouts. See Chapter 7 in [13] for an overview of VLSI design systems.

### **1.1.2 Automatic Layout Systems**

Between the two extremes are the automatic layout systems, which take as input a description of the components and their interconnection requirements, and automatically place the components and route their interconnections. Automatic layout systems vary in the amount of interaction needed from the designer. They range from completely automatic to requiring detailed manual interactions from the designer throughout the design process. This thesis is about an automatic layout system that can support both modes of operation.

## **1.2 Phoenix Overview**

Phoenix is an interactive, hierarchical, automatic VLSI layout system. It addresses one of the most difficult and important combinatorial problems in VLSI layout systems (Section 9.5, [13]): given a collection of blocks with pins on the boundaries, and a collection of nets, which are sets of pins that are to be wired together, find a good way to place the blocks (placement) and run the wires (routing) so that the wires are short and the layout area is small.

The layout problem is separated into a placement phase and a routing phase to make the problem more manageable. The focus of this thesis is on the placement phase. The routing phase has not been completed yet. (The design of the routing phase is based on [7].)

### **1.2.1 Topological Model**

Phoenix uses a topological layout model. The model is topological in the following sense: only relative positions of the blocks matter. Moreover, during the routing phase, if more spaces are needed to route the interconnections, the blocks can be moved apart to increase the widths of the routing channels, without changing the placement topology. This property is one of the main advantages of the topological approach: it strongly de-couples routing from placement.



### **1.2.2 Topological Operators**

A set of topological operators (topological operations) is defined for the model; only these operators can operate on the model. These topological operators are complete, and they have the properties of being extensible and supported by the undo and redo capabilities. The undo and redo capabilities are essential for implementing the placement heuristics developed in this thesis.

### **1.2.3 Modes of Operations**

Phoenix allows the designer to interact with the system in real time. It uses a graphical interface to display information to the designer and the designer can execute the topological operations and enter data through the keyboard and the *mouse*. The system can also be used in a batch (stand-alone) mode. This interactive feature was invaluable for debugging the system in the early implementation phase, and for developing and refining placement heuristics in the later phases.

### **1.2.4. Hierarchical Decomposition**

A large layout is usually decomposed into a hierarchy of smaller, more manageable sub-problems. The hierarchy may be part of the input specified by the designer (in which case it often corresponds to the functional decomposition of the circuit), or obtained by using automatic partitioning algorithms. At each level of the hierarchy, the complete layout including the lower-level blocks and their interconnections is collectively called an *assembly*.

The complexities involved in implementing a truly hierarchical layout system is not address in this work. Nevertheless, Phoenix is hierarchical in that it can hierarchically decompose a layout problem. The model and layout heuristics developed can handle the layout problem of an assembly at any level of the hierarchy.

### **1.2.5 Bottom-up and Top-down Design**

Phoenix supports both bottom-up and top-down design methodologies. In the former method, blocks are placed and routed in an assembly to form a super-block at a higher hierarchical level. In the latter, blocks are placed and shaped to fit within an outline derived from higher-level constraints. The shapes of the blocks obtained then become constraints for designing the lower-level blocks. This thesis describes only the bottom-up and top-down layout at one level of the hierarchy. Nevertheless, the results are applicable to any other levels in the hierarchy.

### **1.2.6 Problem Size**

The number of blocks the layout heuristics are intended to handle approximates the number of circuit components a VLSI designer works with in the layout process. This number is typically on the order of ten to twenty. The topological model, however, can handle an order of magnitude or more blocks in an assembly.

## **1.3 Complexity of IC Layout**

The difficulty of the circuit layout problem is well recognized by both the applied and the theoretical computer scientists. Theoretical computer scientists have proved that many routing and placement related problems, under highly simplified formulation, are NP-complete. NP-complete problems are widely believed to have no polynomial-time algorithms; they can only be solved by methods that require exponential amount of time with respect to the size of the input. This is one of the justifications for using heuristics to solve layout problems. (See [15] for a very accessible discussion on NP-completeness and approximation methods for addressing NP-complete problems.)

## 1.4 Thesis Outline

The main body of the thesis describes the ideas and the concepts developed during the design and implementation of Phoenix. Implementation issues are discussed in the Appendix B.

**Chapter 2** describes the topological layout model and the operations supported by the model. The properties of the topological operators are characterized and a proof of their completeness is given. This chapter contains the major results of the thesis.

**Chapter 3** discusses the placement heuristics. Initial placement, placement improvement, floorplanning and shape determination are described

**Chapter 4** discusses the routing heuristics. The main reference for this chapter is [7].

**Chapter 5** summarizes the thesis and suggests future work.

**Appendix A** shows the preliminary results obtained using the placement heuristics on three block-packing problems and two floorplanning examples.

**Appendix B** discusses the design and implementation of Phoenix. Implementation hints are given.

**This page is intentionally blank**

## Chapter 2

### The Layout Model

Circuit layout, as described in this thesis, is inherently a two-dimensional geometrical problem, but the languages used to implement a solution are linear — proper layers of abstraction must be built to represent the problem in a more tractable form. This chapter gives a conceptual view of the layout model developed. The ideas here form the basis for the rest of the thesis.

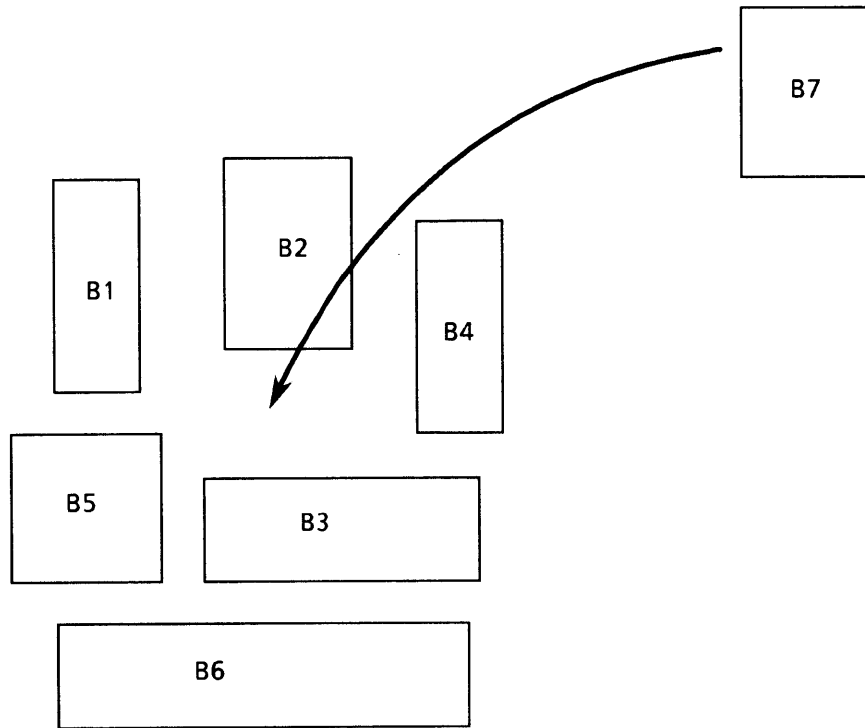
#### 2.1 The Modeling Problem

To address the layout problem effectively, the layout model must provide tools to tackle the complexities in the placement and the routing problems. This chapter is focused on the tools used to tackle the placement problem. (The modeling of the routing problem is discussed in Chapter 4.) First the complexities in modeling the placement problem are described.

The placement problem is to place a collection of blocks on a plane so that they do not overlap and all the interconnections between the blocks can be routed. The objective is to minimize the layout area and the interconnection length.

An example of placing a block is given in Figure 2-1 to illustrate the ideas behind the layout model. The problem is to place the block B7 with the rest of the blocks. The following questions must be answered before B7 can be placed:

- ① How to represent the holes (unoccupied spaces) available to place B7?
- ② Perhaps the hole is not large enough to fit B7; it needs to be enlarged (by moving surrounding blocks apart). How to represent this expansion operation?



**Figure 2-1: Placing a block**

- ③ The hole is too large for B7, wasting the surrounding spaces. Moving the surrounding blocks a little closer is desired. How to represent this compaction operation?

## 2.2 Convex Quadrics

The kinds of shapes the layout model can handle are rectangles with some of their corners removed. In this thesis such shapes are called *convex quadrics* and are specified as follows: the *n*th-order Convex Quadrics, **CQ<sub>n</sub>**, is the class of rectangles with up to n corners removed from each of the four principal corners of the rectangles. The dimensions of the shapes do not matter; only the number and location of the corners removed are relevant. The resultant

figures must remain connected. It is clear that  $CQ_n$  contains  $CQ_m$  for  $n$  larger than  $m$ , and there are  $(n + 1)^4$  different shapes in  $CQ_n$ .

Figure 2-2 shows every representative of  $CQ_1$  partially ordered by the relation, *topologically contains*, a transitive and reflexive relation. An object *topologically contains* another object iff for every corner that is removed from the object, the corresponding corner is also removed from the other object. This is distinct from *geometrical containment* where one object can physically contain another.

## 2.3 Layout Model Overview

The model is essentially an extension of the Channel Intersection Graph Model in [7]. The topological operators on the model are inspired by the *Shrink* and *Grow operations* of the Wall Model in [10]. These three models are topological models; only relative positions matter. The geometry must be computed explicitly and the computation is deferred until needed. Nevertheless, the novel features of the model in this thesis are:

- ① Ability to handle non-rectangular blocks.
- ② A complete and extensible set of topological operators.
- ③ Tracking and backtracking capabilities.

### 2.3.1 A Formal Description

§ An *assembly* is a finite collection of channels and blocks satisfying the topological and the geometrical invariant below.

§ *Channels* are finite line segments.

§ *Blocks* are  $CQ_n$  of arbitrary dimensions. They are either *placed* or *unplaced*, but not both.

#### 2.3.1.1 Topological Invariants:

- ① The assembly is bounded by four channels forming a rectangle.
- ② Only orthogonal channels may intersect.

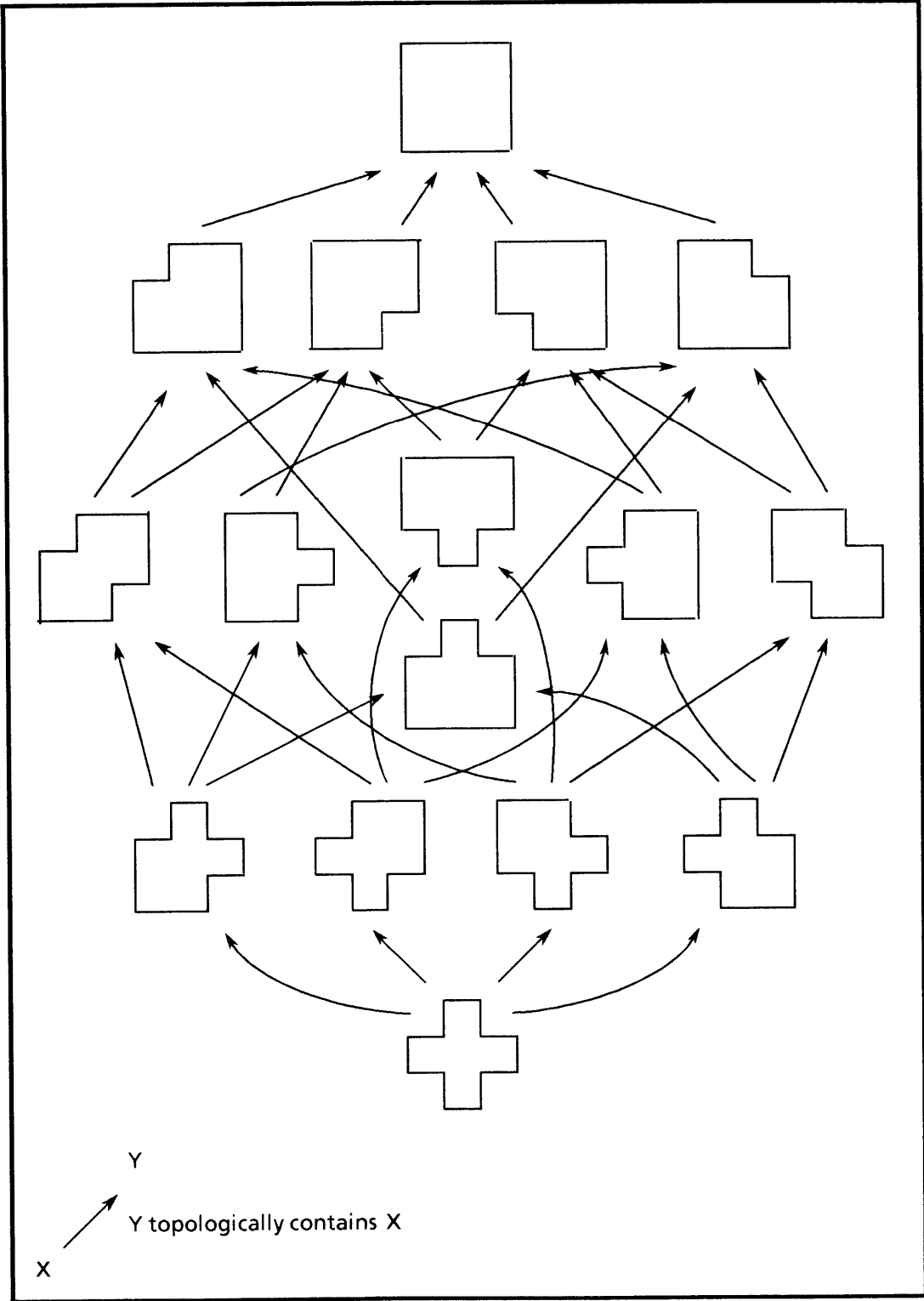


Figure 2-2: Partial ordering of CQ1



- § Two channels intersect to form an *intersection*.
- ③ Every channel is bounded by an intersection at each end.
- § A *topological hole* is a finite region bounded by channels.
- § Two topological holes are *adjacent* iff they lie on opposite sides of a channel.
- § Two holes have the same topology iff they topologically contain each other.
- § Two assemblies have the same topology iff there exists a 1-1 mapping between holes of the same topology in the two assemblies, and the mapping preserves the adjacency relationship between the holes.
- ④ There is a 1-1 correspondence between topological holes and placed blocks. Every topological hole topologically contains its corresponding block.

#### **2.3.1.2 Geometrical Invariant:**

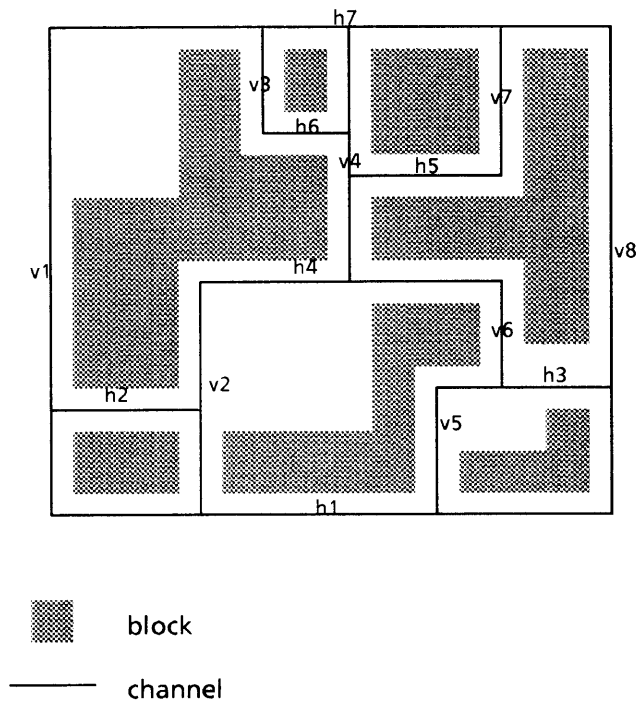
- ① Every topological hole geometrically contains its blocks such that the perpendicular distance between a channel and the blocks on its sides is at least half the channel-width.

### **2.3.2 The Topological Model**

There are two kinds of objects in the topological model, namely, blocks and channels. A circuit layout is represented by an assembly of blocks and channels. (See Figure 2-3)

#### **2.3.2.1 Blocks**

Blocks are abstractions of the layout components. Layout components may have internal structures but only their shapes matter with respect to the model. This information is captured by the dimensions of the blocks and the constraint that blocks do not overlap. The model restricts the shapes of the blocks to be **CQ1**. However, the generalization to **CQn** is possible.



**Figure 2-3:** *Blocks and channels*

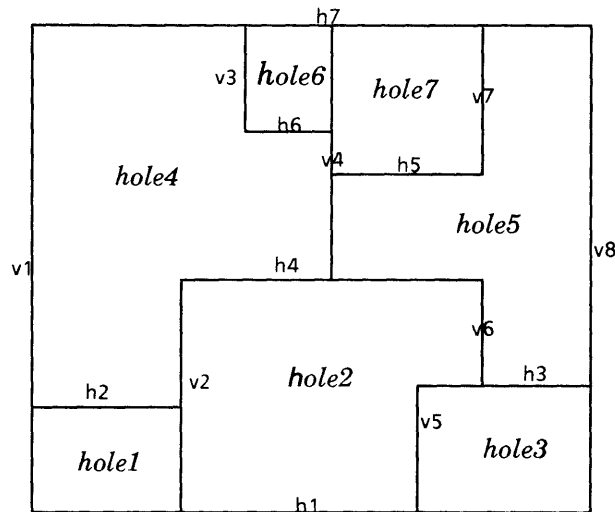
### 2.3.2.2 Channels

Channels are auxiliary line segments surrounding each block. They define the placement topology by partitioning the layout surface into topological holes (see Figure 2-4). Each topological hole contains a block.

Figure 2-5 shows how the same channel topology in Figure 2-3 can accommodate different geometries. Notice how the channels move and stretch. Blocks are placed after positioning the channels.

Channels denote the tracks needed to route the interconnections among the blocks. Each channel has the following attributes:

- ① position



**Figure 2-4: Topological holes**

② channel-width

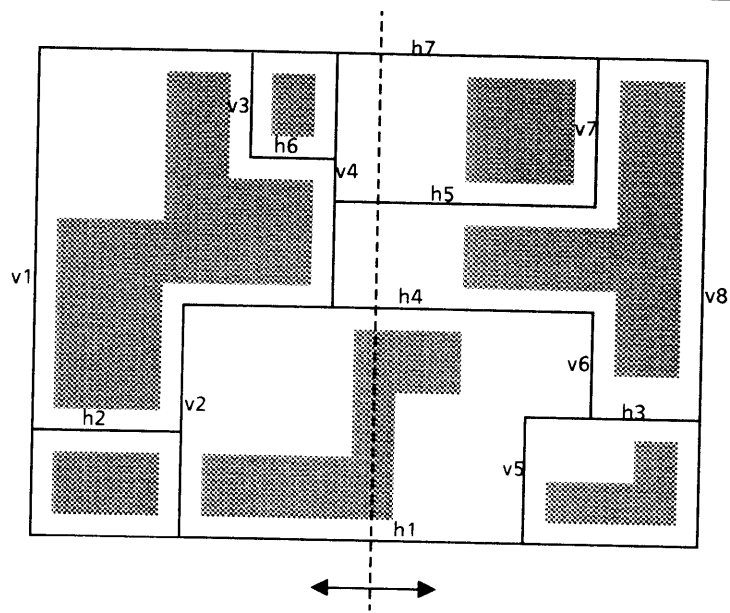
③ slack

The *position* of a channel is the coordinates where the channel lies. The *channel-width* represents the thickness of the channel. Blocks are placed at least half the channel-width from the channel, so that associated with each channel there is a band of empty space at least the channel-width wide. Changing the channel-width does not affect the placement topology.

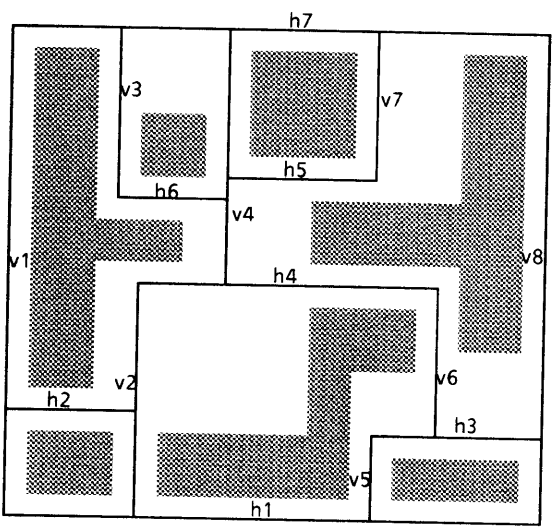
The *slack* is the range of feasible positions of the channel. For example, the slacks of channels h3, h6, v3, v4 and v7 in Figure 2-5 (b) and (c) have a larger range than the rest of the channels. Similarly blocks can also have slacks (in both horizontal and vertical directions).

### 2.3.2.3 Intersections

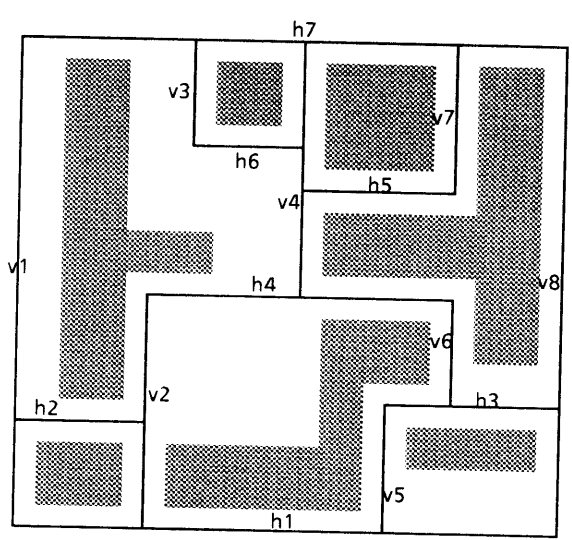
Every channel in the assembly is bounded by two other channels, which determine the length of the bounded channel. The only possible intersections bounding a channel are T-



(a) Channels are stretched horizontally about dotted line



(b) Channels and blocks are positioned from left to right and bottom to top



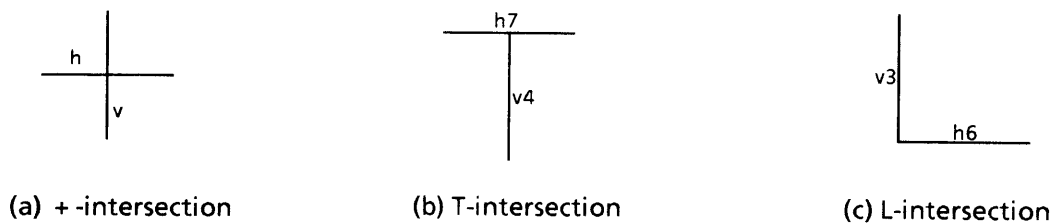
(c) Channels and blocks are positioned from right to left and top to bottom

Figure 2-5: Different geometries with the same topology

intersections and L-intersections. In general, three types of intersections are possible:

- ① +-intersection
- ② T-intersection
- ③ L-intersection

Figure 2-6 shows an example of each type of intersection. The T-intersection and the L-intersection can have four different orientations. There is only one orientation for the +-intersection.



**Figure 2-6:** *Three possible types of channel intersections*

#### 2.3.2.4 Channel Intersection Graph

Every channel keeps a set of pointers to its intersections and surrounding blocks, and every block keeps a set of pointers to its surrounding channels. The pointers between the channels and the intersections implicitly define an underlying graph. This underlying graph is the Channel Intersection Graph (CIG). Any changes made on the assembly updates the underlying CIG.

#### 2.3.2.5 Advantages of the Topological Model

- ① The notion of holes is well-defined.
- ② There is no problem of blocks not fitting a hole. Holes expand and contract to fit blocks of any sizes.

- ③ As a consequence of ②, the same model can be used for top-down placement (floorplanning and shape determination).
- ④ The topological approach ensures the routing completion of all interconnections among the blocks since the channel-widths can be changed to accommodate any amount of interconnections, without affecting the placement topology.
- ⑤ Working in the topological domain is more convenient because well-defined and powerful topological operators are available to modify the placement topology.

### 2.3.3 Layout Model Summary

Instead of working on the layout problem in the geometrical domain, the problem is mapped into an equivalent problem in the topological domain. Working in the topological domain has the advantage of not being overwhelmed by geometrical complexities. Only in the final phase of the layout process, or when the quality of a layout is evaluated, is the geometry computed.

To use the above approach, the following operations are provided by the model:

- ① An operation which maps a placement problem, formulated in the geometrical domain, into the topological domain. This is the **Topologize** operation.
- ② An operation which computes the geometry given the topology. This is the **Geometrize** operation.
- ③ A set of operations which operate on the problem in the topological domain. These are the topological operators.

These operations are discussed in subsequent sections of this chapter.

## 2.4 Topologize Operation

This operation maps the placement problem from the geometrical domain into the topological domain. The inputs are the shapes and the positions of the blocks. **Topologize** assumes that the blocks do not overlap. The objective of **Topologize** is constructing channels for the blocks, and hence defining the placement topology.

**Topologize** is discussed in Section 2.4.1, and an algorithm for **Topologize** is given in Section 2.4.2

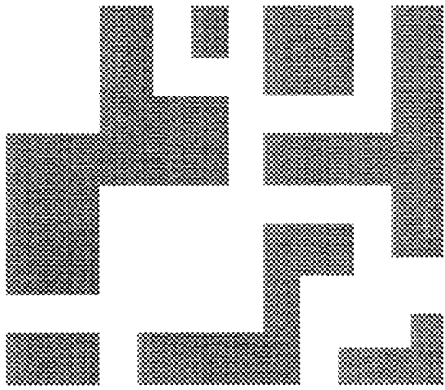
### 2.4.1 Topologize Overview

The algorithm consists of four phases. Figure 2-7 shows the effect of each phase.

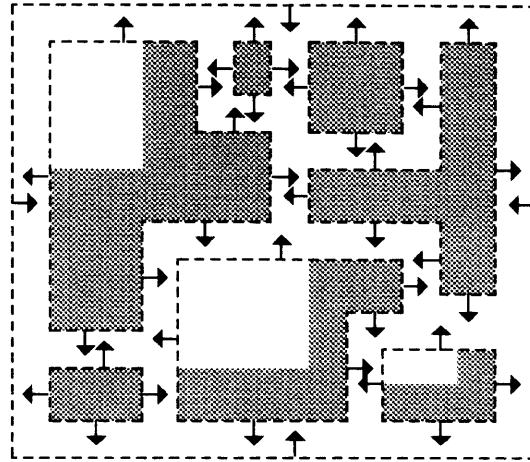
- ① For each block, construct the largest possible outline around it, so that the outlines do not intersect or contain each other. Then, construct a bounding rectangle of the assembly. (An *outline* of a block is a closed figure that topologically and geometrically contains the block and is as short as possible.)
- ② Choose the horizontal or vertical direction as the primary direction. The other direction is considered secondary. Collapse the edges of the outlines in the primary direction so that every channel is perpendicular to the primary direction.
- ③ Collapse the edges of the outlines in the secondary direction. This completes all the channels.
- ④ Use **Geometrize** to reposition the blocks and the channels if necessary.

#### Remarks:

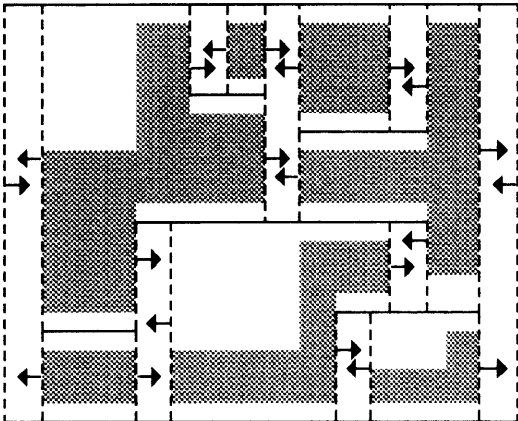
- ① There are different ways of constructing the outlines. (See Figure 2-8.)
- ② After constructing the outlines for the blocks, in phase ①, the number of interior L-intersections is determined. This number is independent of the primary direction chosen.
- ③ It may not be possible to position some channels so that they do not intersect any blocks without moving the blocks. (See Figure 2-9.)



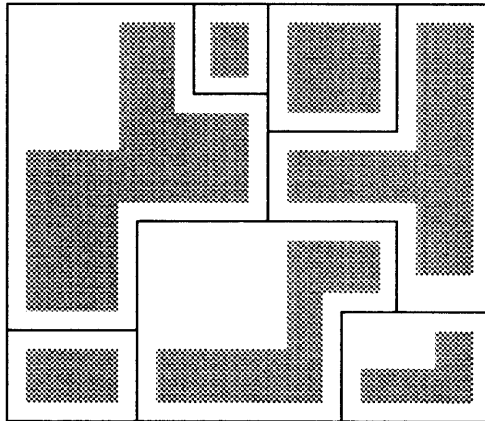
(a) Input to Topologize




(b) After Phase ①



(c) After Phase ②



(d) After Phase ③

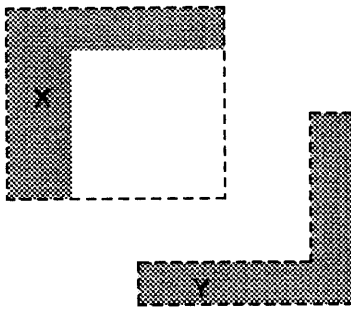
 Outline

 Interval of allowed positions. (Semi-infinite)

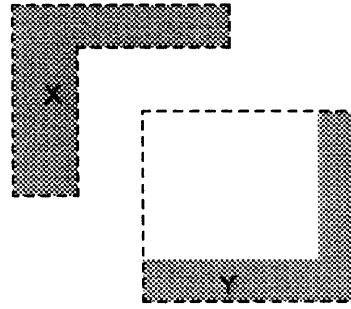
**Note:** Primary direction is vertical

**Figure 2-7: A conceptual view of Topologize**





(a) Block X has higher precedence



(b) Block Y has higher precedence

**Figure 2-8:** *Different ways of forming outlines*

- ④ This algorithm tends to produce longer channels in the primary direction. (See Figure 2-9.)

**Note:** Only L-intersections and T-intersections are formed by the algorithm. +- intersections must be formed explicitly.

### 2.4.2 Topologize Algorithm

This algorithm consists of two routines: **MakeOutlines** and **MakeChannels**. Without loss of generality, let vertical be the primary direction.

#### 2.4.2.1 MakeOutlines Routine

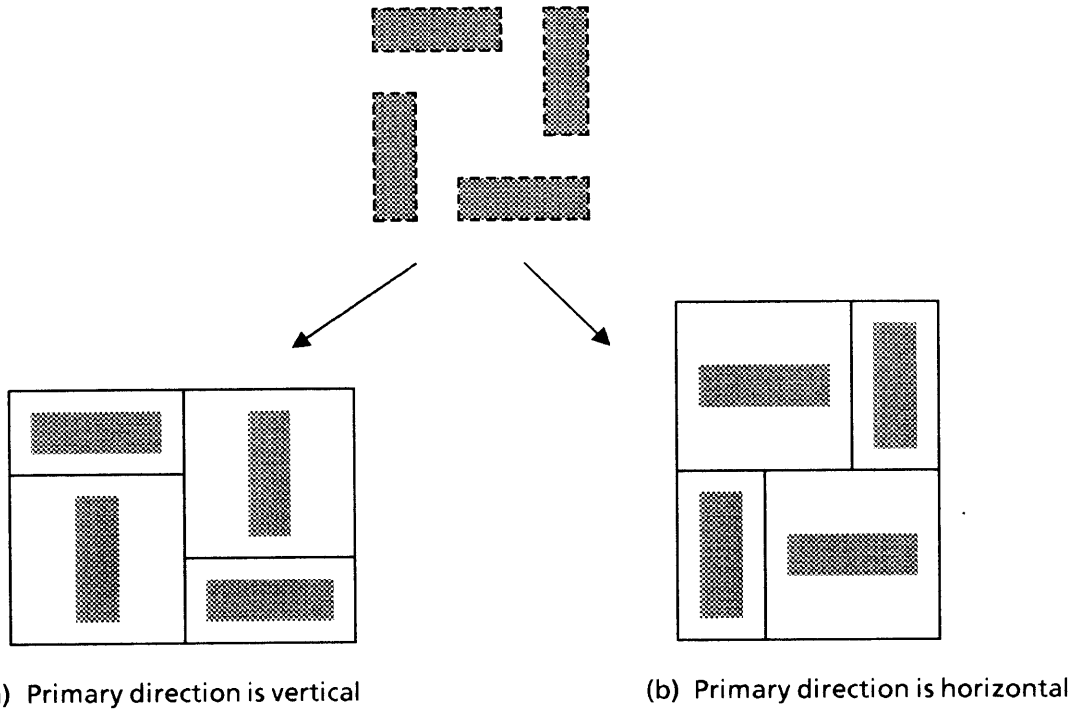
**Input:** The shapes of the blocks and their positions.

**Output:** A rectangle that contains all the blocks and the outlines around them. The blocks do not intersect or contain each other.

**Method:** See Figure 2-10.

**Notations:**

- ① "∩" means intersection.



**Note:** Blocks are repositioned by Geometrize

**Figure 2-9:** *Different ways of forming channels*

② “-” is the comment symbol.

Remark:

The average number of pairwise comparisons between blocks on line 3 can be reduced by sorting the blocks in increasing order as follows:

- ① Sort blocks by the y-coordinates of their bottom edges.
- ② If the y-coordinates are equal then sort by the x-coordinates of the left endpoints of the edges.

This ordering is unique from the assumption that blocks do not overlap. The number of comparisons is reduced by comparing the first block in the list with the rest of the blocks until

```

PROCEDURE MakeOutlines

1. p: list of block pairs ← NIL;    -- conflict pairs

2. Initialize outline of every block to its bounding rectangle;

3. FOR every unordered pair of blocks (X, Y) DO
   IF X ∩ Y THEN ERROR;    -- Should not overlap
   IF (outline of X) ∩ Y THEN retract the corner of X's outline that intersects Y;
   IF X ∩ (outline of Y) THEN retract the corner of Y's outline that intersects X;
   IF (outline of X) ∩ (outline of Y) THEN p ← p + (X, Y); -- found a conflict pair
   ENDLLOOP;

4. FOR every (X, Y) in p DO
   IF (outline of X) ∩ (outline of Y) THEN -- check if X and Y still conflict
     retract either the corner of X or
     the corner of Y so that the outlines do not intersect;
   ENDLLOOP;

5. Construct a bounding rectangle of all blocks

END MakeOutlines

```

Figure 2-10: *MakeOutlines* routine

the y-coordinate of the top edge of the first block is less than the y-coordinate of the bottom edge of the second block. In the worst case, the running time is still *n choose two*.

#### 2.4.2.2 MakeChannels Routine

Input: Output from the **MakeOutlines** routine.

Output: Channels around the blocks.

Method: See Figure 2-11.

Explanations:

- ① *Edge* refers to an edge of the outline.
- ② The *span* of an edge is the interval of the x-coordinates of its two endpoints.
- ③ The *range* of an edge is a semi-infinite interval pointing outward from the coordinate of the edge to infinity in the direction away from the interior. The edge of the bounding rectangle has a range pointing inward.

```

PROCEDURE MakeChannels

1. PosEdges ← ordered list of all horizontal edges with positive range;

2. NegEdges ← ordered list of all horizontal edges with negative range;

3. UNTIL PosEdges = NIL DO
    Curtain: set of intervals ← ∅;
    s ← first edge in PosEdges;
    PosEdges ← PosEdges - s;
    r ← range of s;
    remove all edges from NegEdges with y-coord less than y-coord of s;
    FOR each t in NegEdges in ascending order DO
        IF (span of t) ∩ (span of s) AND
           (span of t does not intersect intervals in Curtain OR coord of t is in r) THEN
            BEGIN
                mark s and t to be in the same cluster;
                r ← r ∩ (range of t);
            END;
        Add (span of t) to Curtain;
        IF some union of intervals in Curtain covers (span of s) THEN EXITLOOP;
    ENDFOR;
ENDLOOP;

4. Collapse all horizontal edges of the same cluster into a horizontal channel

5. FOR each horizontal channel DO
    cluster the pair of adjacent vertical edges at each end of the channel;
    cluster up the pairs of adjacent vertical channels on both sides of the channel;
ENDFOR;

6. Collapse each vertical cluster into a vertical channel;

END MakeChannels

```

Figure 2-11: MakeChannels routine

Remarks:

- ① This algorithm systematically clusters the edges of the outlines.
- ② Operations on a set of intervals can be speeded up by ordering the intervals and incrementally forming unions of intervals to reduce the number of intervals in the set.

### 2.4.3 Topologize Summary

- ① The efficiency of **Topologize** is not critical since it is used only once.
- ② Every channel-width can either be set to some default value or to the resultant range of the corresponding cluster. The latter setting reduces the relative movements between blocks when **Geometrize** is run.

## 2.5 Geometrize

This operation computes the positions of the channels and the blocks. It is called to update the geometry of the assembly after any changes. Given the topology of the assembly and the dimensions of the blocks, it is possible to position the blocks and the channels in such a way that the geometrical invariant (in Section 2.3.1) is preserved. Since this operation is called frequently, its running time is critical. A linear-time algorithm for **Geometrize** using a graph-theoretic approach is described.

### 2.5.1 Geometrize Overview

At the top level, the algorithm consists of two phases:

- ① Channel positioning.

In this phase, the algorithm computes the positions of the channels and their slacks. There is usually no unique way of positioning the channels. The channels are positioned in such a way that the geometrical invariant can be preserved. After this phase the topological holes are completely defined.

- ② Block positioning.

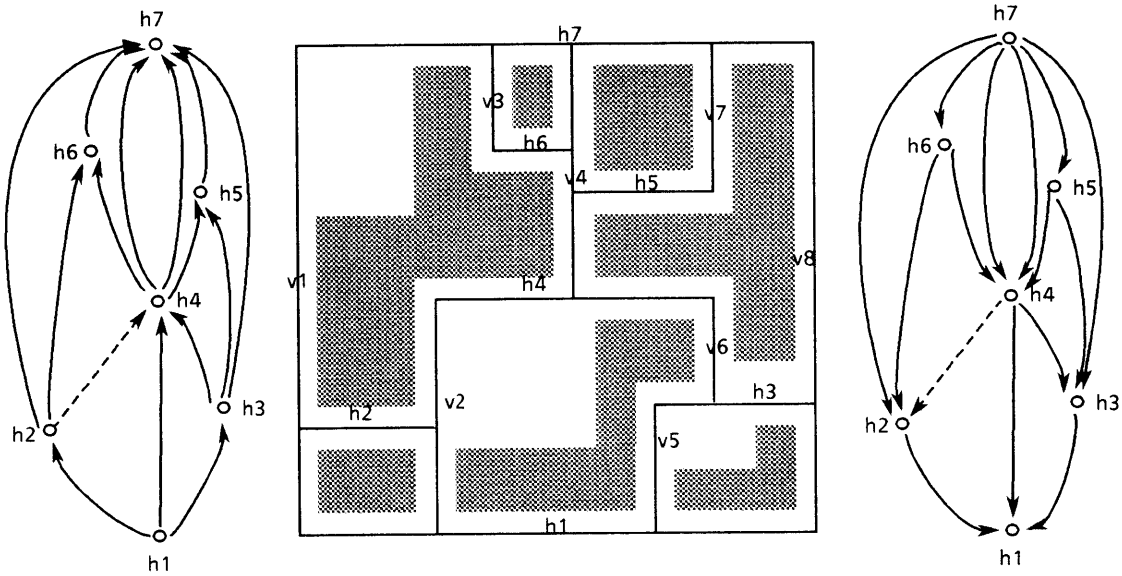
In this phase, the algorithm positions the blocks inside their topological holes so that the perpendicular distances between the edges of the blocks and their corresponding channels are at least half the channel-width, preserving the geometrical invariant. Again there may be no unique way of doing this.

**2.5.1.1 Channel Position Graph**

Channel positioning is further divided into two independent sub-problems: the positioning of horizontal channels and the positioning of vertical channels. The former will be described; the latter is analogous. Positioning the horizontal channels has two phases:

① Construct Channel Position Graph

*Channel Position Graph (CPG)* is a weighted directed acyclic graph. Each node corresponds to a channel and each edge corresponds to a constraint between two channels. The weight on the edge represents the minimum separation between the two channels. (See Figure 2-12.) Two CPGs (bottom-to-top and top-to-bottom) are needed to position the



Bottom-to-top CPG

Top-to-bottom CPG

- Topological constraint, from L-intersection
- Block constraint

**Note:** Weights on the edges are not shown

**Figure 2-12:** Horizontal channel position graphs

horizontal channels and compute their slacks. Four CPGs are needed altogether.

② Compute channel positions and slacks.

Using a CPG, the positions of the nodes can be computed, in time linear with the total number of nodes and edges. This is done by assigning node positions such that the position of a node is assigned only after the positions of all its predecessors have been assigned (i.e., in topological order.) Since the graph is acyclic, this is always possible.

**Remarks:**

① The bottom-to-top CPG is the same as the top-to-bottom CPG except that the directions of the edges are reversed.

② The mapping from topology to geometry is not unique. The positions of a channel computed using the bottom-to-top CPG and using the top-to-bottom CPG may not be the same. The position of a channel computed using a top-to-bottom CPG is always greater than (above) or equal to the position computed using a bottom-to-top CPG.

§ The *slack* of a channel is the interval of the positions computed from the two CPGs.

§ *Critical channels* are channels with zero slacks.

§ A *critical path* is a path in the CPG where every node corresponds to a critical channel.

③ There is at least one critical path between the nodes corresponding to the opposite sides of the channels bounding the assembly. Increasing the weight of an edge on the critical path always increases the dimension of the assembly. (For example, increasing the channel-width of a critical channel or increasing the dimension of a block along its critical dimension will increase the dimension of the assembly by the same amount.) Decreasing the weight does not necessarily reduce the dimension of the assembly since there may be more than one critical path.

- ④ In principle, a channel may be positioned anywhere within its slack. However, to ensure that the topological holes are properly formed, the channels are either positioned using the bottom-to-top CPG or the top-to-bottom CPG.
- ⑤ Channel-width and slack are related to each other.

## 2.5.2 Geometrize Algorithm

### 2.5.2.1 Channel Positioning

Only the algorithm to compute the positions of horizontal channels from bottom to top will be described, the algorithm to compute channel positions from top to bottom is analogous. This algorithm consists of two routines:

#### ① **MakeBot2TopCPG** Routine.

This routine constructs a bottom-to-top CPG based on the topology and the dimensions of the blocks. The algorithm is given in Figure 2-13.

```

PROCEDURE MakeBot2TopCPG
1. Create a node for each horizontal channel;
2. FOR each block DO
   Construct edges of appropriate weights from nodes of lower
   channels to nodes of upper channels;
   ENDLOOP;
3. FOR each L-intersection DO
   Add an appropriate topological constraint to prevent
   horizontal channels from sliding beyond the intersection ;
   ENDLOOP;
END MakeBot2TopCPG

```

**Figure 2-13: MakeBot2TopCPG routine**

Explanations:

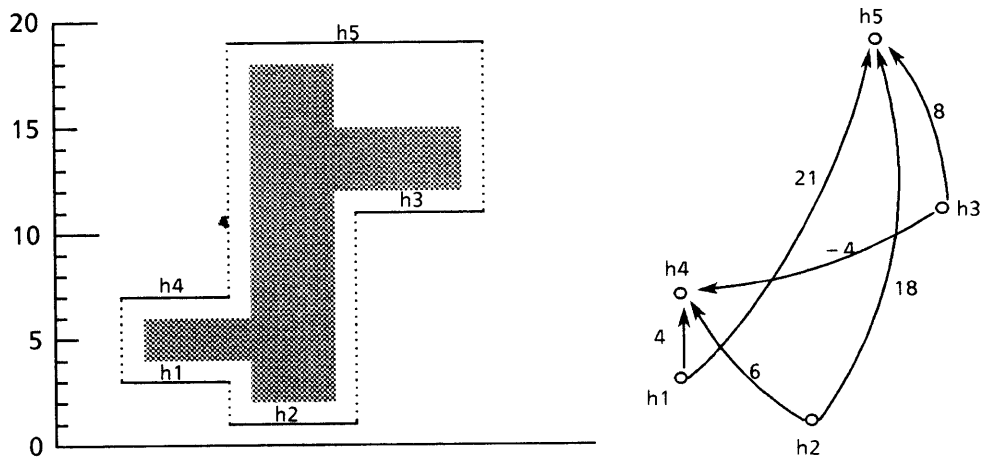
- Line 2: An edge is added to the CPG for each pair of nodes (from lower to upper) corresponding to channels on opposite sides of the topological hole. The weight on the



edge,  $W$ , is given by:

$W = \frac{1}{2} * (\text{sum of the two channel-widths}) + (\text{appropriate constraint from the block})$

(See Figure 2-14.)



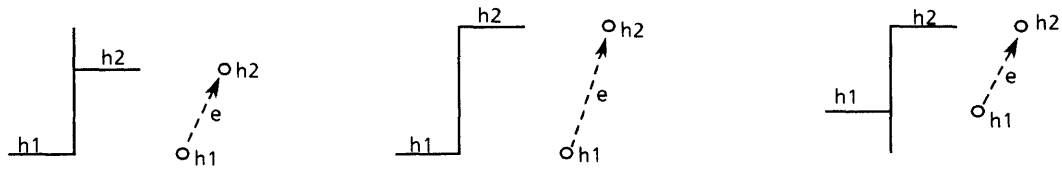
**Note:** All channel-widths are 2 units.  
Upper channels: h4 and h5.  
Lower channels: h1, h2, and h3.

**Figure 2-14:** *Block constraints*

- Line 3: For each interior L-intersection, it may be necessary to add a constraint between the horizontal channel in the L-intersection to the horizontal channel opposite the L-intersection to ensure that the latter channel does not move past the L-intersection. See Figure 2-15.

② **PositionChannelsBot2Top** Routine

This routine takes a CPG and computes the positions of the channels based on the CPG. The routine is shown in Figure 2-16. The running time of this algorithm is  $O(\text{number of nodes} + \text{number of edges})$ .



**Note:**  $e \geq 0$

**Figure 2-15:** *Topological constraints*

```

PROCEDURE PositionChannelsBot2Top
1. Assign positions of all nodes to  $-\infty$ ;
2. Find root and assign position of root to a default value;
3. I: list of nodes  $\leftarrow$  root;
4. UNTIL I = NIL DO
    n  $\leftarrow$  node in I;
    I  $\leftarrow$  I - n;
    FOR each edge (n, m) from n DO
        position of m  $\leftarrow$  MAX[position of m, weight of edge + position of n];
        remove edge;
        decrement incidence count of m by 1;
        IF incidence count of m = 0 THEN I  $\leftarrow$  I + m;
    ENDFOR
ENDLOOP;
END PositionChannelsBot2Top

```

**Figure 2-16:** *PositionChannelsBot2Top routine*

Explanations:

- Each node has a position and an incidence count. *Position* refers to the position of the channel. *Incidence count* is the number of edges pointing at the node.
- The *root* of the CPG is the node with zero incidence count, and there should be exactly one root.

**Remarks:**

- ① Cycles in the CPG can be detected by counting the number of times the loop in line 4 is executed. It should be equal to the number of nodes.
- ② To save space, the bottom-to-top CPG and the top-to-bottom CPG can be merged into one graph.

**2.5.2.2 Block Positioning**

Each block is positioned within its topological hole so that it is at least half the channel-width away from all its surrounding channels. Only when a block is critical in both directions is its position unique.

**2.5.3 Geometrize Summary**

**Geometrize** is the key connection from the topological domain back to the geometrical domain. This operation is needed to compute the geometrical information for the layout. The geometrical information is used to evaluate the quality (or cost) of a layout and is part of the final output of the system. The **Geometrize** algorithm allows the dimensions of the assembly and the critical channels to be determined in linear time. This linear time algorithm is especially effective for handling an assembly with a small number of blocks.

## 2.6 Topological Operators

This section describes the operations on the model. A number of important notions are introduced to show the power of this topological model. The words *operation* and *operator* will be used interchangeably to refer to the topological operators.

### 2.6.1 Topological Operators Overview

The interfaces (arguments and return values) of these operators are inherently complicated because of the amount of information needed to specify the operations. These operators can be called by a client program or by a user through an interactive graphic interface. The latter requires a lot of the arguments to be defaulted for the convenience of the user. Unless stated otherwise, properties mentioned in this section apply to all the operators.

#### 2.6.1.1 Tracking and Backtracking

The system supports these capabilities by keeping the changes made by each operation on two stacks: an undo stack and a redo stack. The redo stack records the sequence of forward changes, while the undo stack the sequence of inverse changes.

When an operator is called it is pushed onto the redo stack and its inverse onto the undo stack. The effect of the operation is undone by popping both stacks and executing the inverse operation that was on the undo stack. Executing the operation that was stored on the redo stack return the system to the state before undoing. This method is both space and time efficient and renders the keeping of multiple solutions feasible.

#### 2.6.1.2 First Class Operations

All first class operations have the following properties:

- ① If the system is in a valid state, calling the operations will either raise an error (because the arguments are not valid) without executing the operations or execute the operations and result in a valid state. This implies built-in checking of arguments in the operations.

This property is essential for debugging and building an interactive user interface. Associated with each first class operation is a predicate that returns true iff executing the operation will not raise an error.

- ② They can be used to build other (composite) first class operations. The argument checking can be turned off temporarily if a composite operation involves some illegal move.
- ③ They are supported by the undo and redo capabilities. When this feature is activated, executing a first class operation will push a frame onto the undo and the redo stacks. This simplifies the bookkeeping of the stacks. For example, when the user calls any operation and then desires to return to the previous state, undoing once will suffice. This property and property ② capture the extensibility of first class operations.

### **2.6.1.3 Simplifications**

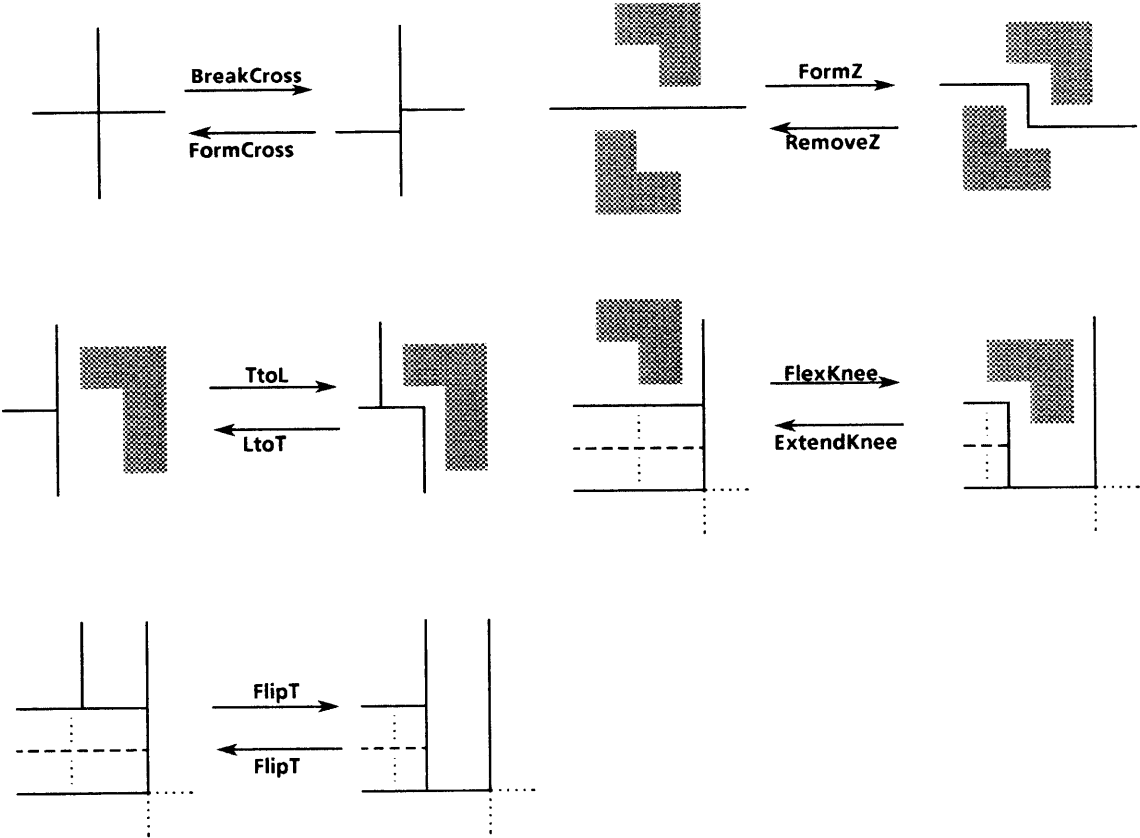
Because the system has a lot of internal states, the functional inverse of an operation may not be its actual inverse. For example, executing the functional inverse of an operation after executing the operation does not necessarily return the system to the original state — the stacks have grown by two frames. It is the actual inverse and not the functional inverse that is pushed onto the undo stack.

For simplicity, however, this distinction will be ignored by treating the system as though it has no internal states. Another simplification is that the operation for only a particular orientation is presented. It should be understood that the same operation can be applied to any of the eight possible orientations. The model has all the symmetries of a rectilinear model.

**2.6.2 Simple Operations**

This section describes the simple topological operators while the more complex ones are described in the next section. The classification of operations into simple and complex is arbitrary and does not reflect any inherent differences between them.

For clarity, the operation and its (functional) inverse, if one exists, are grouped together. More complex operations, however, may not have an easily describable inverse. Refer to Figure 2-17 for the operations in this section.



**Figure 2-17: Simple operations**

**BreakCross** converts a +-intersection into two T-intersections by breaking one of the intersecting channels into two channels at the intersection. The number of channels is increased by one.

**FormCross** takes the ends of two channels each forming a T-intersection on opposite sides of another channel and merges them into a +-intersection. The number of channels is decreased by one.

**FormZ** takes two blocks, each with a corner missing at the corners diagonally across, on opposite sides of a channel and create two L-intersections by breaking the channel and adding an appropriate orthogonal channel so that each of the L- intersections fits into one missing corner. The number of channels is increased by two.

**RemoveZ** takes a channel which has no interior intersections except an L-intersection at each of its ends (the L-intersections are in opposite directions), removes the channel, and merges its intersecting channels. The number of channels is decreased by two.

**TtoL** converts a T-intersection on the side of a channel into an L-intersection by breaking the channel to form an L-intersection and a new T-intersection. The L-intersection fits into a *free* (unused) missing corner of a block on the opposite side of the channel. The number of channels is increased by one.

**LtoT** merges an L-intersection with a T-intersection next to it to form a new T-intersection. The number of channels is decreased by one.

**FlexKnee** takes a channel with an end bounded by a T-intersection and converts it into an L-intersection by splitting and sliding out part of the intersecting channel. The L-intersection fits into a missing corner of a block. The number of channels is increased by one

**ExtendKnee** converts an L-intersection into a T-intersection by merging one of the channels in the L-intersection with another nearby channel, to form the T-intersection on the side of the nearby channel. There must be no obstruction between them. The number of channels is decreased by one.

**FlipT** takes a channel with a T-intersection on its side and bend part of the channel at the intersection, away from the intersection channel so that the intersecting channel and the part bent become one channel. As a result, a new T-intersection bounding the remaining part of the channel is formed. It requires that the part of the channel bent to be bounded by a T-intersection. This operation is its own inverse. The number of channels and the number of T-intersections are unchanged.

**Remarks:**

- ① Topological operations may have certain geometrical characteristics:
  - **BreakCross**, **FormZ**, **TtoL**, and **FlexKnee** tend to result in a smaller assembly, whereas their inverses have the opposite effect.
  - In **FormCross**, when the two channels are merged into one, the slack of the resultant channel is the intersection of the slacks of the two channels. If their slacks do not overlap, then the resultant channel will be a critical channel and the dimension of the assembly along the direction orthogonal to the channel will increase.
- ② When implementing these operations, the arguments of the operations must at least contain enough information to specify the operations and for all possible orientations. There may be drastic difference in complexities between the interface of an operation and the interface of its inverse.



### 2.6.3 Complex Operations

These are the more complicated operators. In some operators, the inverse operators are not implemented because of the complexities in specifying their operations. But in principle they can always be implemented.

**Grow0** places a block by growing a rectangular topological hole and inserting the block into the hole. This topological hole is created by splitting a channel bounded by two of its intersections. The number of channels is increased by one. (See Figure 2-18 for **Grow** and **Shrink** operations.)

**Shrink0** removes a block from a rectangular topological hole and collapses the hole by merging two opposite sides of the hole. There must be either two T-intersections at the ends of one of the sides or two T-intersections diagonally across so that the collapse is not obstructed. The number of channels is decreased by one.

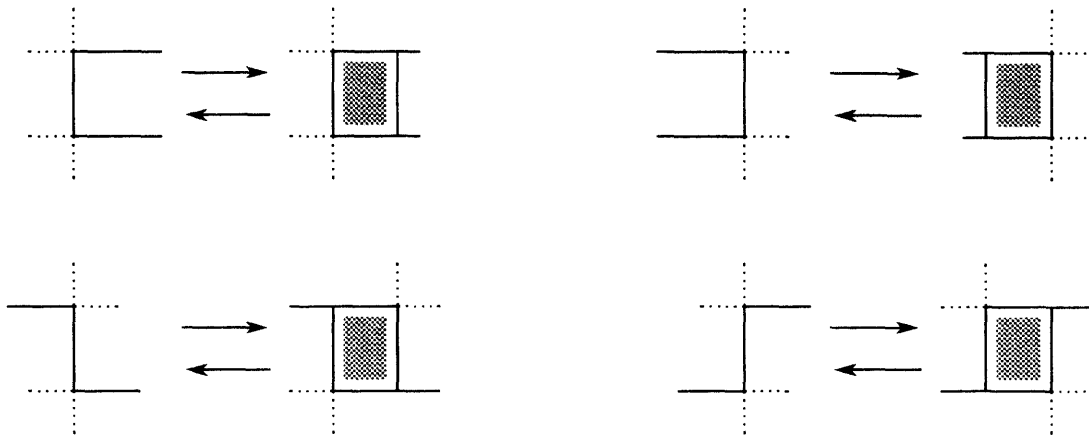
**Grow1** adds a block to the topological hole created at a free corner of a placed block, by adding two channels that form an L-intersection. The number of channels is increased by two.

**Shrink1** removes a block from a rectangular hole bounded by two channels forming an L-intersection. These two channels must have no interior intersections. The number of channels is decreased by two.

**Grow2** is a highly restricted operation. It adds a block with the four surrounding channels, forming a rectangle, to an empty assembly. The number of channels is increased by four.

**Shrink2** removes the last block and its four surrounding channels from the assembly; only an empty assembly remains. The number of channels is decreased by four.

**BreakCrosses** is a composite operation which applies **BreakCross** to all  $+$ -intersections between two specified intersections of a channel. It does not do anything if there are no  $+$ -



(a) Grow0 and Shrink0



(b) Grow1 and Shrink1

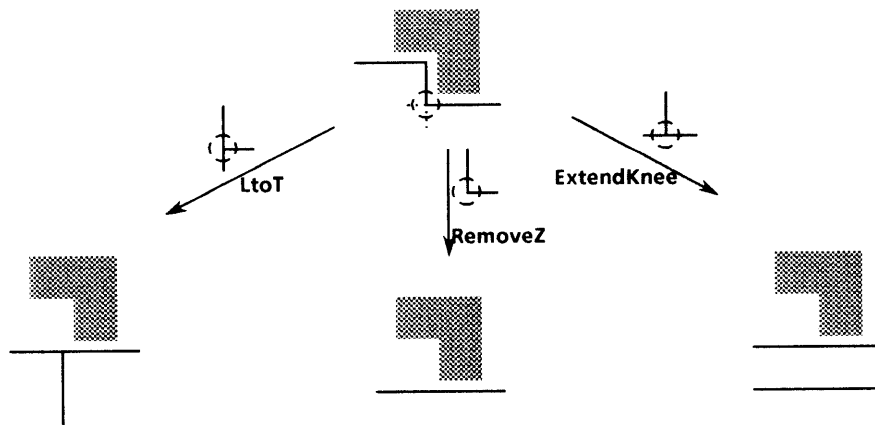
(c) Grow2 and Shrink2

**Figure 2-18:** *Grow and shrink operations*

intersections. The increase in number of channels is equal to the number of + -intersections broken. (No figure.)

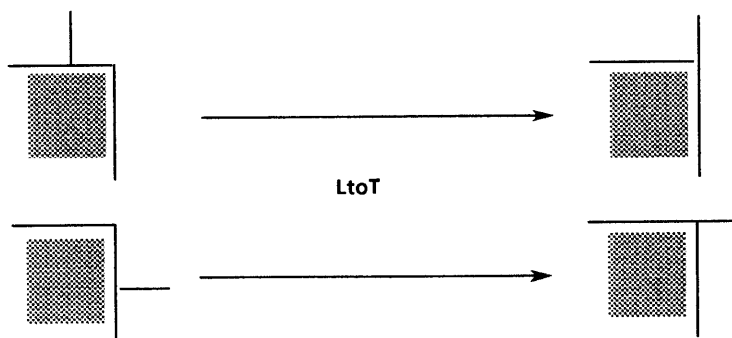
**ClearInteriorKnee** is a composite operation which removes an *interior knee* (an interior L-intersection) from the corresponding corner of a block, freeing the corner of the block. It uses one of **LtoT**, **ExtendKnee**, or **RemoveZ** whichever is appropriate. It may be necessary to apply **BreakCross** to break any + -intersection that is in the way. It is clear that this can

always be done, and perhaps in many ways. (Figure 2-19 shows action taken only on the vertical part of the L-intersection.)



**Figure 2-19: ClearInteriorKnee operation**

**ClearExteriorKnee** is another composite operation similar to **ClearInteriorKnee** except that it is executed from a different perspective. It changes an *exterior knee* (a corner L-intersection) of a topological hole into a corner T-intersection by applying **LtoT** (and **BreakCross** if necessary). This operation may not always achieve its objective (e.g., an L-intersection of two bounding channels). (See Figure 2-20.)



**Figure 2-20: ClearExteriorKnee operation**

**Remarks:**

- ① The last three operations are more heuristic in nature. Their effect on the number of channels and on the topology depends on the situation.
- ② Some geometrical properties of these operations are listed below:
  - **Grow1** always results in an assembly no smaller than the original
  - **Grow0** usually results in an assembly no smaller than the original.

**2.6.4 Primitives**

Sections 2.6.2 and 2.6.3 describe a large number of topological operations. This section unifies them. The issues concerning the primitive operators of the model are addressed.

§ A subset of the operations that can express all the operations of the original set are the *primitives* of the original set.

In the two preceding sections, it is clear that **BreakCrosses**, **ClearInteriorKnee**, and **ClearExteriorKnee** can be expressed by other operators. Hence, a set of primitives for the operations in Sections 2.6.2 and 2.6.3 are as follows:

- ① **FormCross** and **BreakCross**
- ② **LtoT** and **TtoL**
- ③ **FlexKnee** and **ExtendKnee**
- ④ **Grow0** and **Shrink0**
- ⑤ **Grow2** and **Shrink2**

**FormZ** can be expressed by two appropriately chosen **FlexKnee**'s or **TtoL**'s, using **BreakCross** and **FormCross** when some + -intersection is in the way. Similarly, **RemoveZ** can be expressed by **ExtendKnee**'s and **LtoT**'s.

**FlipT** can be expressed by a **FlexKnee** (an invalid intermediate topology may be formed but this is all right) and an **LtoT**. Or equivalently, it can be composed of a **TtoL** and an **ExtendKnee**.

**Grow1** can be expressed as a **Grow0** followed by a **FlexKnee**, and **Shrink1** can be expressed as a **Shrink0** following an **ExtendKnee**.

**FlexKnee** can be expressed as a **FlipT** followed by a **TtoL** (and **ExtendKnee** can be expressed by an **LtoT** followed by a **FlipT**). Hence, another possible set of primitives for the operations is the same list as above but replacing **FlexKnee** and **ExtendKnee** by **FlipT**.

#### Remarks:

- ① For implementation purposes, it may be easier and more efficient to implement a composite operation (non-primitive) as though it is a primitive operation. Examples are **Shrink1** and **Grow1**.
- ② At the level beneath these operations, it is possible to express them by a yet smaller set of implementation primitives. However, these primitives by themselves may not be valid topological operators.

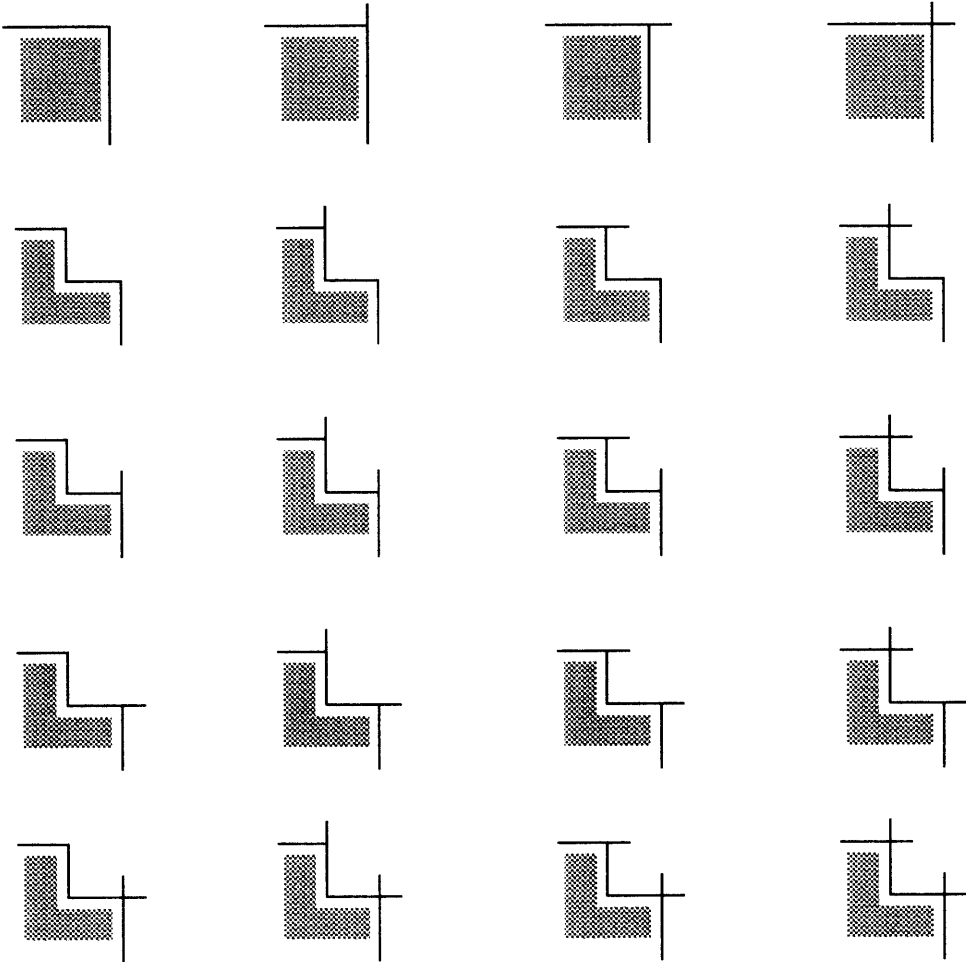
### 2.6.5 Shrink

This composite operation is the key to proving that the topological operators are complete. This operation removes any block in an assembly and shrinks its *topological hole*. It includes **Shrink0**, **Shrink1**, and **Shrink2** which are restricted to removing blocks from rectangular *topological holes* only. It is assumed in the rest of the discussion that there is more than one block in the assembly. Removing the last block can always be handled by **Shrink2**.

#### 2.6.5.1 Complexity of Shrink

The complexity of **Shrink** is largely in the closing up of the topological hole containing the block. In principle, it is possible to collapse the hole by studying the intersections of

channels surrounding the hole. In particular, it is possible to collapse most (but not all, see Figure 2-24) topological holes simply by performing actions based on only the corner intersections.



**Figure 2-21:** *Twenty possible corner intersections for CQ1*

However, for CQ1, there are  $20^4$  different combinations of corners. (See Figure 2-21.) It is possible to pre-compute a table of actions based only on this corner information. The drawbacks with this approach are:

- ① Generating this table of actions may be non-trivial, in light of the fact that there are many ways of collapsing some topological holes.

- ② The table may take up a lot of memory.
- ③ This approach does not readily generalize to **CQ2**, **CQ3**, and so on.

### 2.6.5.2 Overview of Shrink

A heuristic approach is used instead. The idea is to apply a series of reductions to simplify the problem to a case that can be handled by **Shrink0** and **Shrink1**. This approach can always remove a block but does not consider all the possible ways of doing it. This is justifiable because different ways of removing a block do not necessarily result in drastically different topologies. In case it does make a difference, one can always use the basic operations and apply **Shrink0**.

### 2.6.5.3 Algorithm for Shrink

- ① Apply **ClearInteriorKnee** to every corner that contains an interior knee.
- ② Apply **ClearExteriorKnee** to every corner that contains an exterior knee.
- ③ Apply either **Shrink0** or **Shrink1** to remove the block.

### 2.6.5.4 Proof of Correctness

After step ① the resultant topological hole is rectangular, since **ClearInteriorKnee** can always remove the interior knee. Step ② may not be able to remove all the exterior knees but a little case analysis shows that there cannot be more than two exterior knees in the resultant topological hole. (If there are two they must be diagonally across the topological hole.) At step ③, case analysis shows that the resultant hole must be collapsible by either **Shrink0** or **Shrink1**.

### 2.6.5.5 Shrink Summary

- ① **Shrink** is a composite operation.
- ② At each step there may be many ways of achieving the objective. It may be desirable to give *hints* as arguments to **Shrink** to allow finer control.

### 2.6.5.6 Completeness

**Duality:** The functional inverse of any operation in the set can be composed by a sequence of operations in the set. This is slightly weaker than requiring that the functional inverse of any operation be in the set, but is enough to ensure that if a sequence of operations changes a topology to another topology then there exists another sequence that changes it back.

**Completeness:** There exists a sequence of operations in the set that transforms any valid topology to any other valid topology. It is clear that completeness implies duality.

**Claim.** The operations in Sections 2.6.2 and 2.6.3 are complete.

**Proof.** These operations satisfy the duality properties since the primitives of the operations occur in dual pairs. To prove completeness, use **Shrink** to remove all blocks in the first assembly and all the blocks in the second assembly. The two empty assemblies are topologically equivalent. By duality, for each sequence of **Shrink**'s there exists a sequence of inverses that returns the assembly to its original state. Now, to go from one to the other, apply the sequence of **Shrink**'s of the former and then apply the sequence of inverses of the latter (in reverse order).



### 2.6.6 Grow

This is the functional inverse of **Shrink**, and in principle can be implemented by reversing the steps in **Shrink** and applying the inverse to each step as follows:

- ① Use **Grow0** or **Grow1** to add a block to the assembly.
- ② Try all possible ways of forming an exterior knee at each corner (using **TtoL**).
- ③ For each missing corner in the block, try all possible ways of forming an interior knee (using **FlexKnee**, **TtoL**, or **FormZ**).

#### Remarks:

- It is easy to see that the number of ways of growing a new topological hole onto any assembly is finite. So the above algorithm can only give a finite number of possibilities.
- Step ② of the above algorithm depends on the existence of free missing corners of surrounding blocks, while step ③ depends on the missing corners of the block.

Because of the large number of ways of growing a new topological hole, specifying the new hole precisely requires a complicated interface. A restricted **Grow** will be described. It has a relatively simple interface like **Grow0** but includes both **Grow0** and **Grow1**. But first the notion of generalized slack is discussed.

#### 2.6.6.1 Generalized Slack

Recall that the slack of a channel is the range of feasible positions of the channel. Intuitively it is clear that a good place to add a block using **Grow0** is to a channel that has slack matching one dimension of the block and the distance between the two bounding channels matching the other dimension. So slacks can be useful estimators for growing new topological holes.

This notion can be further generalized. Suppose all blocks on each side of a channel are moved as far away from the channel as possible (with the other channels placed at the further

extremes of their slacks) the spaces between the channel and the blocks on its sides determine the generalized slack of the channel. (See Figure 2-22)

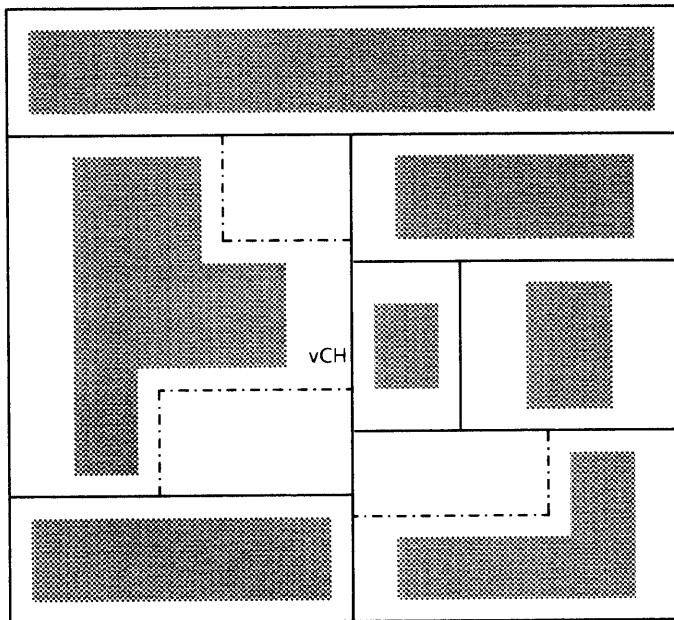
### 2.6.6.2 Restricted Grow

This grow is a direct generalization of **Grow0**, which is specified by a reference channel and the two intersecting channels (or intersections) bounding the region to be split. The generalization is that these bounding channels may include pseudo-channels of appropriate directions. *Pseudo-channels* (see Figure 2-22) are associated with the edges of a free missing corner of a block, and potentially become real channels when that corner is used.

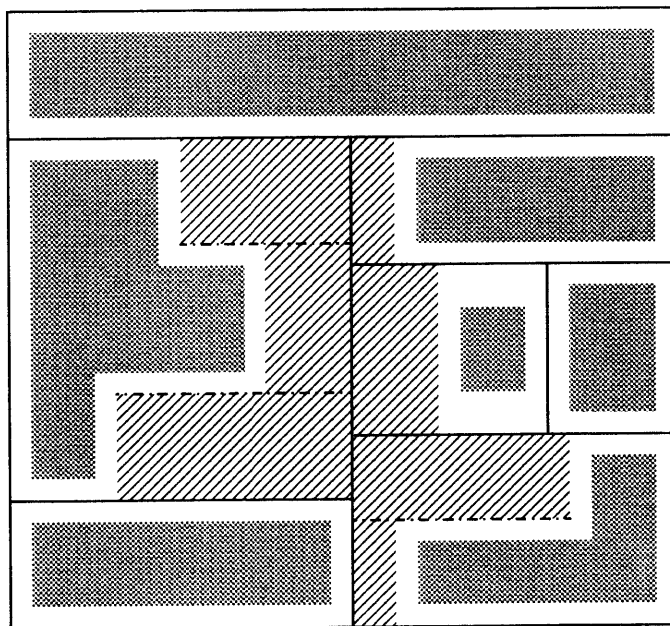
This restricted **Grow** is composed of **Grow0** followed by **FlexKnee**. The number of **FlexKnee**'s needed depends on the number of pseudo-channels used. (See Figure 2-23.) It may be necessary to apply **BreakCrosses** to remove any + -intersections between the bounding channels.

#### Remarks:

- ① It is clear that this **Grow** is restricted because it only forms rectangular holes.
- ② It does not enumerate all possible rectangular holes. (See Figure 2-24.)
- ③ It includes **Grow0**, **Grow1**, and **Grow2**.





(a) Channels are positioned from left to right and blocks are centered in their holes

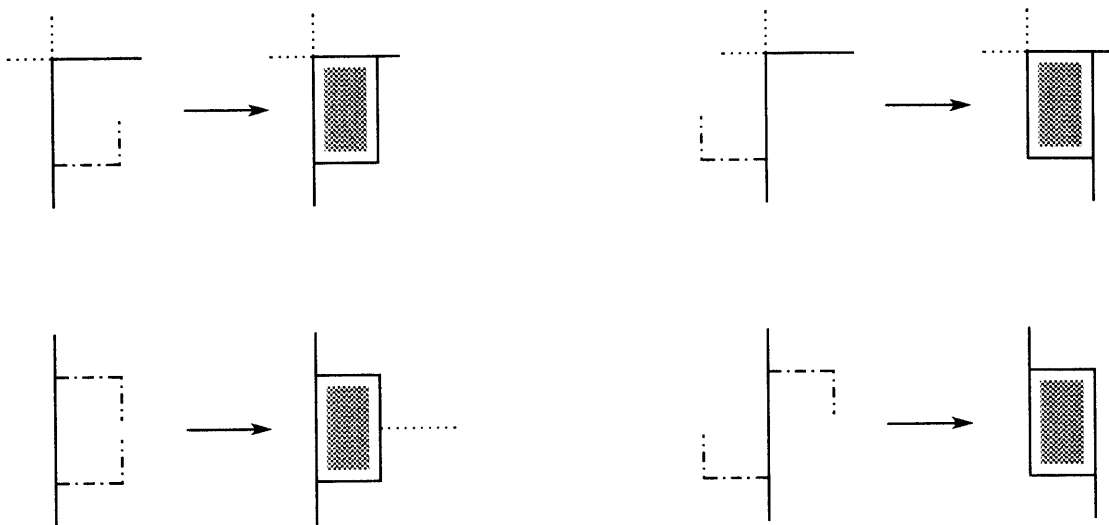


→ | ← Slack of vCH

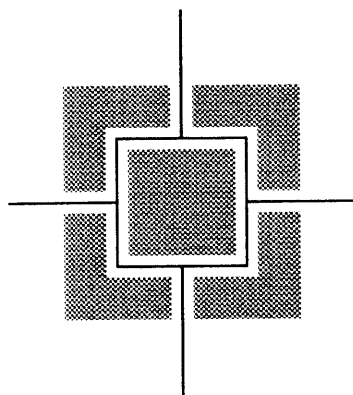
(b) Channels and blocks are moved away from vCH

 Generalized slack of vCH  
 Pseudo-channel

**Figure 2-22: Generalized slack**



**Figure 2-23:** *Some restricted grow operations using pseudo-channels*



**Figure 2-24:** *A topological hole formed from pseudo-channels only*

## 2.6.7 Miscellaneous Operations

This section describes various operations supported by the model. Again, all except **Save**, **Restore**, and the stack operations are first class operations.

### 2.6.7.1 Stack Operations

**StackSize** returns the number of frames in the undo (or redo) stack.

**GetTrack** returns a copy of the sequence of operations from a specified depth to the top of the redo stack.

**Redo** takes a sequence of operations from **GetTrack** and returns the system (in an appropriate state) to the state described by the sequence, by successively applying the operations. (If the system is not in the proper starting state then some operation in the sequence will raise an error.)

**Undo** undoes a specified number (less than or equal to the stack size) of steps by successively applying and popping the operations on the undo stack. This returns the system to an old state. The redo stack is also popped to ensure that both stacks have the same size.

### 2.6.7.2 Other Operations

**SetBlockShape** changes the shape of the block to any specified shape. This operation is used in shape determination.

**SwapBlocks** swaps two blocks with respect to their topological holes. This can easily be generalized to permutation of any number of blocks. It is useful for placement improvement.

**OrientBlock** changes the orientation of a block to any of the eight possible orientations. This operation includes mirroring and rotating the block.

**Remarks:**

- ① The above three operations are their own inverses.
- ② In any of the three operations, if a block does not fit its assigned topological hole, the hole can be reshaped using **ClearInteriorKnee**.

**SplitHole** splits a topological hole into two by adding a channel and replacing the block by two blocks. Depending on where the channel is added, the final number of channels may increase by one, decrease by one, or not change at all.

**MergeHole** is the functional inverse of **SplitHole**. It merges two holes by removing part of a channel and replacing the two blocks by an appropriate block. The resultant hole must be a **CQ1**. These two operations are used in floorplanning.

**Save** creates a text file describing the system.

**Restore** takes the above text file and reconstruct the system.

**2.6.8 Topological Operators Summary**

The topological operators described above are the basic tools for the placement problem. They modify the placement topology, and the resultant geometry is computed using **Geometrize**. The tracking and backtracking support for these operations makes them especially effective for addressing the placement problem.

## 2.7 Layout Model Summary

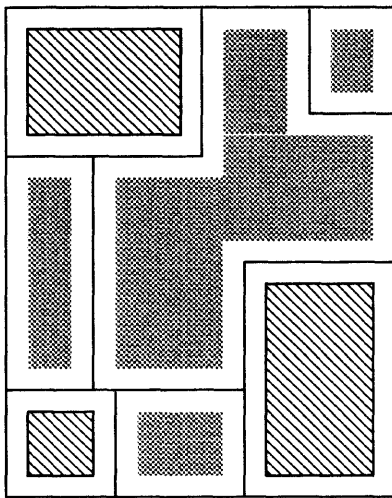
### 2.7.1 Other Layout Model Issues

Here are some issues not discussed in the previous sections.

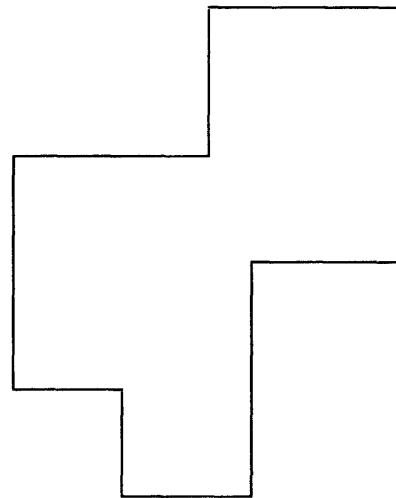
#### 2.7.1.1 Fake Blocks

Fake blocks are auxiliary blocks that are not part of the input. There are a number of uses for them:

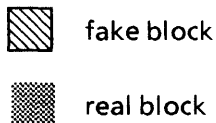
##### ① Corner Pads



Assembly with fake blocks to shape its corners



Shape of assembly



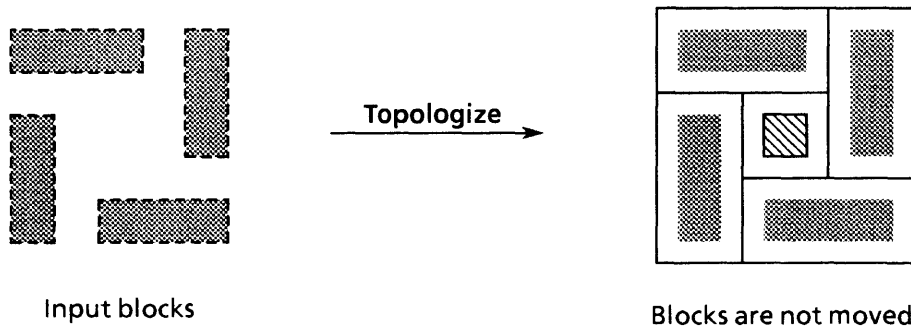
**Figure 2-25:** *Using fake blocks to shape corners of an assembly*

There is an asymmetry in that blocks can be non-rectangular in shape whereas the assembly must be rectangular. This is necessary to ensure a small set of topological operations. This asymmetry can be patched by using fake blocks at the missing corners of

the assembly. (See Figure 2-25.) Operations on the assembly (including the fake blocks) are unchanged except that there must be some built-in precautionary mechanisms to prevent unintended displacement of a fake block from its corner.

② **More Accurate Placement**

One of the drawbacks of the topological model is that it may be hard to position the blocks accurately. This can be partially rectified by placing fake blocks in the assembly to assist in positioning the real blocks (at their desired locations). The price is increase complication in channel topology. Recall that in **Topologize** some of the blocks may have to be displaced to ensure that the channels formed do not intersect the blocks. (See Figure 2-9.) This problem can also be solved using fake blocks. (See Figure 2-26)



**Figure 2-26:** *Using fake block to augment Topologize*

**2.7.1.2 External Constraints**

Recall that in constructing a **CPG** for the assembly, the edges on the **CPG** correspond to two kinds of constraints: topological constraints and block constraints. It is possible to introduce a third kind of constraint, namely, the external constraint. This is a directed constraint between any two channels of the same orientation (horizontal or vertical) as long as it does not result in a cycle in the **CPG** when combined with other constraints.



### 2.7.1.3 A More Efficient Geometrize

The algorithm for **Geometrize** in Section 2.6 requires the **CPG's** to be constructed each time it is executed. A more efficient alternative is to reuse the same **CPG's**. This is simplified by the one-to-one correspondence between channels in the assembly and nodes in the **CPG's**. For each operation that affects the channels or the blocks it should also modify the **CPG's** accordingly and mark the affected nodes. When an update on geometry is needed, **Geometrize** needs to compute only the positions of those nodes affected. This is done by propagating the changes along the edges as far as necessary. On the average, this algorithm is faster than the earlier algorithm. The price is additional complexities in implementing the topological operations. Nevertheless, **Geometrize** must still update the incidence counts of all the nodes each time it is called. With the proper data structures this update can be done in  $O(\text{number of nodes})$  computations.

### 2.7.2 Layout Model Conclusion

This topological model, when coupled with a powerful set of topological operations with tracking and backtracking capabilities, is a versatile model for the layout problem. In the next chapter, the use of the operations and the power of the tracking and backtracking capabilities will be discussed.

**This page is intentionally blank**

## Chapter 3

### Placement Heuristics

The goal of this chapter is to show the power of the topological layout model. Heuristics for the placement problem are described. They are intended to suggest, not limit, possibilities. These heuristics have not been tested extensively; their effectiveness is best determined by empirical verifications.

#### 3.1 Problem Statement for Placement

Given a collection of blocks, some of which have variable dimensions. Place them on a plane and specify their dimensions so that no blocks overlap and all their interconnections can be routed. The objective is to minimize the layout area and the interconnection length.

The positioning of the blocks is called *placement*; specifying their dimensions, *shape determination*; and connecting the interconnections, *routing*. In this thesis, placement includes both placing the blocks and specifying their dimensions. Routing is discussed in the next chapter.

#### 3.2 Placement Overview

In the hierarchical layout method, placement is usually classified into bottom-up and top-down. *Bottom-up placement* combines sub-blocks into a block, and blocks into a super-block. It proceeds from a lower to a higher level, with the dimensions of all combining sub-blocks known at each level. For *top-down placement*, the situation is reversed. Blocks are placed within a super-block, whose dimensions have been determined earlier. Since these blocks have not been constructed yet, their shapes can only be approximated from known

information about their interiors. These approximated shapes become constraints for placing and shaping the internal sub-blocks, propagating the shape constraints downwards. Both cases presume an underlying hierarchy, which is either given as part of the input or derived from it. In the former, the hierarchy usually corresponds to the hierarchy of the logic design, and in the latter, the hierarchy is usually derived from the interconnections between the blocks.

A large layout problem is often divided into many smaller sub-problems — divide-and-conquer. This subdivision may or may not involve a hierarchy. One technique is floorplanning. *Floorplanning* partitions the layout surface into regions and assigns blocks to them. Blocks in the same regions are placed independently of the other blocks, reducing the size of the problem. This technique is often used in top-down placement though it can also be used for bottom-up placement.

Placement is also classified into initial placement and placement improvement. *Initial placement* refers to adding blocks to a given seed placement or an empty assembly. Initial placement can be random (arbitrarily placing the blocks) or constructive (adding one block at a time to the assembly). *Placement improvement* refers to improving on a given placement. The needs for placement improvement decreases with the quality of initial placement, and is superfluous if the initial placement is optimum.

An underlying assumption is that there is a cost function (or figure of merits) that evaluates how good a placement is. All the search techniques in this chapter are designed to optimize the placement by minimizing the cost function.

### 3.3 Review of Search Techniques

Heuristic search is based on one of the oldest and most often used problem solving techniques — trial and error. In solving optimization problems, its effectiveness is between that of trial and error and that of algorithms that give the optimum answer. Neither guarantees getting the optimum answer in a short time; the more time that is available the more likely it is to get a better solution. Heuristic search is often used when there is no known algorithm for finding the optimum solution in a reasonable amount of time. This is the philosophy in heuristic search — try the best and hope it is acceptable.

This section describes a number of search heuristics. The most basic is the local search, a common technique used in the past. The remaining heuristics are just extensions of local search. First, some basic terminology is introduced.

#### 3.3.1 Basic Search Concepts

In a search problem, the following will be assumed to be given:  $\mathbf{F}$ , the search space (the set of all feasible solutions) and  $c$ , the cost function which maps the search space into the real line or the integers. Elements of  $\mathbf{F}$  will be referred to as *points* or *solutions*, and the cost of a solution,  $s$ , will be denoted by  $c(s)$ .

To formalize search, the notion of neighborhood is introduced. Intuitively, the *neighborhood* of a point  $s \in \mathbf{F}$  is a set of points close to  $s$ , and is denoted by  $\mathbf{N}(s)$ . The points in  $\mathbf{N}(s)$  are called *neighbors* of  $s$ . A *better neighbor* of  $s$  is any  $t \in \mathbf{N}(s)$  such that  $c(t) < c(s)$ .

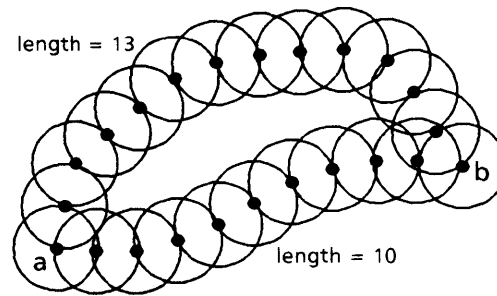
To generate these neighbors, a perturbator is used. This is the *neighborhood function*,  $\mathbf{N}$ , which generates a list of neighbors of  $s$  in arbitrary order. The size of this list will be denoted by  $|\mathbf{N}(s)|$ . (See [5] for a more thorough treatment on local search and neighborhood.)

#### Remarks:

- One way to represent a neighbor of a point is by the sequence of operations needed to go from the point to the neighbor.

- For any two distinct points,  $a$  and  $b$ , the following properties are assumed:
  - ① If  $a$  is a neighbor of  $b$  then  $b$  is a neighbor of  $a$ .
  - ② There exists a sequence of points from  $a$  to  $b$  such that consecutive points in the sequence are neighbors.

This sequence is called a *path*, and a *step* in the path corresponds to going from one point in the path to the next. There may be more than one path from  $a$  to  $b$ . (See Figure 3-1.) The *length* of the path is the number of steps needed to traverse the path.



○ • A neighborhood

**Figure 3-1:** Paths between two points in a search space

- A distinction should be made between evaluating and estimating the cost associated with a neighbor. Unless the evaluation is incremental, to *evaluate* the cost of a neighbor requires visiting the neighbor, applying the cost function, recording the cost, and then using the undo capability to return to the previous point. The undo and redo capabilities are essential here. A lot of computations are reduced if the cost associated with a neighbor can be evaluated incrementally. An example of this is to use an *estimator* which predicts the change in cost, based on the path between the two points. It is hard to find a good estimator.

### 3.3.2 Local Search

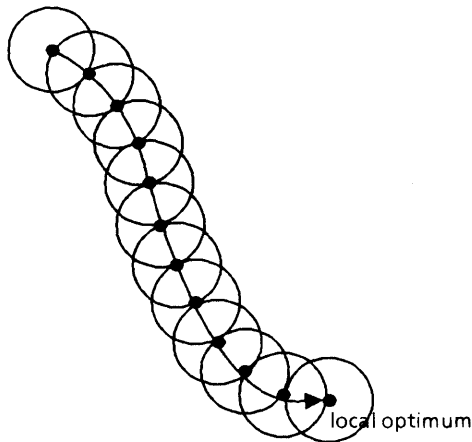
```
PROCEDURE LocalSearch[s]
1.  $t \leftarrow s$ ; -- initialize t to starting position, s
2. WHILE Improve[t] # "no" DO
    $t \leftarrow \text{Improve}[t]$ ;
   ENDLOOP;
3. RETURN[t];
END LocalSearch
```

**Figure 3-2:** *Local search algorithm*

The idea behind *local search* is this: start from an initial solution,  $s$ , improve on  $s$  by going to its first better neighbor. Repeat until a point with no better neighbor is reached. (See Figure 3-2.) The procedure for finding a better neighbor is shown in Figure 3-3. Figure 3-4 depicts a conceptual view of this search technique.

```
PROCEDURE Improve[t]
1. FOR  $n \in N(t)$  DO
   IF  $c(n) < c(t)$  THEN RETURN[n];
   ENDLOOP;
2. RETURN["no"]; -- No Improvement
END Improve;
```

**Figure 3-3:** *Improve routine*



**Figure 3-4:** *Conceptual view of local search*

### 3.3.3 Steepest Descent

This is an attempt to do something better than local search. Instead of going to the first better neighbor *steepest descent* chooses the best better neighbor. The algorithm is the same as local search (Figure 3-2) except that **Improve** (Figure 3-4) is replaced by **BestImprove** (Figure 3-5). Whether it actually gives a better final solution than local search remains to be

```

PROCEDURE BestImprove[t]
1.  $b \leftarrow t$ ; -- initialize best neighbor
2. FOR  $n \in N(t)$  DO
   IF  $c(n) < c(b)$  THEN  $b \leftarrow n$ ;
   ENDLLOOP;
3. IF  $c(b) < c(t)$ 
   THEN RETURN[b]
   ELSE RETURN["no"];
END BestImprove;

```

**Figure 3-5:** *BestImprove routine*

seen. When compared to steepest descent, local search saves a lot of time because it accepts



the first improvement in cost, whereas steepest descent always evaluates the entire neighborhood.

### 3.3.4 Look-ahead Search

Like other greedy heuristics, local search and steepest descent are easily trapped by local minimums. This problem can be mitigated by increasing the depth of the search. One way of doing this is to use a more complicated neighborhood function to generate neighborhoods containing more points. Another approach is to use the same neighborhood function but more frequently — instead of just looking at the neighbors of a point, look at the neighbors of its neighbors, and so on. (Unless extra processing is taken, some points will be considered more than once.) The latter approach is the idea behind look-ahead search.

*Look-ahead search* has the same structure as local search with the **Improve** routine replaced by **LookaheadImprove**. (See Figure 3-6.) It always chooses the first better

```
PROCEDURE LookaheadImprove[t]
1. FOR n ∈ N(t) DO
   IF cℓ(n) < c(t) THEN RETURN[n];
   ENDOOP;
2. RETURN["no"]; -- No Improvement
END LookaheadImprove;
```

**Figure 3-6:** *LookaheadImprove routine*

neighbor, except that the look-ahead cost function,  $c^\ell$  is used in place of the immediate cost function to evaluate the neighbors.  $c^\ell$  is characterized by the parameter,  $\ell$ , the number of steps look-ahead. ( $\ell \geq 1$ , and  $c^1$  is same as the immediate cost function,  $c$ .)

The idea in look-ahead evaluation is that a point,  $t$ , is evaluated by computing the minimum cost over all points that can be reached from  $t$  in no more than  $(\ell - 1)$  steps. (See Figure 3-7.) Hence, to compute  $c^\ell(s)$  approximately  $\|N(s)\|^{(\ell - 1)}$  points have to be evaluated

```

PROCEDURE  $c^{\ell}[t]$ 
1. IF  $\ell = 1$  THEN RETURN[ $c(t)$ ]; -- immediate cost
2.  $a \leftarrow c(t)$ ;    -- initialize look-ahead cost
3. FOR  $n \in N(t)$  DO
    IF  $c^{\ell-1}(n) < a$  THEN  $a \leftarrow c^{\ell-1}(n)$ ;
    ENDOLOOP;
4. RETURN[ $a$ ];
END  $c^{\ell}$ ;

```

**Figure 3-7:** *Look-ahead cost function*

(assuming every point has approximately  $N(s)$  neighbors).

**Extensions:**

- ① Instead of going to the first better neighbor, a variation is to choose the best better neighbor (analogous to steepest descent).
- ② One major problem with the above look-ahead evaluation is that often  $\|N(s)\|$  is very large, so  $\ell$  has to be small to keep the computations manageable. A patch is to introduce a new parameter,  $b$ , to bound the breadth of the evaluation. This is the *bounded look-ahead cost function*,  $c_b^{\ell}$ , where  $b \geq 1$ . It only looks beyond the  $b$  neighbors with the smallest immediate costs. (See Figure 3-8.) The assumption in the bounded look-ahead cost function is that the immediate costs of a point and of its neighbors are correlated. To compute  $c_b^{\ell}$  of a point, approximately  $b^{\ell-1}$  points are evaluated.

The bounded look-ahead evaluation is an example of using pruning to reduce the size of a problem. The advantage of pruning is the possibility of increasing the depth of the search, and the price paid is the possibility of pruning away some good solutions. In bounded look-ahead evaluation there is a tradeoff between  $b$  and  $\ell$ , the search can be further generalized by allowing  $b$  and  $\ell$  to vary during the search.

```

PROCEDURE  $c_b[t]$ 
1. IF  $l = 1$  THEN RETURN[ $c(t)$ ]; -- immediate cost
2.  $a \leftarrow c(t)$ ; -- initialize bounded look-ahead cost
3.  $B \leftarrow$  (b points in  $N(t)$  with the smallest immediate cost)
4. FOR  $n \in B$  DO
   IF  $c^{(l-1)}_b(n) < a$  THEN  $a \leftarrow c^{(l-1)}_b(n)$ ;
   ENDOOP;
5. RETURN[ $a$ ];
END  $c_b$ ;

```

**Figure 3-8:** *Bounded look-ahead cost function*

- ③ The look-ahead search is highly conservative. It makes only one step towards an improvement though the cost evaluation traverses further. A variation is to move more steps towards the improvement.

### 3.3.5 Multi-path Search

A major shortcoming of the search heuristics described earlier is that they only give one solution. (Multiple solutions are desirable because the cost function may not capture every aspect of the placement problem.) Each of these searches basically traverses a linear path until it is trapped by a local optimum. The results obtained depend strongly on the starting points used. One way of getting many solutions is to repeat the search using different starting positions. The more trials, the more likely it is to obtain better solutions. Another approach is the multi-path search.

Previous searches can choose only one branch to explore further (e.g., the first in local search and the best in steepest descent). The choice may be bad. In a multi-path search, multiple branches are explored systematically (e.g., breadth-first or depth-first). A depth-first version is described in Figure 3-9.

```

PROCEDURE MultipathSearch[s]

1. B ← (first b better neighbors of s); -- ordered list of branches to explore

2. IF B = ∅
   THEN K ← K + s; -- add local optimal to K, the "collector"
        -- K only retains the k best local optimums
   ELSE FOR n ∈ B DO
        MultipathSearch[n]; -- search branches depth-first
   ENDOLOOP;

END MultipathSearch

```

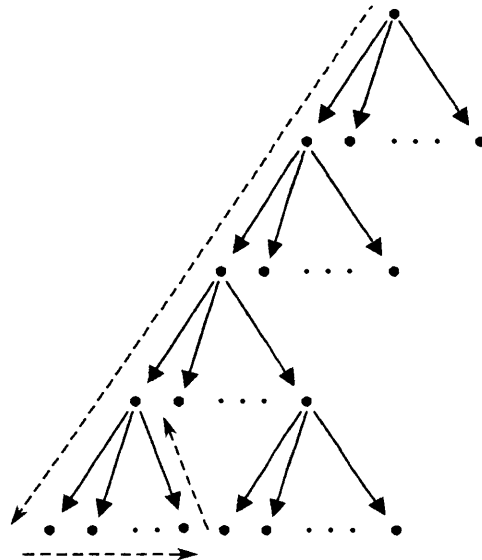
**Figure 3-9:** *Multi-path search (depth-first)*

The search is characterized by *b*, the branch factor, and *k*, the number of solutions kept. The first *b* better neighbors are explored further, they are recursively explored in a depth-first manner. When a local optimum is found, it is added to the collector, *K*, which only retains *k* solutions with the least costs. (Note that *B* is a local variable of the *MultipathSearch* procedure in Figure 3-9, whereas *K* is a global variable that has been initialized to the empty set before starting the search.)

Only local optimums are collected because they cannot be readily improved. (Also see adaptation of this search technique to initial placement in Section 3.4.3.) Figure 3-10 depicts a conceptual view of this search.

**Extensions:**

- ① Again analogous to steepest descent, the best *b* better neighbors can be explored instead of exploring the first *b* better neighbors.
- ② Look-ahead cost function or bounded look-ahead cost function can be used in place of the immediate cost function.
- ③ In collecting the solutions, a check may be added to ensure that the solutions retained are significantly different (e.g., check that the costs are different).



Search tree is traversed depth-first, then left to right

**Figure 3-10:** *Conceptual view of multi-path search (depth-first)*

- ④ One common problem with multi-path search is that the size of the search tree grows exponentially; the search may not terminate by itself in any reasonable amount of time. There are two remedies: abort the search prematurely and keep what has been collected, or use pruning. Pruning, if properly incorporated, is preferred to aborting the search. The latter can be viewed as a special kind of pruning.
- ⑤ This search technique can be improved by combining depth-first with breadth-first in the search. A breadth-first search explores all points on the same level before exploring the next level (some pruning may be needed). The problem with breadth-first search is that if the good solutions lie far from the initial position it will take a very long time for the search to reach there. Whereas if a depth-first search makes a bad choice at the beginning of the search then it will take a long time to rectify the situation.

### 3.3.6 Summary on Search Heuristics

Four search techniques have been described, and many variations on them have been suggested. These searches may be used at a micro level (e.g., to shape a block) or at a macro level (e.g., to add many blocks to an assembly). The undo and redo capabilities of the layout model are essential for implementing these search techniques or any other complicated searches that need tracking and backtracking. These capabilities also make the keeping of many solutions feasible. (A search technique that is not discussed here is the *simulated annealing*. See [2] for an overview.)

### 3.4 Placement

This section describes the application of the search techniques in Section 3.3 to the following placement problems: shape determination, floorplanning, initial placement, and placement improvement. Only a high-level and sketchy discussion is presented, since the purpose of this section is to show how the search techniques are used.

#### 3.4.1 Shape Determination

This is part of the top-down design process. The goal of shape determination is to specify the dimensions of the soft blocks. *Soft blocks* are those blocks whose dimensions may be changed. They correspond to those circuit components that have not been designed yet. The dimensions of these blocks then become constraints for designing (placing and shaping) their constituent sub-blocks. These constraints may become input to a program or used by a circuit designer. When there are conflicts in them, a few iterations of top-down and bottom-up design are needed before the conflicts are resolved.

The basic operation on the layout model used in shape determination is **SetBlockShape**. The shapes of soft blocks can only be estimated; the more information known about these blocks, the better is the estimation. The area of a soft block may be approximated based on estimation of the total area of its sub-blocks plus the area taken up by their interconnections.

Five types of constraints are suggested to model the shape determination problem. Each of them models the varying degrees of softness of a block. (See Figure 3-11.) In all cases the total area of the block remains unchanged and the resultant shape must be **CQ1**. These constraints are listed below:

① x-continuous and y-continuous — constant total area.

The shape may vary continuously along both horizontal and vertical directions.

② x-continuous and y-discrete — constant total length of horizontal stripes.

The shape can only vary continuously in the horizontal direction; the vertical dimensions must take discrete values.

③ x-discrete and y-continuous — constant total length of vertical stripes.

The shape can only vary continuously in vertical direction; the horizontal dimensions must take discrete values.

④ x-discrete and y-discrete — *Tile Model*. Constant total number of tiles.

Both horizontal and vertical dimensions can only take discrete values.

⑤ Rigid type — all dimensions are fixed.

These are the hard blocks and their shapes cannot be changed.

Extensions on these types include *union type* where the shapes must be one of the union of several other types, and *functional type* which can take arbitrary user-defined constraints and includes all others as special cases.

When a block is added to an assembly, shape determination may be used to match the shape of the block to the site where it is added. This involves searching for a good site to match a chosen block shape and finding a good shape to match a chosen site. The generalized slacks may be a good hint for the matchings. (See Figure 3-12.)

After all the blocks are placed, shape determination may be used to improve the geometry of the placement. The shapes of the soft blocks are perturbed to reduce the placement cost. Any search heuristics in Section 3.3 are directly applicable.

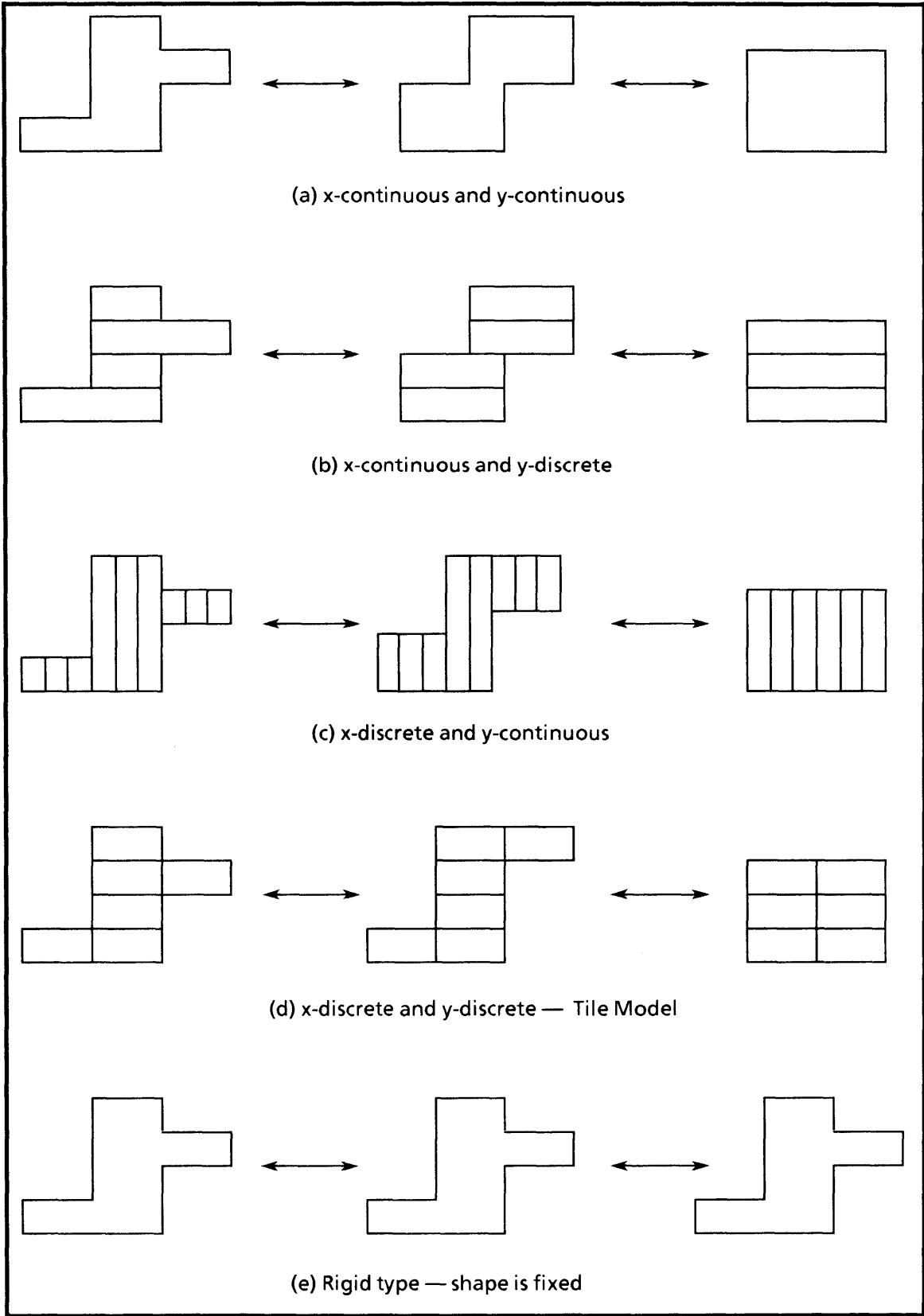
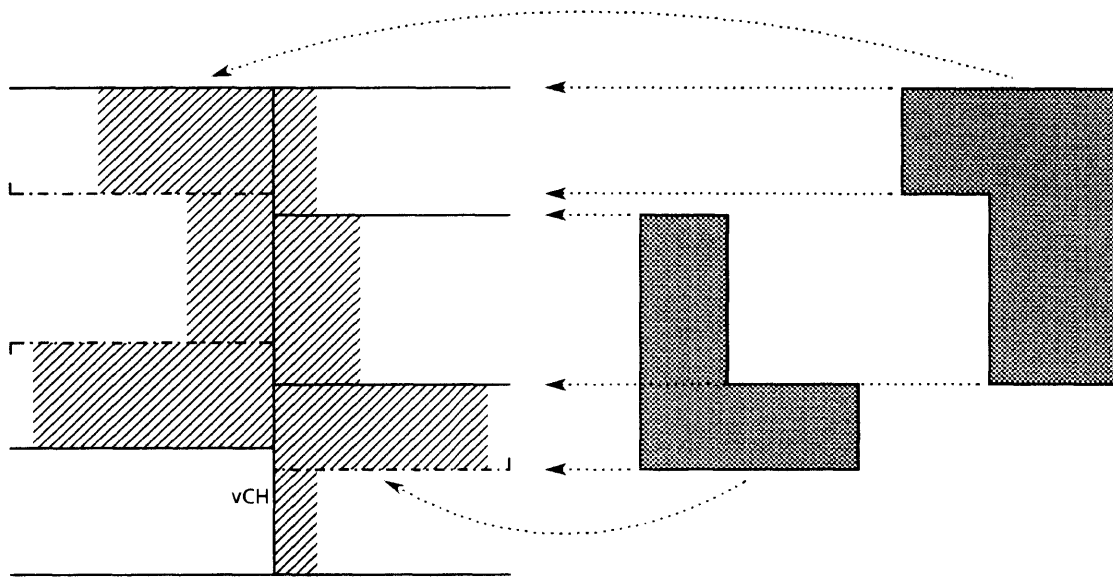




Figure 3-11: Five types of shape constraints





Two different shapes are formed based on the generalized slack

-  Generalized slack of vCH
-  Pseudo-channel

**Figure 3-12:** *Using generalized slack as hints for shape determination*

**Remarks:**

- ① There is another complication in shape determination. To model the fact that some components are used more than once in a circuit, the notion of class is used. All blocks in the same *class* must have the same shape (though they may be oriented with respect to each other); changing the shape of any block in a class affects the shapes of all other blocks in the class.
- ② Because the blocks are soft, it is not clear where the positions of their pins are. This makes the computation of net length difficult
- ③ When compared to rectangular blocks, **CQ1** have many more degrees of freedom in shaping them. This not only means more potential for doing things better, but also for

doing them worse. Additional structures may be needed to exploit the interactions between different types of blocks (e.g., hard & hard, soft & hard, and soft & soft), different classes of blocks, and different block orientations.

### 3.4.2 Floorplanning

The objective is to generate a floorplan of the layout. It may be used to impose a hierarchy on the layout or divide the problem into smaller sub-problems. The basic topological operation needed is a special case of **SplitHole** to split a rectangular hole into two rectangular holes. The topological model is especially suitable for floorplanning since the topological holes tile the whole layout surface of the assembly.

To generate a hierarchy, a min-cut algorithm is used. (See Figure 3-13.) This algorithm partitions a set of blocks into two sets of roughly equal sizes and minimizes the the cut count between the partitions. The *cut count* is the number of interconnection nets that connect some blocks between the two partitions. This min-cut algorithm is carried out recursively on the partitions until they are of the desired sizes. The result is a *min-cut tree*.

In a min-cut tree, each non-leaf node has two children. It is possible to generate a floorplan from the min-cut tree by applying a **SplitHole** operation for each non-leaf node in the min-cut tree. (See Figure 3-14.) Given a min-cut tree, there are many ways to generate a floorplan, since for each non-leaf node in the tree there are four ways of splitting the hole corresponding to the node. (See Figure 3-15.) Horizontal and vertical splittings are usually alternated to give a more natural looking floorplan.

#### Remarks:

- ① The algorithm in Figure 3-13 uses a local search to improve the cut counts; other search techniques may also be used.
- ② A more elaborate way of computing cut count is to consider the interconnections from blocks in other partitions. For example, when computing the cut count between blocks D

```

PROCEDURE Mincut[S]

1. A ← (first half of the blocks in S); -- arbitrarily assign half of S to A

2. B ← S - A; -- assign remaining blocks to B

3. cutCount ← (cut count of A and B);

4. improve ← TRUE;

5. UNTIL improve = FALSE DO
    improve ← FALSE;
    P ← (A × B); -- All possible pairing of blocks in A and blocks in B
    FOR each (a, b) ∈ P DO BEGIN
        move a to B and move b to A; -- swap (a, b)
        IF (cut count of A and B) < cutCount
            THEN -- found an improvement
                cutCount ← cut count of A and B;
                improve ← TRUE;
                EXITLOOP; -- exit inner loop
            ELSE
                move a to A and move b to B; -- undo swap
        ENDLOOP;
    ENDLOOP;

6. RETURN[A, B]; -- return the partitions

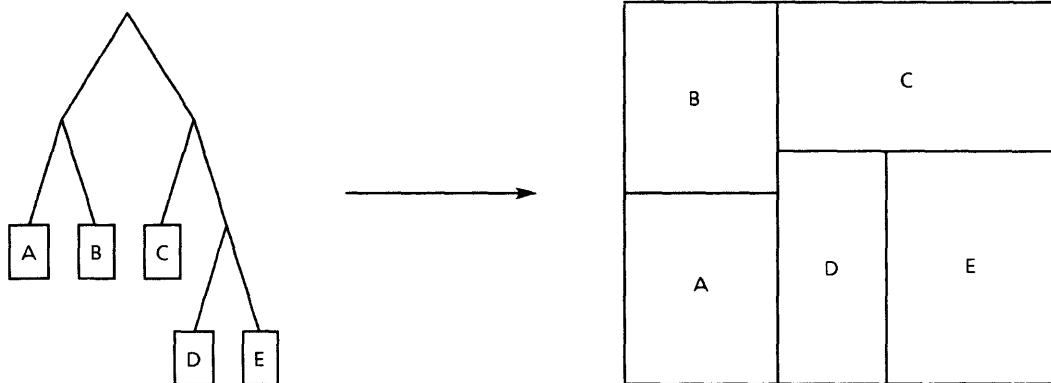
END Mincut;

```

**Figure 3-13:** *Min-cut algorithm*

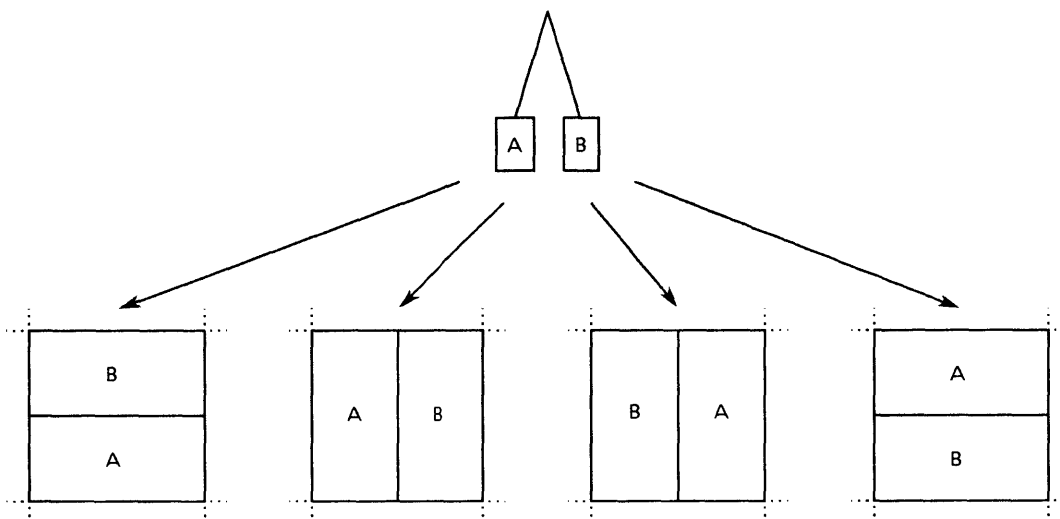
and E in Figure 3-14, the more elaborate computation takes into considerations the nets between blocks A and E, and B and E (because from the topology, blocks A and B are to the left of block E). For this to work the min-cut algorithm must be executed in parallel with the floorplanning process. Search techniques in Section 3.3 may be used to decide which one of the four ways of splitting a topological hole should be used.

- ③ There is no inherent reason why the min-cut tree must be a binary tree, though the ease in computing cut count is a factor. Generalization to non-binary min-cut tree and correspondingly producing non-binary splittings is possible. Another generalization is to allow non-rectangular holes in the floorplan.



**Note:** The area of a topological hole is equal to the area of the blocks in its partition

**Figure 3-14:** *Generating a floorplan from a min-cut tree*



**Figure 3-15:** *Four different ways of splitting a topological hole*

- ④ More complicated cut count computation can be incorporated to handle nets that connect more than two blocks.

### 3.4.3 Initial Placement

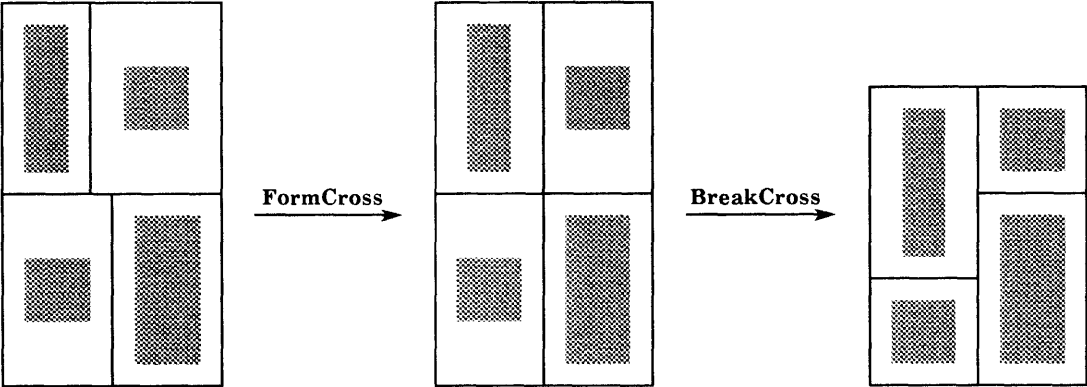
The goal of initial placement is to add blocks to an assembly containing zero (empty assembly) or more (seed placement) placed blocks. Initial placement can be random or constructive. A random initial placement is a quick attempt at getting a placement. The responsibility of arriving at a good placement is left to the placement improvement phase. A random initial placement can be constructed by randomly placing the blocks in the assembly without using the cost function or by using a program to generate valid placements from interpreting sequences of random numbers.

Constructive placement adds blocks to the assembly one at a time. Three approaches are suggested:

- ① One approach adds a small number of blocks, randomly to the assembly and applies the search techniques in Section 3.3 to improve the results. When a local optimum is reached, more blocks are added. This is similar to doing small scale placement improvements.
- ② A more natural approach involves modifying the notion of neighborhood. Search techniques in Section 3.3 are used, but each step now corresponds to adding one block to the assembly and the search terminates when all blocks are added. Only points with the same number of steps away from the initial point can be compared. The algorithm first orders the blocks. (This is the order in which the blocks are to be placed.) A multi-path look-ahead search is used to choose the best  $b$  next steps. (The notion of better neighborhood is not applicable because a point cannot be compared with its neighbors. But the notion of the best next steps is well defined.) Since at each stage, there may be a large number of possible next steps, the bounded look-ahead cost function is used. In addition, a combination of depth-first and breadth-first search is used.
- ③ Another approach is to use floorplanning to partition the layout surface until each region contains one block; this becomes the initial placement. Placement improvement is needed to improve the cost. This approach is proposed in [4].

### 3.4.4 Placement Improvement

This is the final phase of placement before routing the interconnections. The basic idea of placement improvement is to perturb the given placement to get a better result. The ease of finding an improvement decreases with the quality of the initial placement. For minor perturbations, some of the topological operations that can be used are: **FlipT**, **OrientBlock**, **BreakCross** and **FormCross** (see Figure 3-16), and so on, using the slacks of the channels as hints for choosing the proper operations. More major perturbations would be to reposition blocks using **SwapBlocks**, or selectively remove some blocks using **Shrink** and apply the initial placement heuristic to add them back to the assembly. The geometry can also be improved using shape determination. One comforting fact about placement improvement is that the worst it can do is to do no better.



**Figure 3-16:** *Using BreakCross and FormCross for placement improvement*

### 3.4.5 Other Placement Issues

Some issues ignored in the earlier discussions are placement of the input and output ports and external constraints on the shape of the assembly. In bottom-up placement, the ports are placed along the boundary of the assembly after placing the blocks. In addition, there is usually no constraint on the shape of the assembly, except that the cost function may prohibit

large layout area or unnatural looking assembly shape. In top-down placement, the ports have been placed and the shape of the assembly is determined from higher-level constraints. The cost function should take all of these factors into consideration.

### **3.5 Cost Function**

This is an important component of the optimization problem, but finding a good cost function is a hard problem in itself. There are two different views in implementing a good cost function. One view is that a good cost function should model what an experienced circuit designer thinks a good placement is. The other view is that it is quite meaningless to talk about the cost function of a placement without taking routing into consideration; a good cost function should predict the quality of the layout after all the interconnections are routed. In the first approach, getting a good cost function is hard because of the difficulties in modeling the judgment process of an expert in evaluating a placement. In the second approach the problem is also hard because of the coupling with routing.

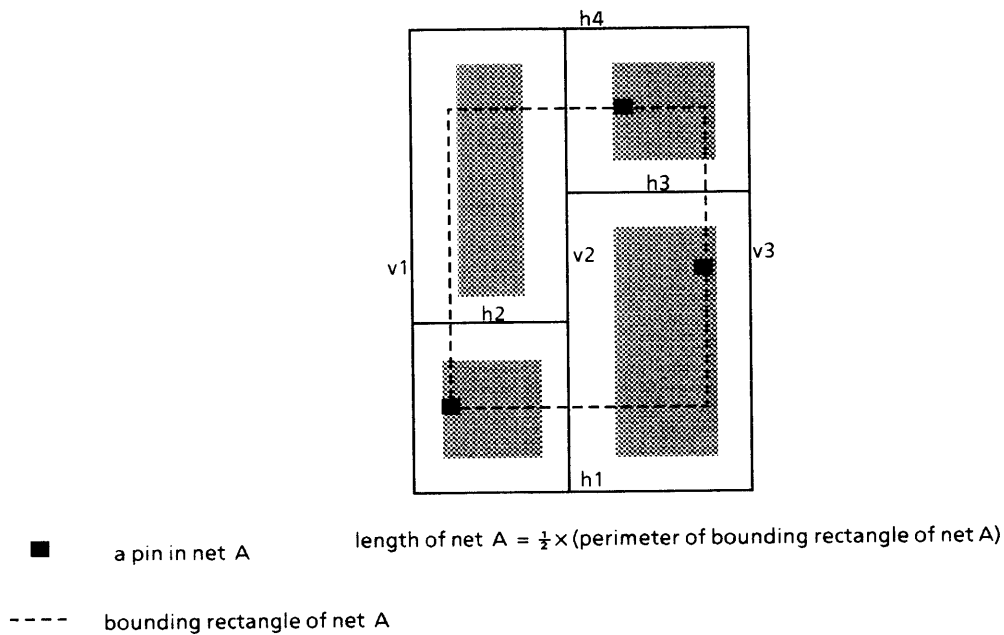
This section proposes some factors that a good cost function should incorporate. A possible cost function is a weighted sum of the following cost components:

- ① Total layout area
- ② User constraints
- ③ Total net length

The total layout area translates directly into fabrication cost. It is the area of the assembly (corrected for any missing corners). User constraints could be in the form of bounds on dimensions, on the ratio between the length and width of the assembly, on the length of critical nets, and so on. The total net length measures the sum of the interconnection lengths. Since the interconnections have not been routed, the net length can only be estimated.

### 3.5.1 Net Length Estimation

The total net length is the sum of individual net lengths. A common estimate of the net length is half the perimeter of the rectangle that bounds all pins in the net. (See Figure 3-17.)



**Figure 3-17:** *Estimating net length*

There may be added complications if the pin of a block corresponds to more than one physical pin. In any case, this is only a rough estimate of the actual length of the routed interconnections.

The penalty for large net length is the reduction in the speed of the circuit and more layout area needed by the interconnections. The latter may take up a significant portion of the layout area.

### 3.5.2 Channel-width Estimation

One nice property about the topological model is that if all the channel-widths can be estimated accurately the total layout area can also be estimated accurately. This may be too



difficult to achieve. A more reasonable objective is that if all the errors in the estimation of channel-widths cancel out, then the total layout area can still be estimated accurately. That is assuming that the router is smart enough to increase or decrease the channel-widths depending on the amount of space needed by the interconnections.

One way to estimate the channel-widths is to first assign default width to all channels and update the channel-widths as part of the cost function evaluation. This has the advantage of combining the interconnection and the layout area contributions into a single term. This approach changes the geometry when the cost function is evaluated. But care must be taken to ensure that the cost function is bounded and converges quickly. (Repeated applications of the cost function should not make the layout grow larger and larger, and if the cost oscillates, the oscillation should have small amplitude and high damping.)

The channel-width may be written as a weighted sums of a number of factors. These factors in order of increasing complexities are as follows:

① Local pin density.

This counts the number of pins on the edges of the blocks on both sides of the channel.

② Total net length.

This contributes a value that is a function of the total net length to every channel.

③ Net and channel overlap.

(For example overlap between the bounding rectangle of net A and channels h2, v2, and h3 in Figure 3-17.) This is an attempt to do something better by taking into considerations both the local and the global aspect of the problem.

④ Nearest neighbor routing.

This only routes the nearest neighbors. And is much simpler than routing all the blocks. The idea behind this is that if the placement is good then only the nearby neighbors have the most interconnections in common. Allowances can be made for nets that connect to blocks that are not neighbors.

⑤ Global routing of the interconnections.

Assign nets to channels without actually computing the exact positions of the nets. This gives a fairly accurate estimate of the channel-width but is probably too time consuming for large placement problems.

⑥ Actual routing of the interconnections.

Although the most accurate, this is too time consuming to be practical for most problems.

There is a tradeoff between the time spent to compute an accurate, but time consuming, cost and the time spent searching for placement solutions. One compromise is to use a simple cost function at the beginning and more complicated cost functions later to narrow down the better solutions.

### **3.6 Placement Summary**

The search techniques and the placement heuristics described are best thought of as tools for solving parts of the placement problem. Combining these tools effectively to address the problem requires more work. Another important aspects in using the search heuristics effectively is developing good pruning techniques.

Formulating the placement problem as an optimization problem is only one way of modeling the real problem. Often the optimization model is not a good one because the cost function neglects some important aspects in the actual problem. This is one of the arguments for getting multiple solutions and for having an interactive interface to study the solutions obtained (by playing back the sequence of operations). The interactive interface can also be used to edit the solutions obtained and fine tune the cost function.

The tracking and backtracking capabilities are essential for implementing the heuristics suggested here. With its efficient undo and redo capabilities, the model can support any reasonable placement heuristics but the effectiveness of these heuristics in solving real-life problems is best determined by empirical studies.

This page is intentionally blank

# Chapter 4

## Routing Heuristics

This chapter gives a quick overview of the routing heuristics. Refer to [7] for a more thorough treatment on the subject.

### 4.1 Problem Statement for Routing

Given the placement topology, the channel-widths, and the interconnections between the blocks, route the interconnection nets along the channels so that the nets do not intersect the blocks except to make connections with the blocks. The objective is to minimize the layout area and the length of the interconnection nets.

An interconnection net can only make connection at specific points of the block. These points are the *pins*. A *net* can be described as a set of pins, and to *route* a net is to connect up all the pins.

### 4.2 Routing Overview

Routing is the final phase of the layout problem. The part of the layout system that is responsible for this phase of the layout is the *router*. The output of the router are the final positions of the blocks and the complete description of routed interconnections.

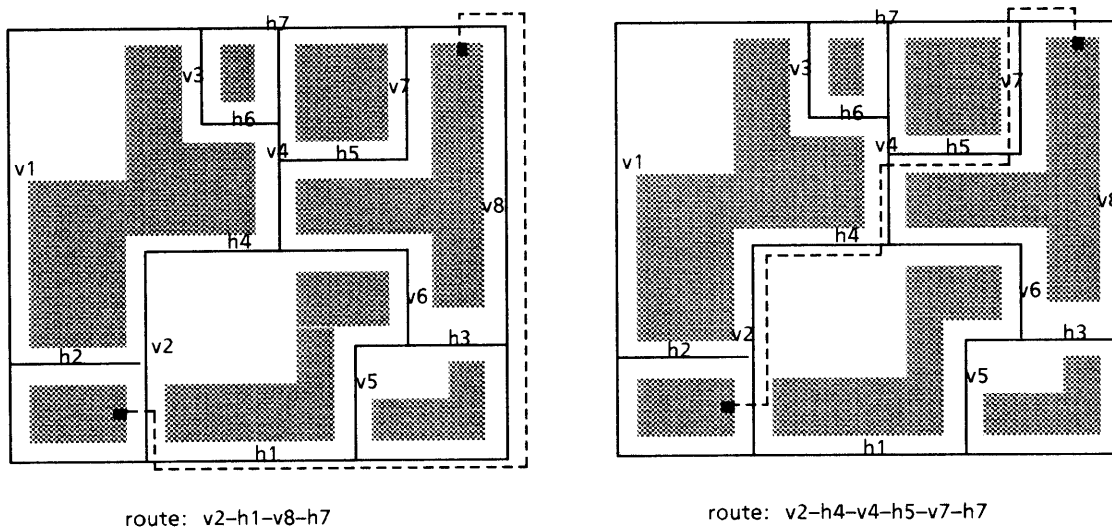
The router does not change the placement topology of the layout. However, if the channel-widths are inadequate to route the interconnection nets it may change them and reposition the blocks. A good router should consider the non-uniformity in the spaces on both sides of the channels, and should reduce the channel-widths if they have extra spaces.

Routing is divided into global routing and detailed routing. Global routing is executed

before detailed routing. Global routing is described in Section 4.3, and detailed routing, in Section 4.4.

### 4.3 Global Routing

This phase assigns nets to channels and prepares the nets for detailed routing. The assignment of nets to channels only defines the topology of the nets, but does not specify any coordinates. The exact positioning of the net is left to the detailed router. Global routing is also known as loose routing. Figure 4-1 shows two different global routings of a two-pin net.

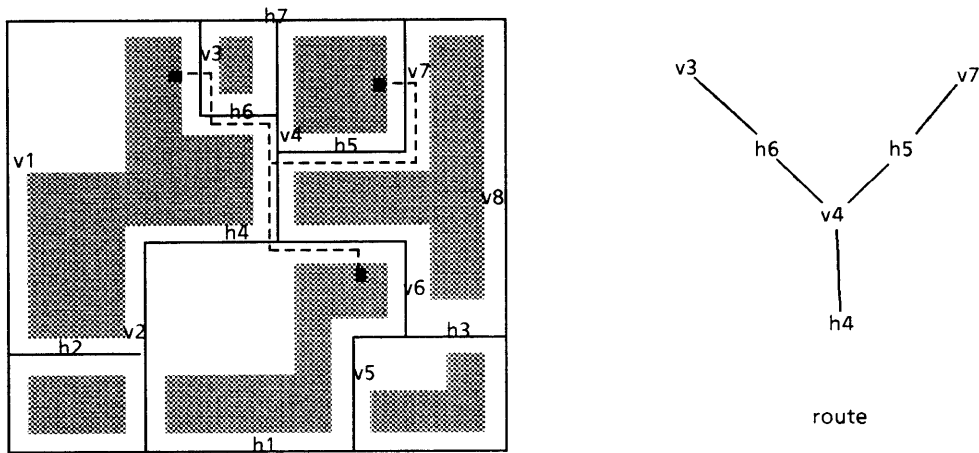


**Figure 4-1:** *Two different loose routes of a two-pin net*

The global routing phase topologically assigns all the nets to the channels so that the following objectives can be achieved in the detailed routing phase:

- ① The total layout area is minimized.
- ② The user constraints are satisfied (e.g., signal nets that are critical to the performance of the circuit are short).
- ③ The nets are well distributed to avoid congestion of the nets in some channels.

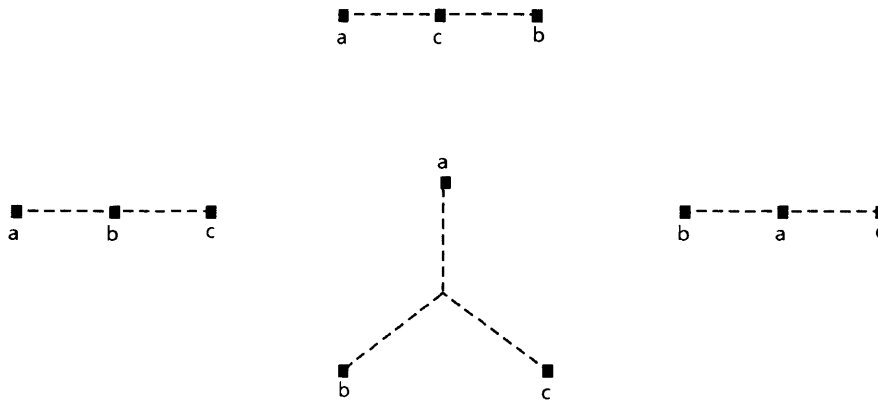
The topology of the intersections (between the channels) forms the underlying graph for global routing. This graph is the *Channel Intersection Graph* (CIG). The algorithm to route a two-pin net is equivalent to finding the shortest path between the two pins in the CIG. For a multi-pin net the global routing of this net is equivalent to finding the minimum Steiner tree connecting all the pins in the net. Figure 4-2 shows a global route of a three-pin net. The



**Figure 4-2:** *Loose route of a three-pin net*

complexity of finding a global route of a net grows quickly with the number of pins in the net. In addition, the number of different net topologies grows exponentially with the number of pins. Figure 4-3 shows the four possible topologies for a three-pin net.

Global routing all the nets is much harder than routing individual nets because of the interactions between the nets. The problem is often formulated as an optimization problem with a cost function that takes into consideration various factors affecting the routing quality including the three above. Again, as in the placement problem, search heuristics are used to find local optimal solutions. Similarly, global routing may be divided into initial routing and routing improvement.



**Figure 4-3:** *Four possible topologies of a three-pin net*

#### 4.4 Detailed Routing

In this final phase of routing, coordinates are assigned. Net segments are assigned to tracks on the channels. These segments are then connected up at the intersections. Finally blocks are positioned. All the geometrical information becomes the final output of layout. The detailed router does not change the global routing of the nets. Detailed routing is essentially derived directly from the global routes of the nets.

After the global routing phase, a fairly accurate estimate of the channel-widths can be obtained. Nevertheless, the channel-widths cannot be precisely determined until all the net segments are assigned to their tracks in the channels. Track assignment is done channel by channel in a specific linear routing order. If the channel-width is not sufficient to accommodate all the net segments in the channel, the channel-width is increased to create more routing tracks. The positions of the blocks and the channels have to be repositioned using **Geometrize**. The routing has to be backed up to an appropriate point and restarted. A few iterations of increasing channel-widths and routing may be needed before all the nets can be routed.



To ensure routing completion of all nets and termination of the routing algorithm, the channels must be routed in the order specified by the channel order constraint graph derived from the CIG.

#### 4.4.1 Channel Order Constraint Graph

A *Channel Order Constraint Graph* (COCG) is a directed graph in which every node corresponds to a channel in the assembly. Each edge in the graph correspond to a routing order constraint between the two corresponding channels. An edge from one node to another node indicates that the channel corresponding to the first node must be routed before the channel corresponding to the second node.

The COCG depends only on the channel topology and is not affected by changing the channel-widths or repositioning the channels and the blocks.

The basic idea behind an order constraint between two channels is that if changing the channel-width of one channel affects the track assignment of another channel (e.g., changing the relative positions of the pins on the sides of the channel) then the first channel must be routed before the second channel. This gives rise to an edge from the node corresponding to the first channel to the node corresponding to the second channel.

The COCG for a placement topology may contain cycles, in which case it is an indication of conflicts in the channel routing order. Special processing is need to resolve these conflicts to ensure 100% routing completion of the nets and termination of the routing algorithm.

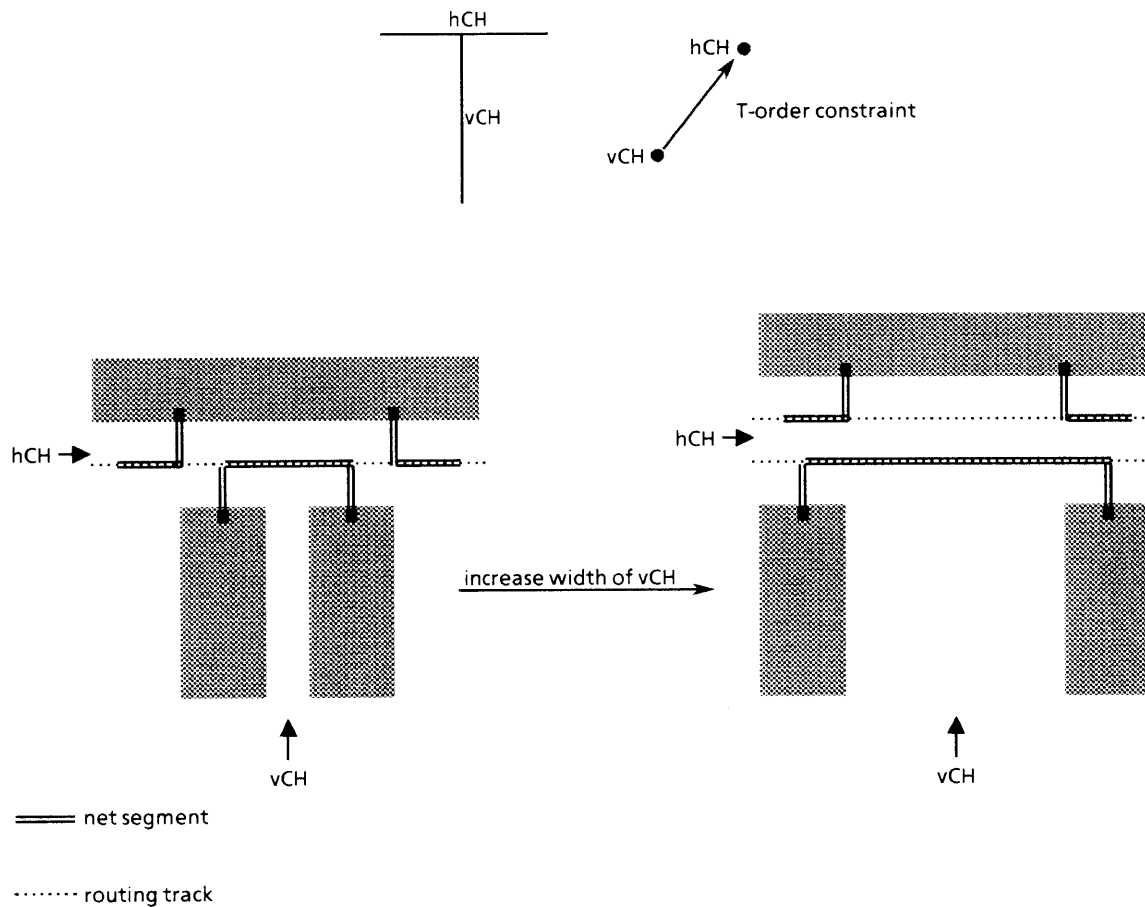
There are four kinds of edges in the COCG, each corresponding to a class of channel intersections:

- ① T-order constraint
- ② Generalized T-order constraint
- ③ L-order constraint
- ④ Generalized L-order constraint

These constraints are discussed below

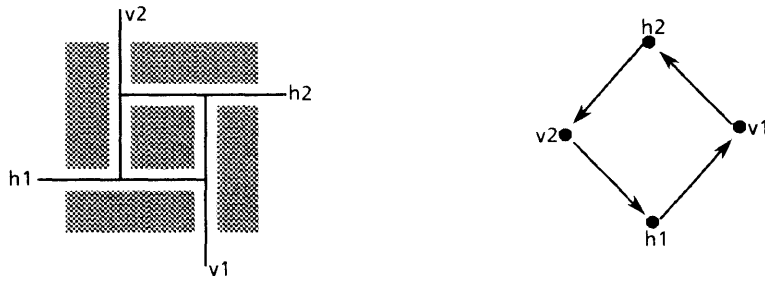
#### 4.4.1.1 T-order Constraint

This is the direct consequence of the fact that a channel (except if it is one of the four bounding channels) must always be routed before its bounding channels. This is because changing the width of the channel affects the relative pin positions of its bounding channels.



**Figure 4-4:** *T-order constraint*

Figure 4-4 shows how increasing the width of  $vCH$  increases the number of routing tracks needed to route the net segments in  $hCH$ . Figure 4-5 shows how four T-order constraints may form a routing order conflict loop.



**Figure 4-5:** *Channel routing order conflict loop*

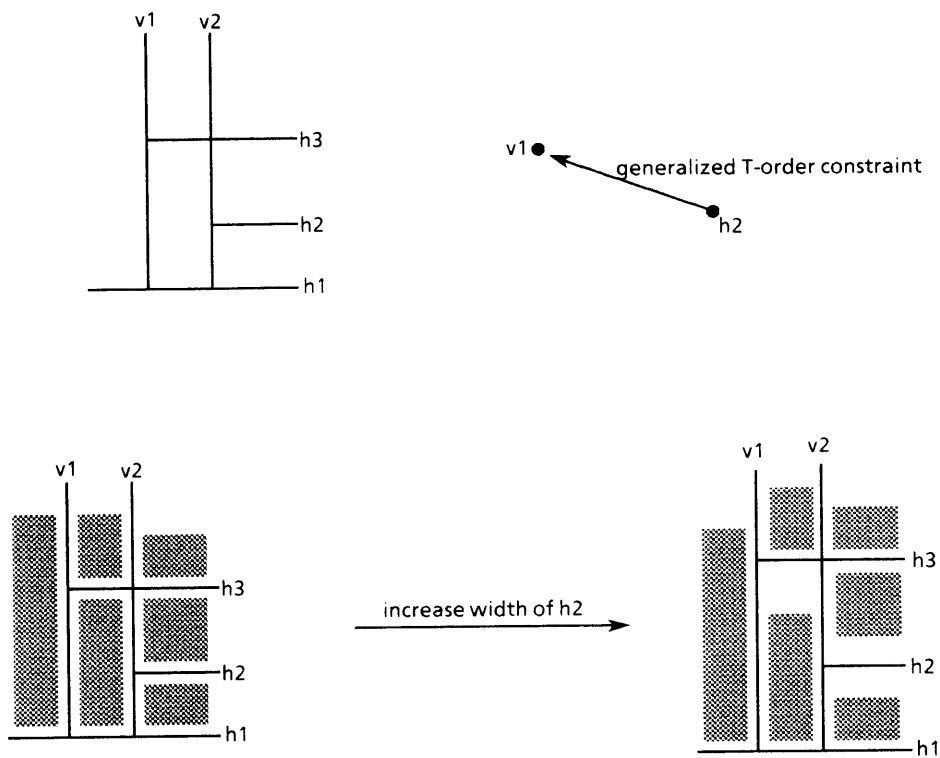
#### 4.4.1.2 Generalized T-order Constraint

A +-intersection does not impose a routing order between the two intersecting channels because changing the width of one channel moves blocks on both sides of the other channel by equal amounts. Thus, the tracking assignment for the second channel is not directly affected. The +-intersection, however, may interact with T-intersection giving rise to a generalized T-order constraint.

Figure 4-6 shows how changing the channel-width of h2 moves the position of the T-intersection between h3 and v1, changing the relative pin positions in v1. This imposes a routing order constraint between h2 and v1. In general, any channel, hCH, that intersects v2 but does not intersect v1 results in a generalized T-order constraint from hCH to v1.

#### 4.4.1.3 L-order Constraint

An L-order constraint can be viewed as a symmetric T-order constraint pair since changing the width of a channel affects the routing of the other channel and vice versa. Figure 4-7 shows how changing the width of vCH can affect the track assignment of hCH. By symmetry of the L-intersection, the converse is also true. Thus, corresponding to each interior L-intersection, an L-order constraint cycle is formed between the nodes corresponding to the



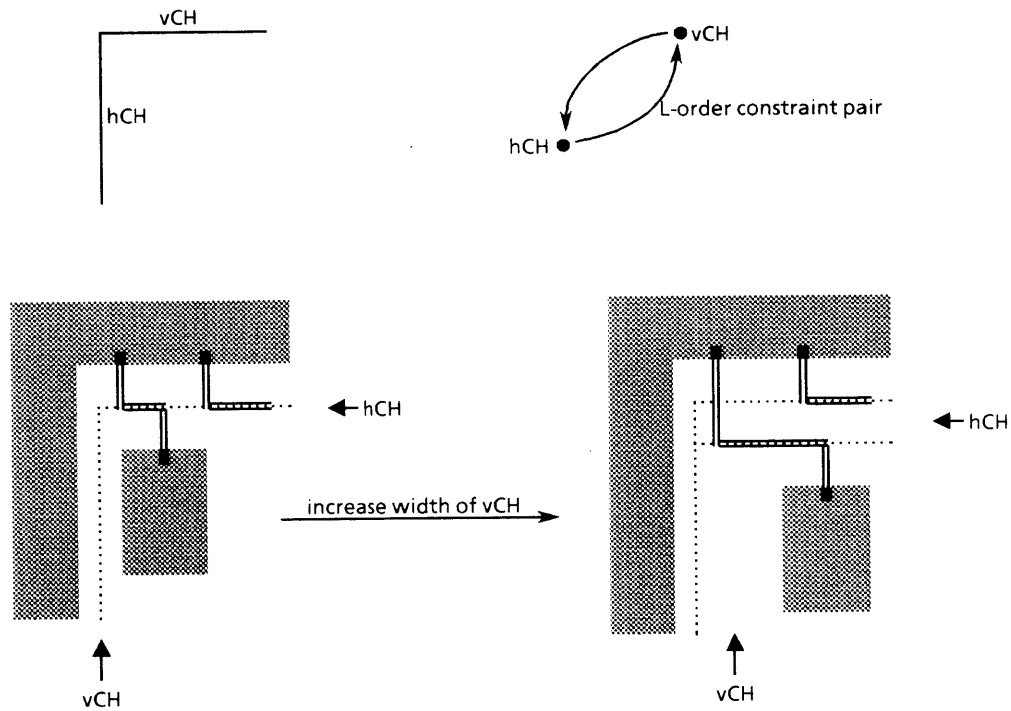
**Figure 4-6:** *Generalized T-order constraint*

two channels in the intersection. (Note that the L-intersections formed by the four bounding channels do not impose any order constraint.)

#### 4.4.1.4 Generalized L-order Constraints

The presence of interior an L-intersections in the topology not only forms a cyclic constraints between the two channels but may also affect the channel with an intersection next to the L-intersection.

Figure 4-8 shows how changing the width of h2 changes the relative positions between the two blocks to the left and to the right of v2 affecting the relative pin positions in v2. In

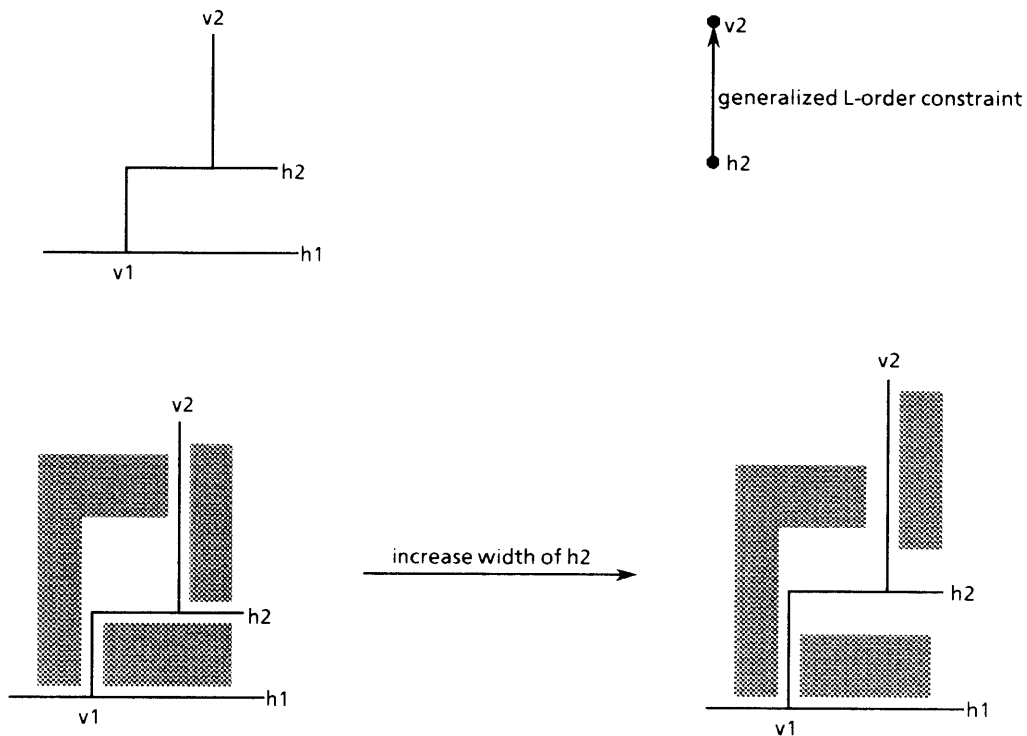


**Figure 4-7:** *L-order constraint*

general, any channel,  $vCH$ , that intersects  $h2$  next to the L-intersection ( $v1$  and  $h2$ ) with part of  $vCH$  above  $h2$ , causes a routing order constraint from  $h2$  to  $vCH$ .

#### 4.4.1.5 Routing Order Constraint Summary

In detailed routing, the channels must be routed sequentially. The routing order constraints are imposed by local channel intersections, forming a COCG. A cycle in the COCG indicates a conflict in the routing order constraints. The presence of interior L-intersections increases greatly the complexities of the COCG. Special heuristics must be used to handle these conflicts to ensure the routing completion of all nets and termination of the routing algorithm. These heuristics are discussed in Section 4.4.2



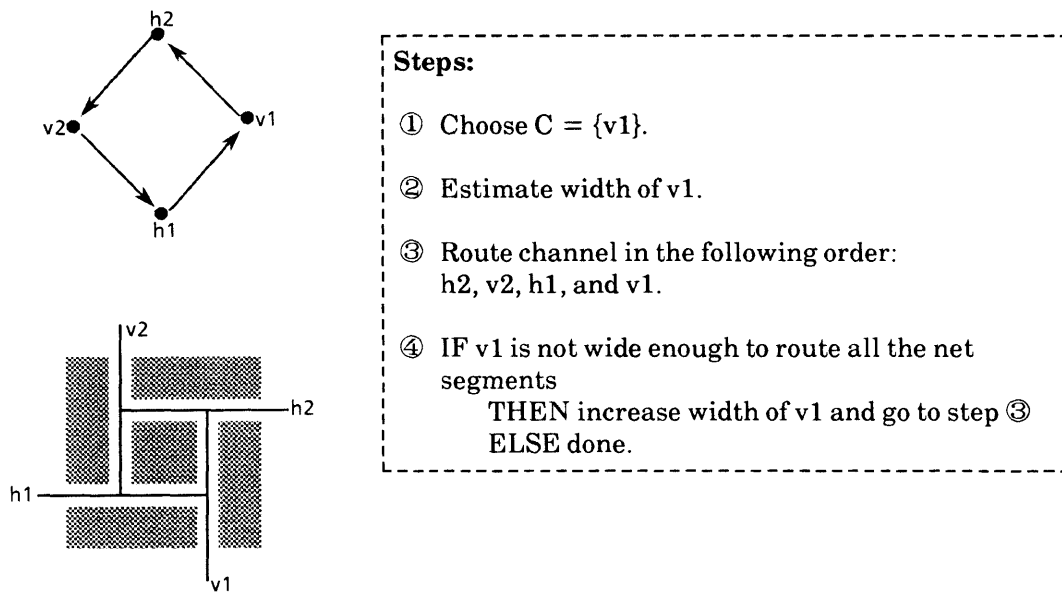
**Figure 4-8:** *Generalized L-order constraint*

#### 4.4.2 Resolving Routing Order Conflicts

There are two approaches to handling conflicts in the channel routing order. The first approach is to restrict the topology so that there are no cycles in the COCG. (See [11] on how this is done when all the blocks are rectangular.) This restriction may be imposed during the initial placement phase or the placement improvement phase. In general, this approach cannot be used if the blocks are convex quadrics. In order to use the space associated with a missing corner of a block, an interior L-intersection must be formed, resulting in a routing order conflict loop in the COCG. This approach is not very useful if the blocks are convex quadrics.

The second approach requires running the detailed router a number of times. The idea in

this approach is that if all the channel-widths can somehow be known or can be bounded then the channels can be routed in any order. The only reason there is a routing order is that if a channel is too narrow, then to ensure that all the net segments in that channel can be routed the channel-width has to be increased. This change in channel-width may affect the routing of other channels. So this approach basically selects a set of channels involved in the routing order conflicts and keeps increasing the widths of these channels until all nets can be routed. (Figure 4-9 shows an example on how a routing order conflict is resolved using this approach.)



**Figure 4-9:** Resolving channel routing order conflict loop example

The steps are as follows:

- ① Choose a set of channels,  $C$ , that are directly involved in the order constraint conflict.
- ② Using information from the global route and previous estimates, estimate the widths of all the channels in  $C$ .
- ③ Now using the COCG, route the channels as follows:
  - Nodes with zero incidence count (no edge pointing at them) are routed first and their channel-widths are determined exactly from the routing.

- A node can only be routed if the channel-widths of all its predecessors are either known (because they have been routed) or were estimated in ②.

The set **C** is chosen to be the smallest possible set of channels such that every loop in the COCG must contain a node from this set. If more than one such sets exist then the set with the least number of critical channels is chosen (because increasing the width of a critical channel increases the layout area directly).

Eventually the router will try to route some channel in **C**. If during track assignment it is found that the width of the channel is inadequate for the nets, then the channel-width is increased, and the routing is backed up to an appropriate point and restarted.

The above algorithm must terminate because only the widths of the channels in **C** are estimated and their estimation are increased if found to be inadequate. The increase in channel-widths of channels in **C** must terminate because the number tracks needed in any channel is bounded by the total number of net segments assigned to that channel.

#### **4.5 Routing Summary**

Routing has two phases: global routing and detailed routing. Global routing topologically assigns nets to channels and defines the basic routing topology of the interconnections. According to the assignments made in global routing, the detailed router positions net segments in the routing tracks of the channels without changing the global routing. Only after track assignment can the width of a channel be known exactly.

Routing order constraints arise in detailed routing because of the influence on the number of tracks required in a channel by the widths of other channels. This routing order is described by the COCG. The COCG depends only on the topology of the channels, and a cycle in the COCG indicates a routing order conflict. Routing order conflicts can be handled by iterating the detailed router and routing the channels in a specific order. 100% routing completion and termination of the routing algorithm can be assured.



## **Chapter 5**

### **Conclusions**

This chapter concludes the discussion on the topological approach. Section 5.3 contains suggestions for future research.

#### **5.1 Conclusion**

Phoenix is an extremely powerful research tool for studying the VLSI layout problem, in particular, the placement problem. The topological model strongly de-couples placement from routing. The channel-widths can be changed without affecting the placement topology, thus, guaranteeing 100% routing completion. The tracking and backtracking support for the topological operations allows the development of extremely complicated placement heuristics and makes the keeping of multiple solutions feasible. The simplicity in managing the undo and redo stacks, along with the extensibility of the operations makes the system highly programmable. The interactive interface of Phoenix and its ability to keep multiple solutions allows detailed study of search heuristics by replaying the sequence of derivation of the solutions. This feedback is invaluable for developing more effective placement heuristics and cost functions.

#### **5.2 Contributions**

This thesis is a systematic and thorough study of the topological approach for addressing the layout problem. A novel topological model that can handle convex quadrics effectively has been proposed. A complete and extensible set of topological operations supported by tracking and backtracking capabilities have been implemented. Powerful heuristics based on the

tracking and backtracking capabilities have been developed to address both bottom-up and top-down placement problems.

### **5.3 Suggestions for Future Work**

Many questions remain unanswered; some of them require extensive experimental studies and some require more research.

#### **5.3.1 Comparing Search Heuristics**

Many search techniques, with more extensions and variations, have been proposed in Section 3.3. Extensive experimental studies are needed to compare their effectiveness in solving actual layout problems. Some examples are local search with steepest descent, look-ahead search with cost functions of different look-ahead depth, multi-path search with repeated linear search (using different initial points), and look-ahead search that moves a different number of steps towards the best point evaluated.

It is easier to compare the effectiveness of the search heuristics by considering optimization problems with simple cost functions. An example is the problem of packing a collection of rigid blocks on a plane as closely as possible, using the area of their bounding rectangle as the cost function. This problem is not only interesting in itself but may provide insights into developing more effective search heuristics for the placement problem.

#### **5.3.2 Combining Layout Heuristics**

The placement heuristics in Chapter 3 (initial placement, placement improvement, floorplanning, and shape determination) and the routing heuristics in Chapter 4 (global routing and detailed routing) are only tools for addressing specific aspects of the layout problem. The problem of combining these tools effectively remains to be solved.

One way is to use floorplanning and shape determination in a top-down manner to determine shape constraints for the blocks. The shape constraints are propagated downwards

until all blocks at that level have already been designed. Initial placement and placement improvement are then used to place these blocks within the shape constraints, in a bottom-up fashion. If the shape constraints cannot be satisfied, then the top-down process has to be backtracked and repeated. A few iterations between bottom-up and top-down may be needed before the shape constraints can be satisfied at every level, with the blocks compactly placed. Initially, a simple cost function may be used to obtain many solutions. More elaborate cost functions are used in later stages to prune the solutions, and the final decisions are left to the designer.

Many variations on the above are possible. Finding the most effective way of combining these heuristics to solve layout problems is beyond the scope of this thesis.

### **5.3.3 Implementing Hierarchical Decomposition**

More research and implementation experience are necessary to determine the proper data and control structures needed to implement a truly hierarchical layout system.

### **5.3.4 Missing Corners**

More research and actual experience are needed to determine how circuit layout programs and circuit designers can use the missing corners in convex quadrics effectively to design components of such shapes. The question of using these shapes effectively cannot be answered until more layout programs and designers use circuit components with these shapes.

### **5.3.5 More Complex Shapes**

Extending the topological model to shapes more general than convex quadrics is suggested. An example is to consider C-shapes. One problem is that the channels between the upper and the lower arms of the C cannot increase in channel-width arbitrarily, unless the body of the C can be stretched.

### 5.3.6 Topological Routing Operations

This work has been confined to the topological operations on the model that affect the placement topology. An analogous study can be carried out to define and implement first class operations for global and detailed routing so that complicated search heuristics can also be developed for the router.

### 5.3.7 Solving Channel Routing Order Conflicts

One of the unsatisfactory aspects in the routing phase is that in the presence of routing order conflicts, the detailed router has to be iterated an unknown number of times to route all the interconnections. The shortcomings in this approach are as follows:

- ① The detailed router does more than just determining the channel-widths. For example, it actually routes the net segments when only values for the channel-widths are needed.
- ② The heuristics only *relax* (increase) the channel-widths but they do not *contract* (decrease) the width of a channel involved in the order conflict resolution even though the channel-width may be too wide. This is to ensure that the algorithm terminates. Nevertheless, there is still no easy way to predict the number of iterations needed before all the channel-width constraints can be satisfied.

The following suggestions are made to address these shortcomings. They are only suggestions, more research is needed to determine their feasibility.

- ① Instead of using the detailed router to determine the channel-widths, the channel-widths may be determined using a detailed router simulator (e.g., system of constraint equations) to simulate the detailed router without actually routing any nets. Hopefully, this will give a faster way of determining the channel-widths. The simulator should also output

sufficient information (e.g., track assignments) to ensure that the detailed router can route within the compute channel-widths.

- ② An alternative to the relaxation technique is the contraction technique which starts with over estimations of channel-widths (for those channels involved in resolving routing order conflict) and gradually reduce the channel-widths one by one while ensuring that all the interconnections can be routed, after each contraction. An advantage of using the contraction technique is that it can be terminated anytime depending on how much computation time one is willing to pay for a more compact layout.

This page is intentionally blank

# Appendix A

## Results

The results presented in this appendix are preliminary results obtained during the final week of my assignment. These results do not reflect the full potential of Phoenix.

### A.1 Results Overview

Since the router was not ready by the time I completed the assignment, complete layout examples could not be obtained. Nevertheless, to test the topological model and the placement heuristics, the following placement examples were considered:

- ① Three block-packing examples.
- ② Two floorplanning examples.

The rest of the appendix discusses these examples.

### A.2 Overview of Block-packing Examples

#### A.2.1 Block-packing Problem Statement

Given a collection of blocks, place them on a plane without changing their orientation so that they do not overlap and the area of the rectangle bounding all the blocks is minimized.

#### A.2.2 Block-packing Heuristics

The blocks are first sorted in a linear order. The blocks are then placed according to their order using search heuristics.

The search heuristics used is a multi-path search with look-ahead. (See the second approach to initial placement suggested in Section 3.4.3.) Each step in the search corresponds to placing one block onto the assembly. The search terminates when all the blocks are placed.

This search is characterized by the three parameters below:

- ①  $\ell$ , the depth (extent) of look-ahead.
- ②  $b$ , the branch factor of the search, which bounds the breadth of evaluation in the look-ahead cost function.
- ③  $n$ , the number of solutions retained during the intermediate stages of the search.

The search is a combination of depth-first and breadth-first. At each *stage*, the search looks at solutions  $\ell$  steps ahead with a branch factor of  $b$ . The best  $n$  of the  $b^\ell$  solutions evaluated are collected for further exploration in the next stage. Except for the last stage, the search always moves  $\ell$  steps per stage and explores the  $n$  solutions collected from best to worst (depth-first).

Pruning is not used in the search. The search can be forcibly terminated at any time by the user. The cost function used is the area of the rectangle bounding all the placed blocks.

### A.2.3 Discussion of Block-packing Examples

The examples used are modifications of the examples used to debug the system during the early stages of the implementation; they are not specifically constructed for the packing problem. The modifications made are as follows:

- ① All channel-widths have been set to zero.
- ② Blocks are enlarged slightly so that they fit together into a rectangle without any gaps inside. (Note: This information is not used in designing the packing heuristics.)

Three examples are tested. The examples are listed below:

- ① A nine-block example (page 100 to page 102).
- ② Another nine-block example (page 103 to page 106).
- ③ A twelve-block example (page 107 to page 123).

**Remark:** The system and the search heuristics have not been optimized yet.



## A.3 Overview of Floorplanning Examples

### A.3.1 Floorplanning Problem Statement

Given a collection of blocks and their interconnections, generate a floorplan of the layout so that each topological hole contains one block and the cut counts among the partitions are minimized.

### A.3.2 Floorplanning Heuristics

The heuristics is basically an implementation of the ideas in Section 3.4.2. The steps are listed below:

- ① Generate a binary min-cut tree.
- ② Use **Grow2** to add a fake square block to the empty assembly.

This block corresponds to the root node in the min-cut tree and has area equal to the total area of the input blocks.

- ③ Use **SplitHole** to split the square block vertically into two new fake blocks of the same vertical dimensions as the square block.

Each new block corresponds to a child of the root node and has area equal to the total area of all the blocks assigned to that child.

- ④ Step ③ is repeated recursively on the new blocks, alternating between vertical and horizontal cuts, until every block in the assembly corresponds to a leaf-node in the min-cut tree.
- ⑤ Replace the fake blocks with the corresponding real blocks

### A.3.3 Discussion of Floorplanning Examples

Two floorplanning examples are considered. The results are shown in page 125 to page 127.

#### A.4 Discussion of Results

The preliminary results from the block-packing examples show that (at least for the three examples considered) heuristic search can be used to pack about ten or less blocks effectively and achieve results close to an optimum packing.

After some study (using the interactive interface) on how the results are formed, I conclude that a reason why no optimum result is obtained is that the neighborhood function used is incomplete — the function only considers using **FormZ** and **TtoL** to compact the layout but does not try to use **FlexKnee**. A lesson from this is that the tracking and backtracking feature can also be used to study the cause of failure.

Another conclusion from the experiments is that having a large set of first class operations makes the system highly programmable. The implementation of the search heuristics for the packing problem is slightly more than one textual page of code, and the implementation of the floorplanning heuristics takes less than  $\frac{3}{4}$  page. Not counting the time spent on implementing the basic structures in the heuristics, I implemented and debugged the packing heuristics and the floorplanning heuristics in one evening. Moreover, by using a suitable cost function, the same heuristics can be used for the placement problem. The ease in programming an automatic layout system opens the way for circuit designers to develop layout heuristics specifically for their layout problems.

## A.5 Block-packing Results

### A.5.1 Block-packing Example One

- ① Number of blocks = 9.
- ② Total Area = 16800 unit<sup>2</sup>.
- ③ Search Parameters:
  - $\ell = 3$
  - $b = 8$
  - $n = 6$
- ④ Time taken  $\approx$  3 minutes. (Search was terminated by the user).
- ⑤ Pages: 101 to 102.

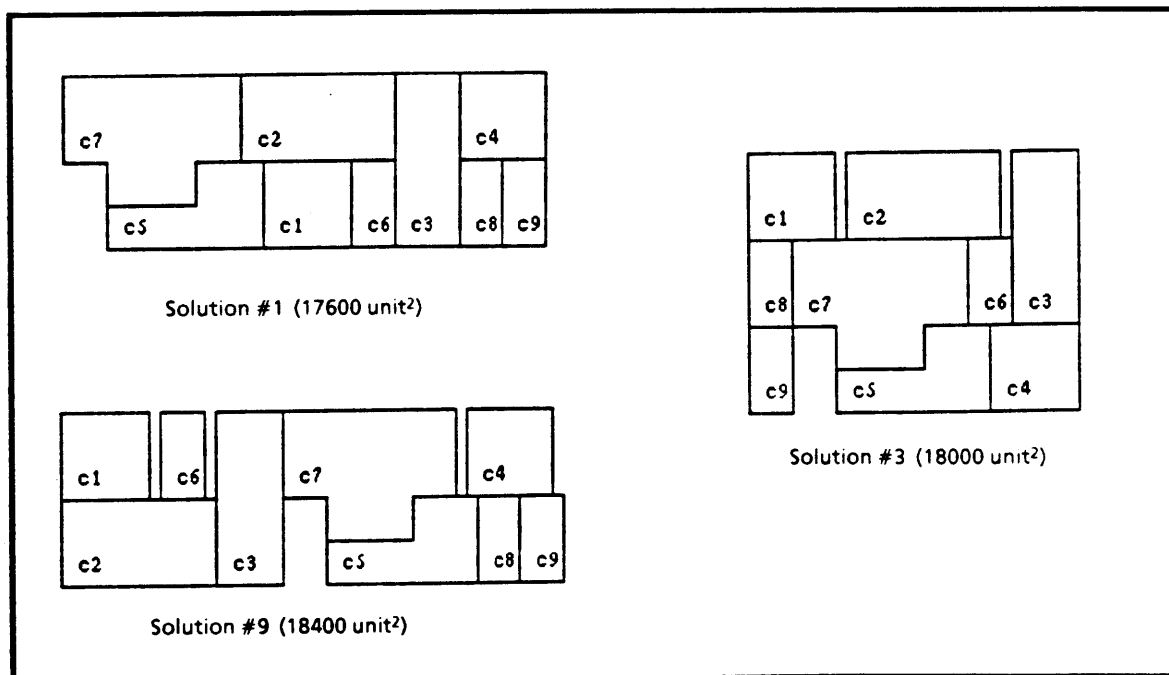
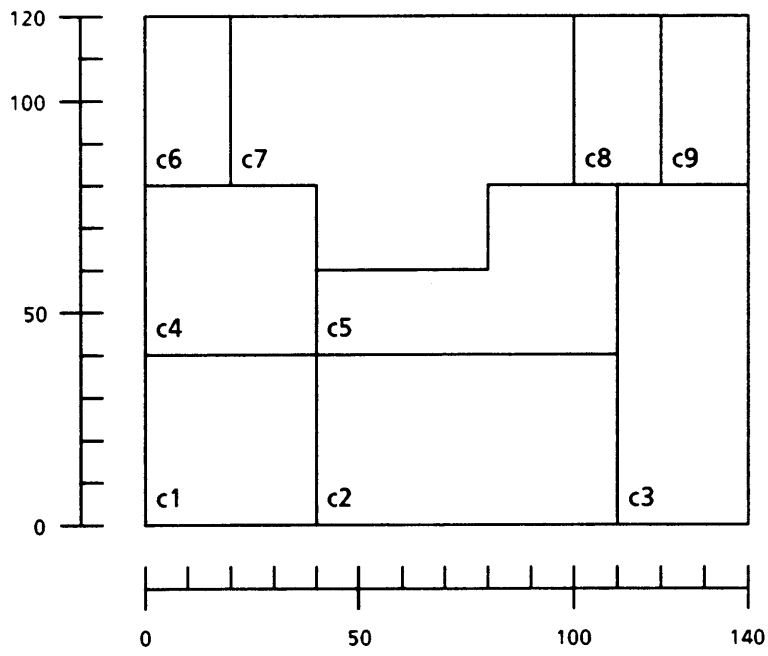


Figure A-1: Three solutions for Block-packing Example One



Total Area = 16800 unit<sup>2</sup>

**Figure A-2:** *Original solution to Block-packing Example One*

### A.5.2 Block-packing Example Two

- ① **Number of blocks** = 9.
- ② **Total Area** = 38000 unit<sup>2</sup>.
- ③ **Search Parameters:**
  - $l = 3$
  - $b = 9$
  - $n = 5$
- ④ **Time taken**  $\approx$  20 minutes. (Search ran to completion.)
- ⑤ **Pages:** 103 to 106.

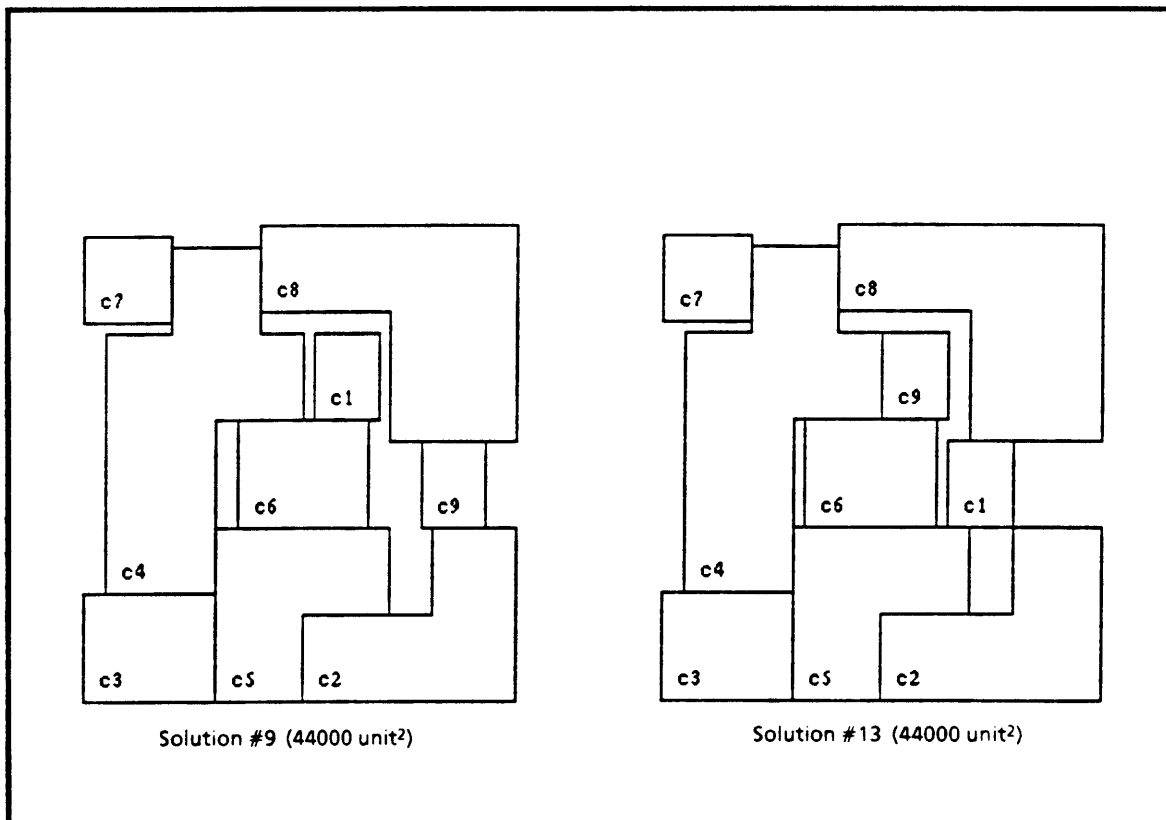
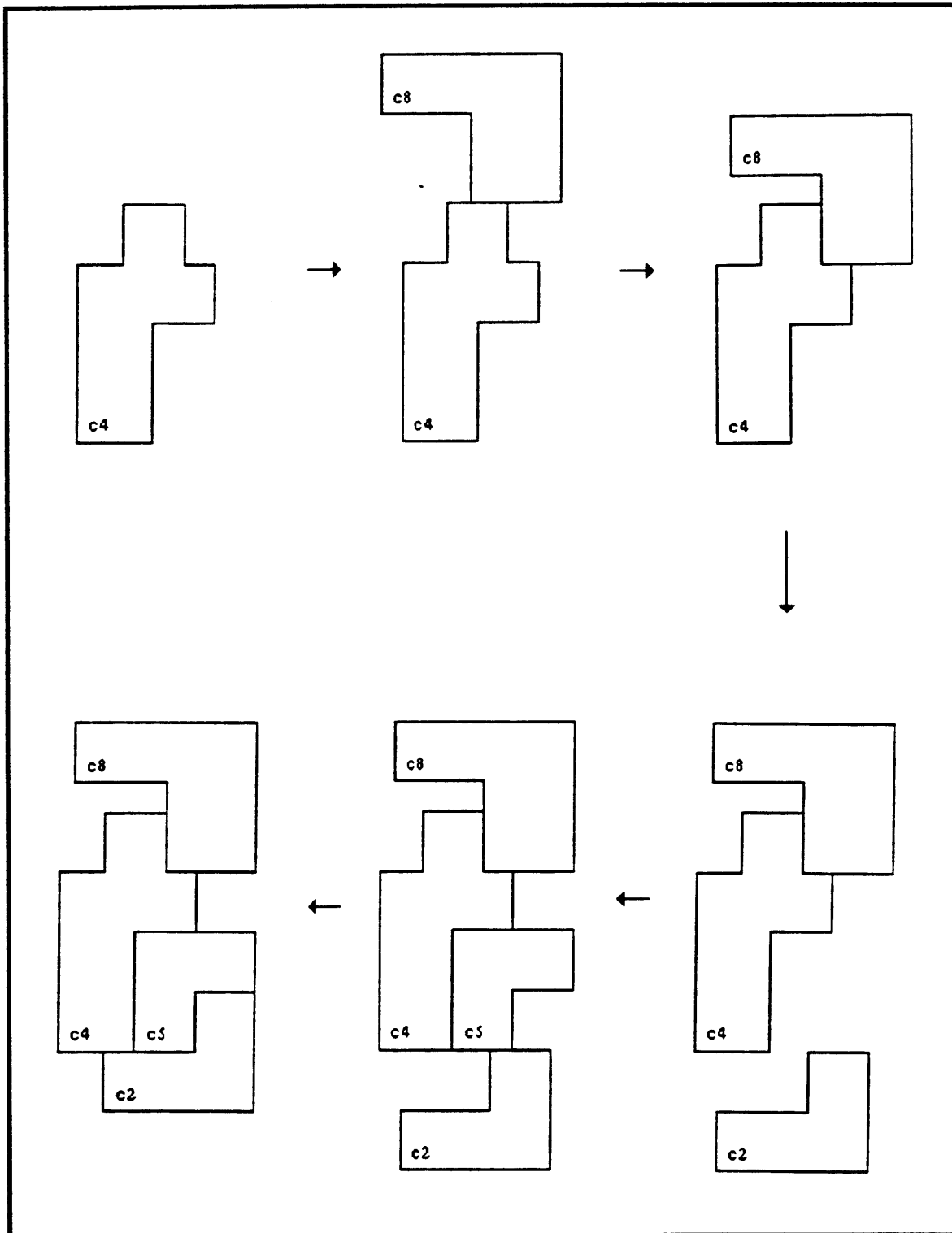
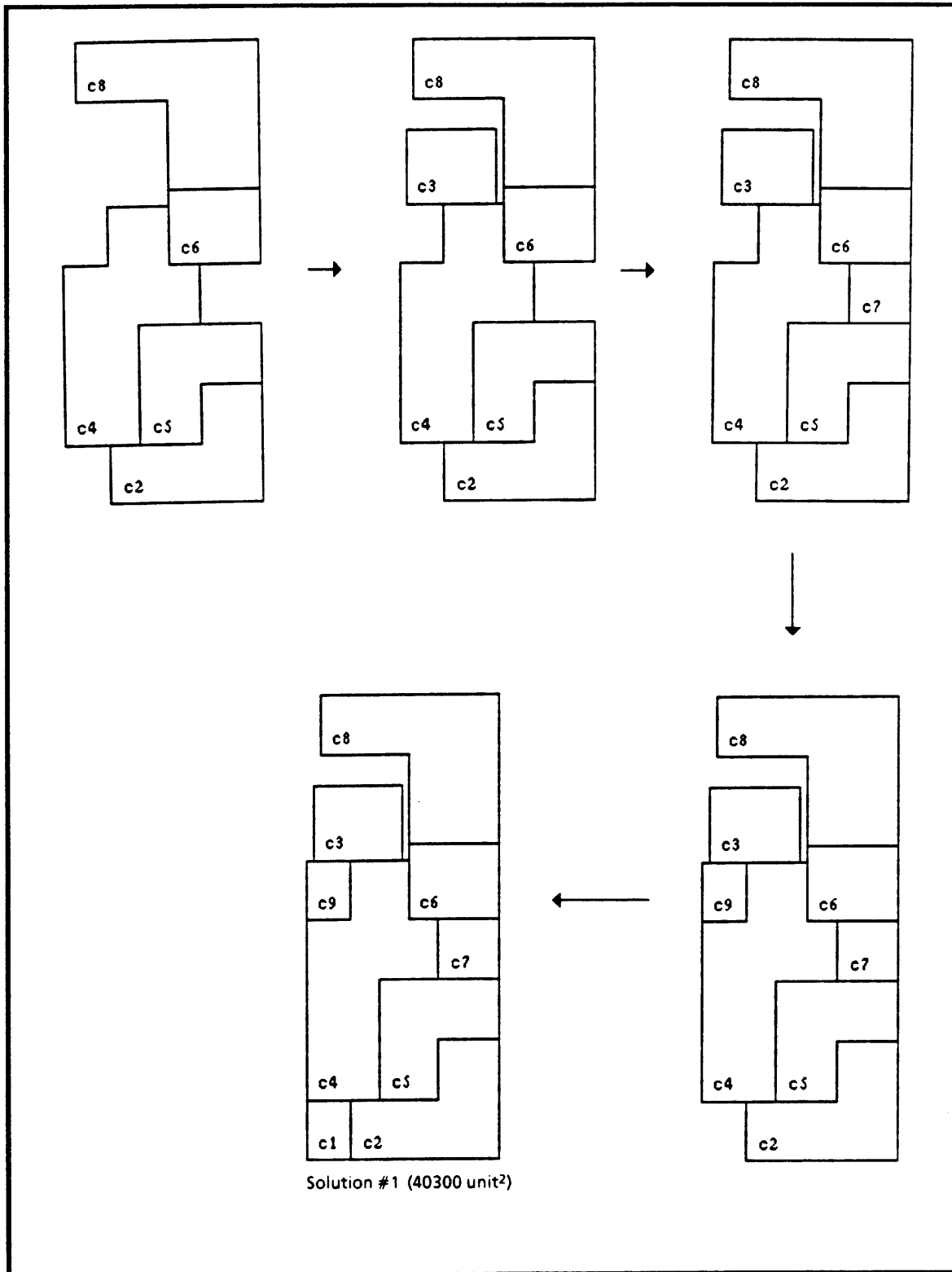


Figure A-3: Two solutions for Block-packing Example Two



**Figure A-4: Block-packing Example Two: Derivation of Solution #1 (part 1)**



**Figure A-5: Block-packing Example Two: Derivation of Solution #1 (part 2)**

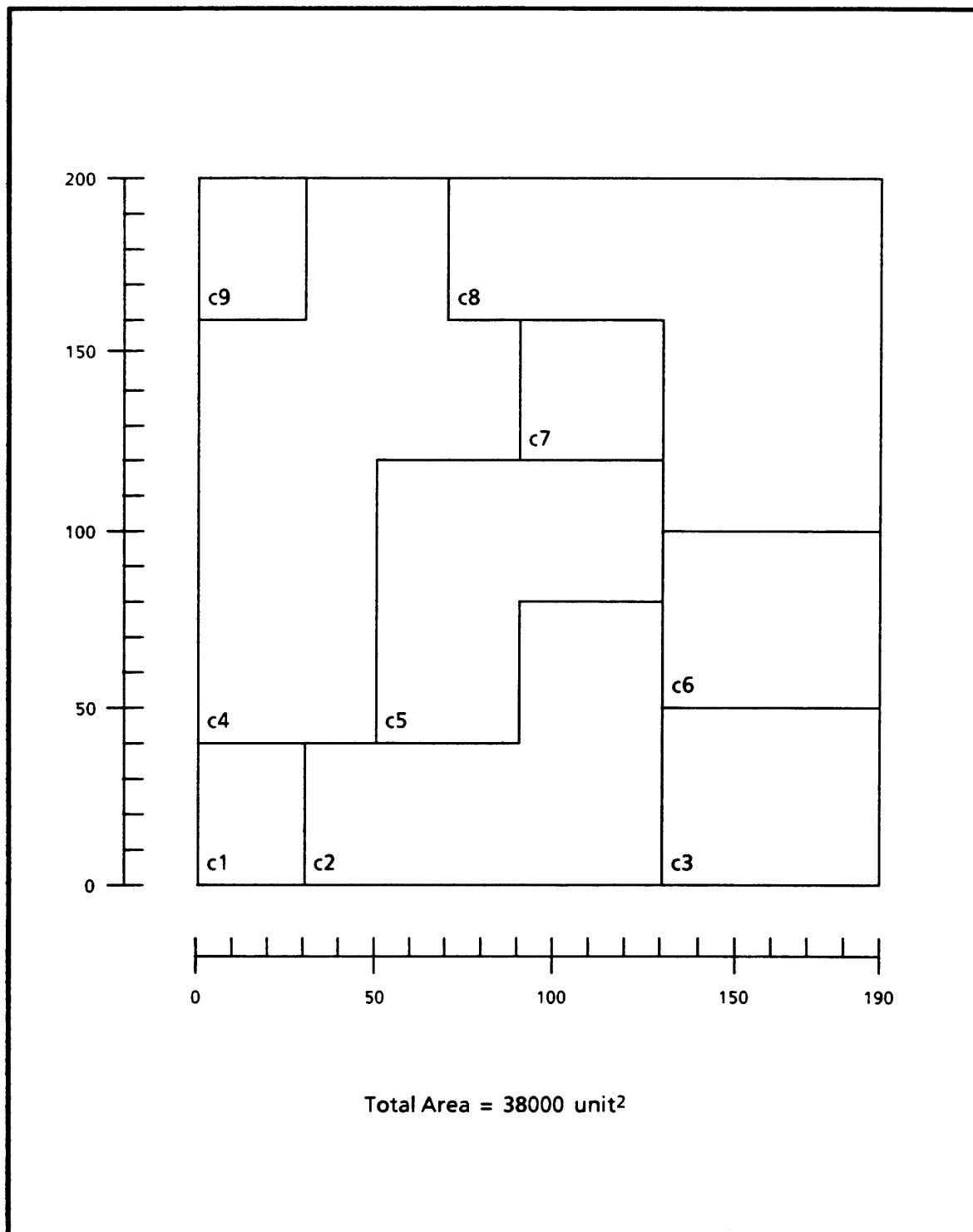


Figure A-6: Original solution to Block-packing Example Two



### A.5.3 Block-packing Example Three

① **Number of blocks** = 12.

② **Total Area** = 70400 unit<sup>2</sup>.

③ **Search Parameters:**

1st stage:  $\ell = 5$ ,  $b = 30$ , and  $n = 20$ .

2nd stage:  $\ell = 3$ ,  $b = 10$ , and  $n = 20$ .

3rd stage:  $\ell = 3$ ,  $b = 10$ , and  $n = 20$ .

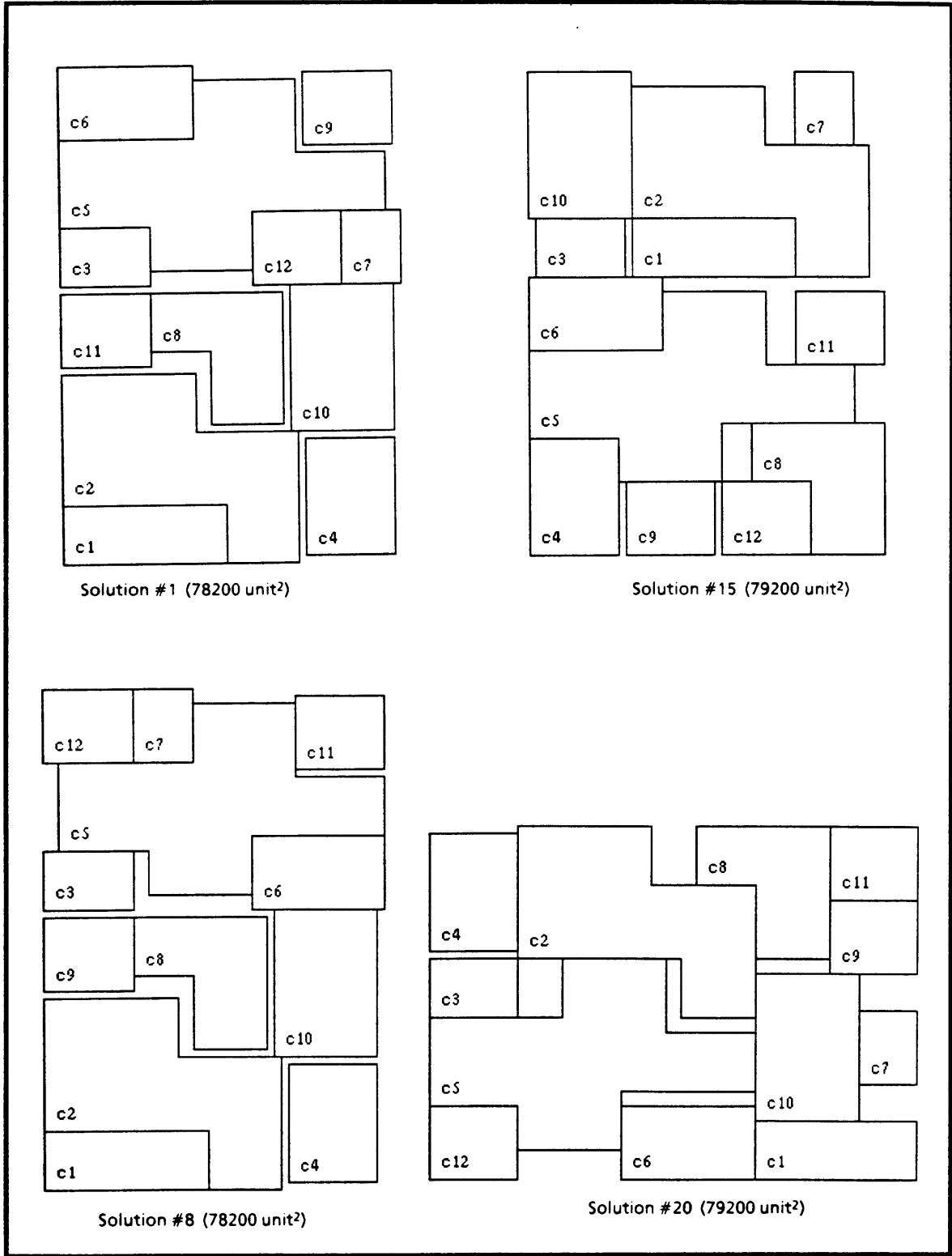
4th stage:  $\ell = 2$ ,  $b = 10$ , and  $n = 20$ .

④ **Time taken**  $\approx$  1 hour. (Search was terminated by the user.)

⑤ **Pages:** 108 to 123.

⑥ **Remarks:**

- This is an example of varying the search parameters at different stages of the search.
- Figures A-9 to A-21 step backward the derivation of solution#7.



**Figure A-7: Four solutions for Block-packing Example Three**

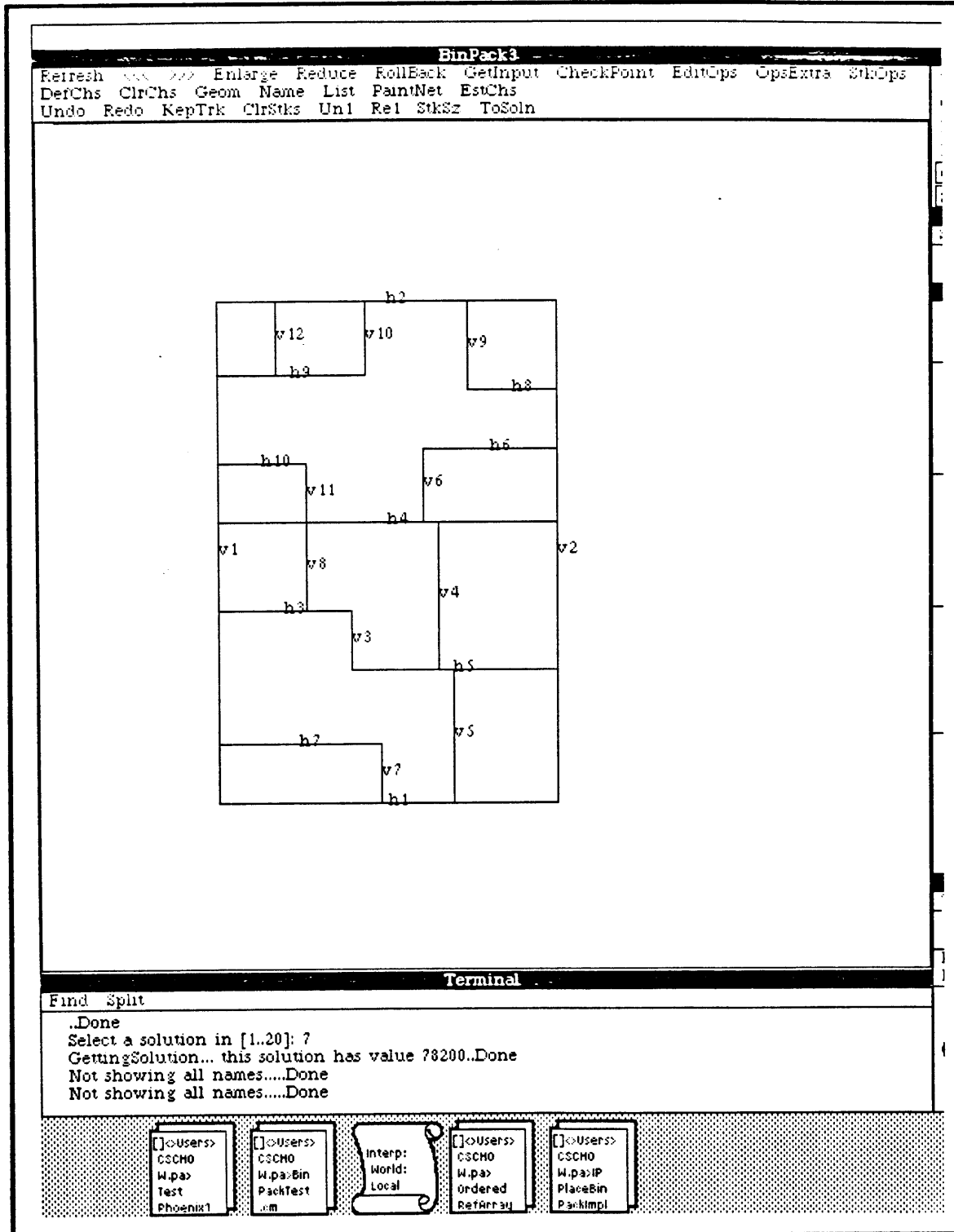


Figure A-8: Block-packing Example Three: Solution #7 (channels only)

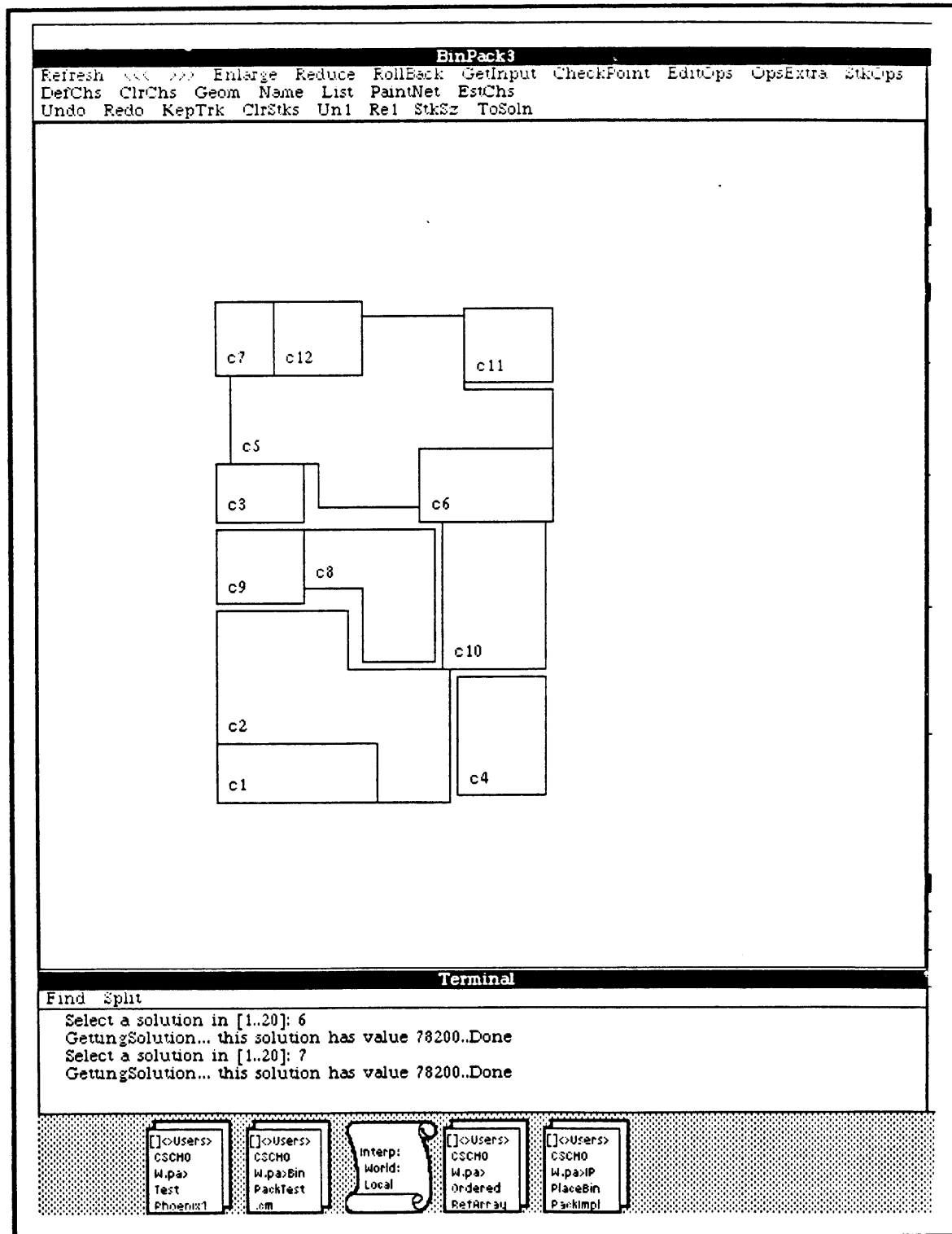


Figure A-9: Block-packing Example Three: Solution #7 (initial position)

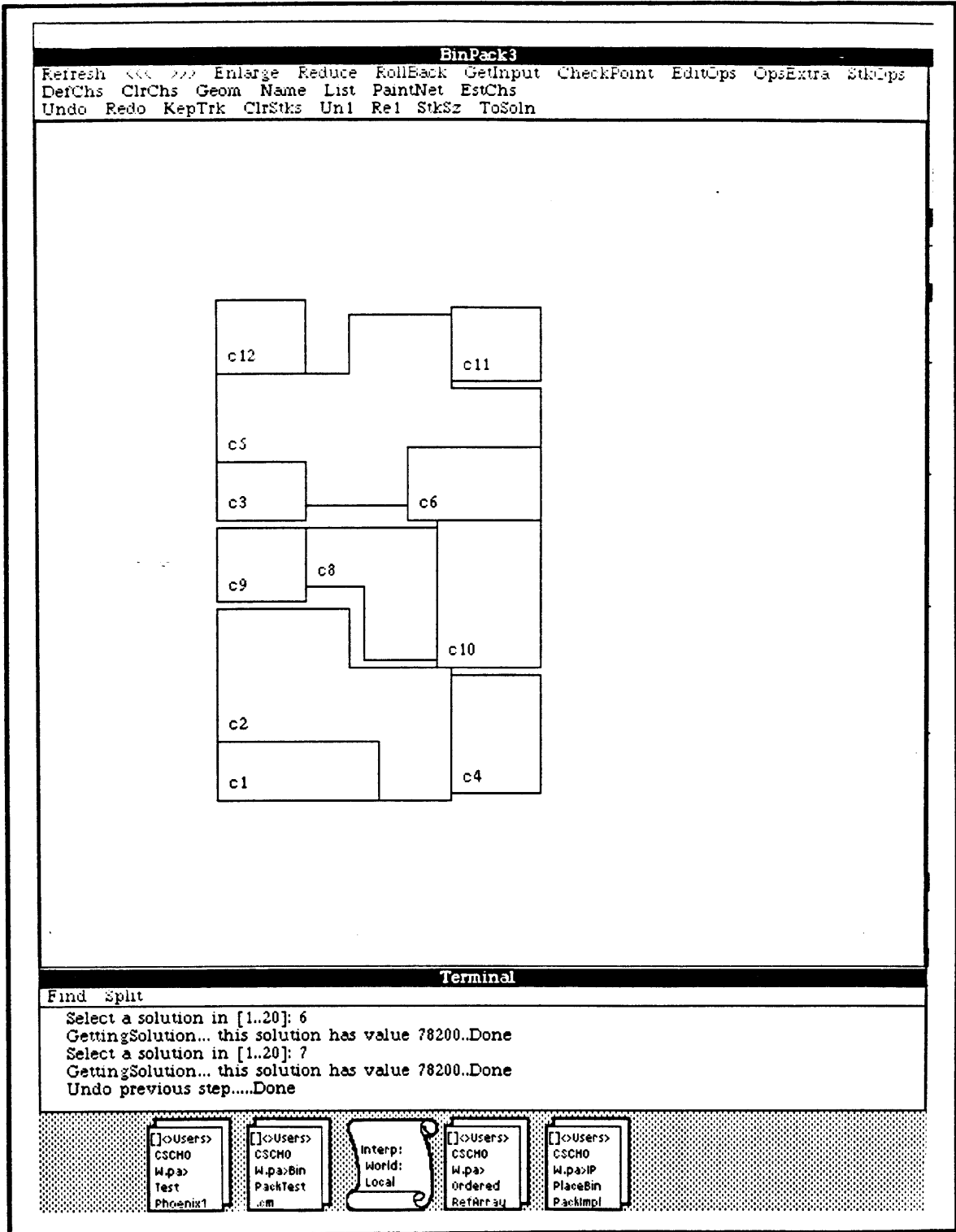


Figure A-10: Block-packing Example Three: Solution #7 (1 step back)

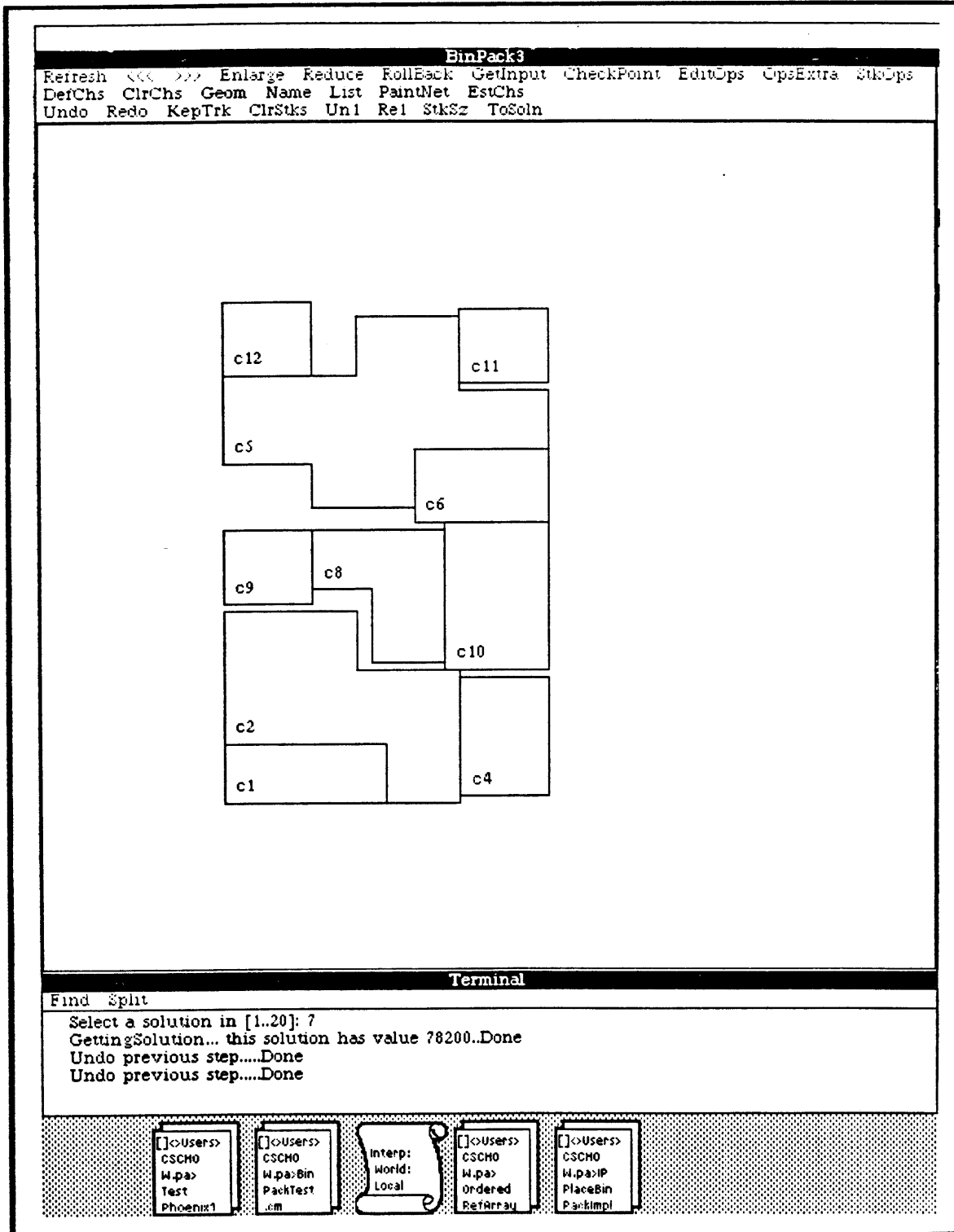


Figure A-11: Block-packing Example Three: Solution #7 (2 steps back)

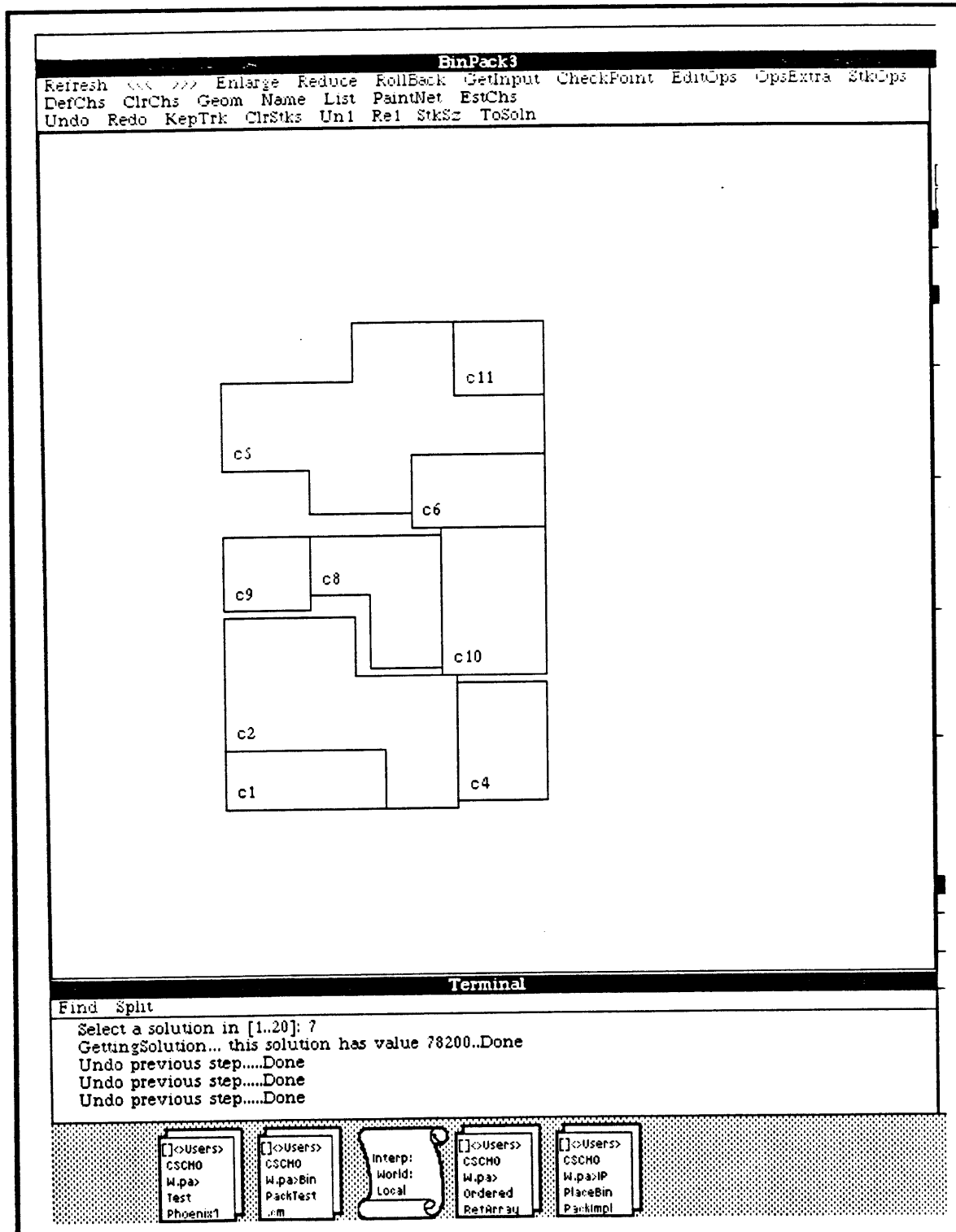


Figure A-12: Block-packing Example Three: Solution #7 (3 steps back)

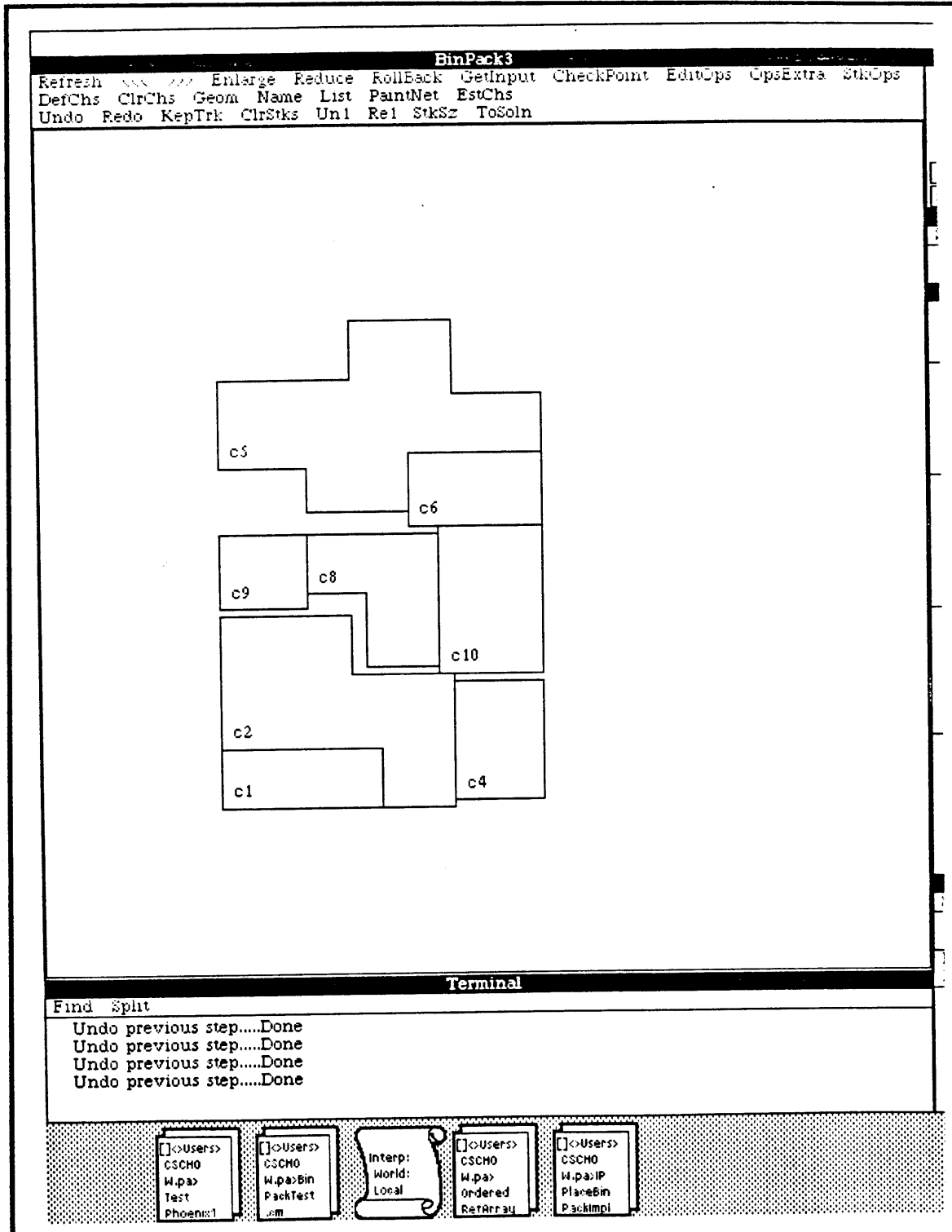


Figure A-13: Block-packing Example Three: Solution #7 (4 steps back)



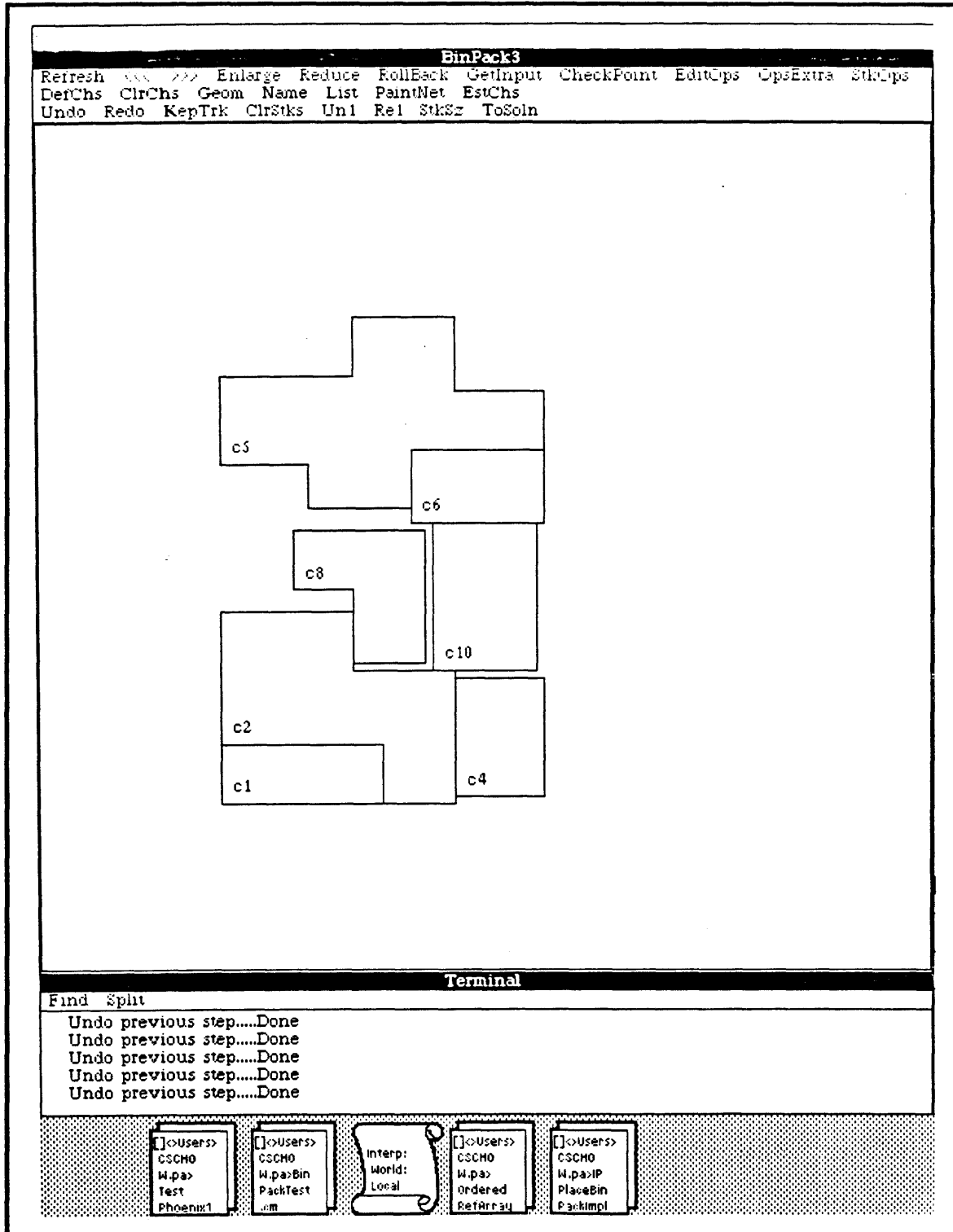


Figure A-14: Block-packing Example Three: Solution #7 (5 steps back)

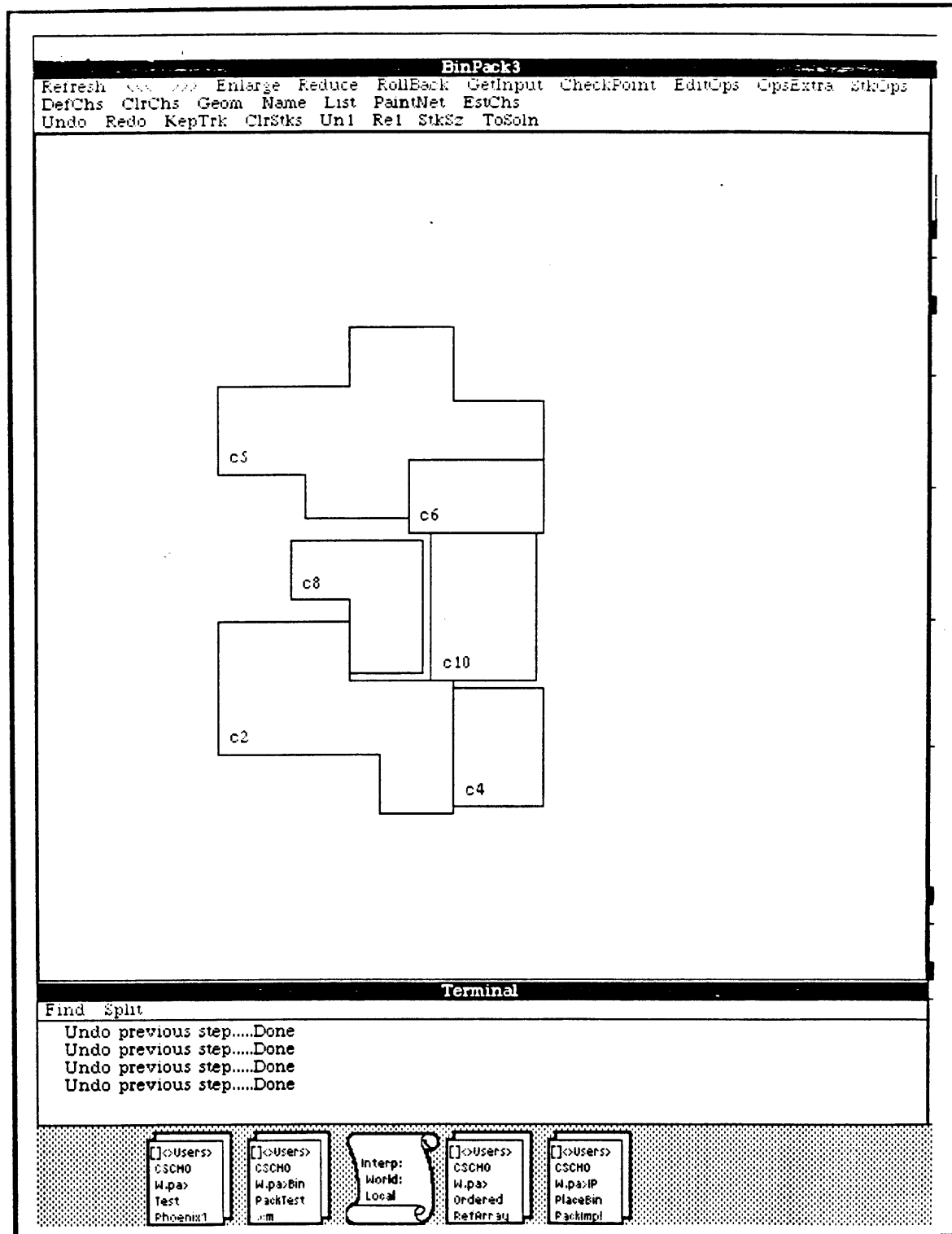


Figure A-15: Block-packing Example Three: Solution #7 (6 steps back)

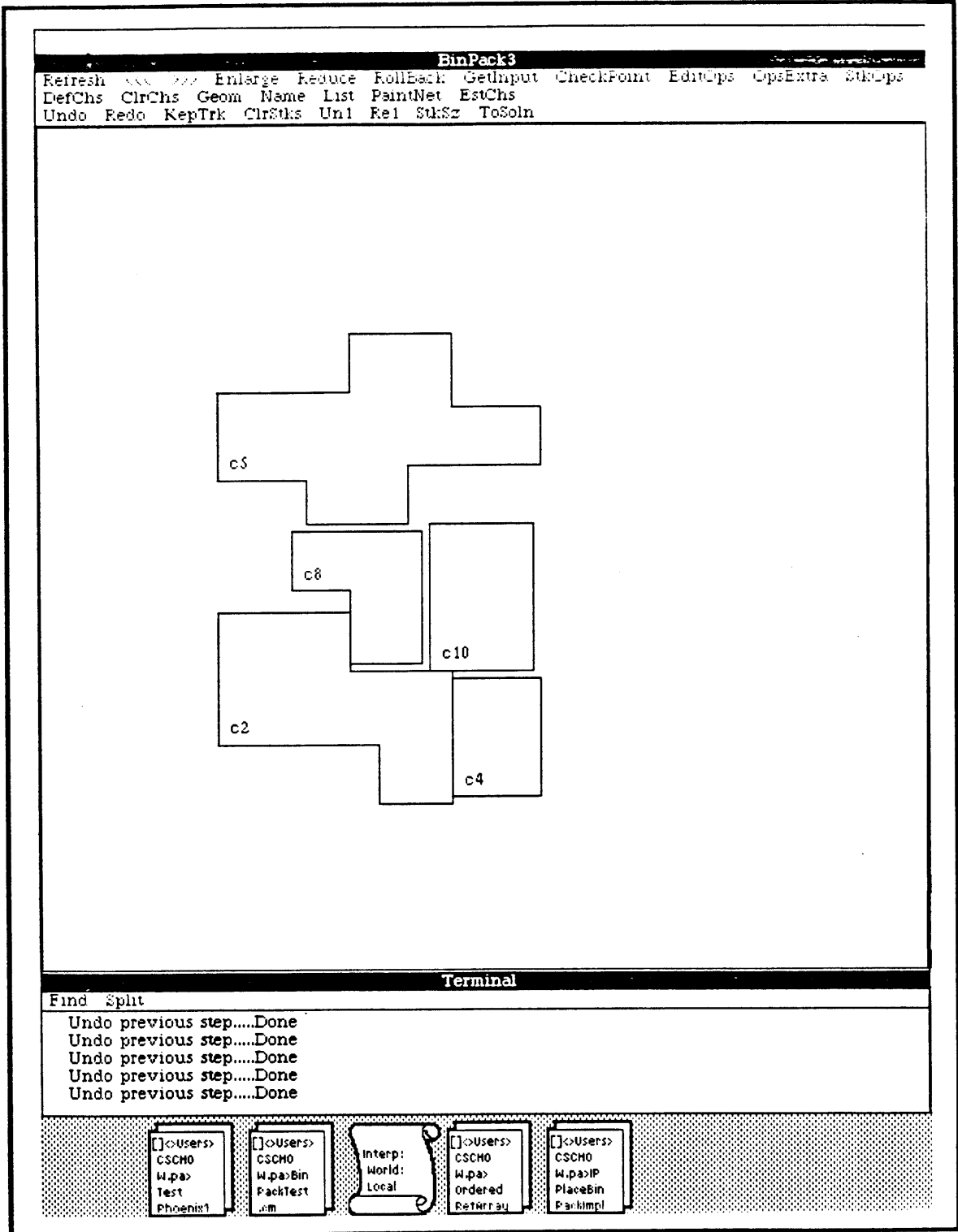


Figure A-16: Block-packing Example Three: Solution #7 (7 steps back)

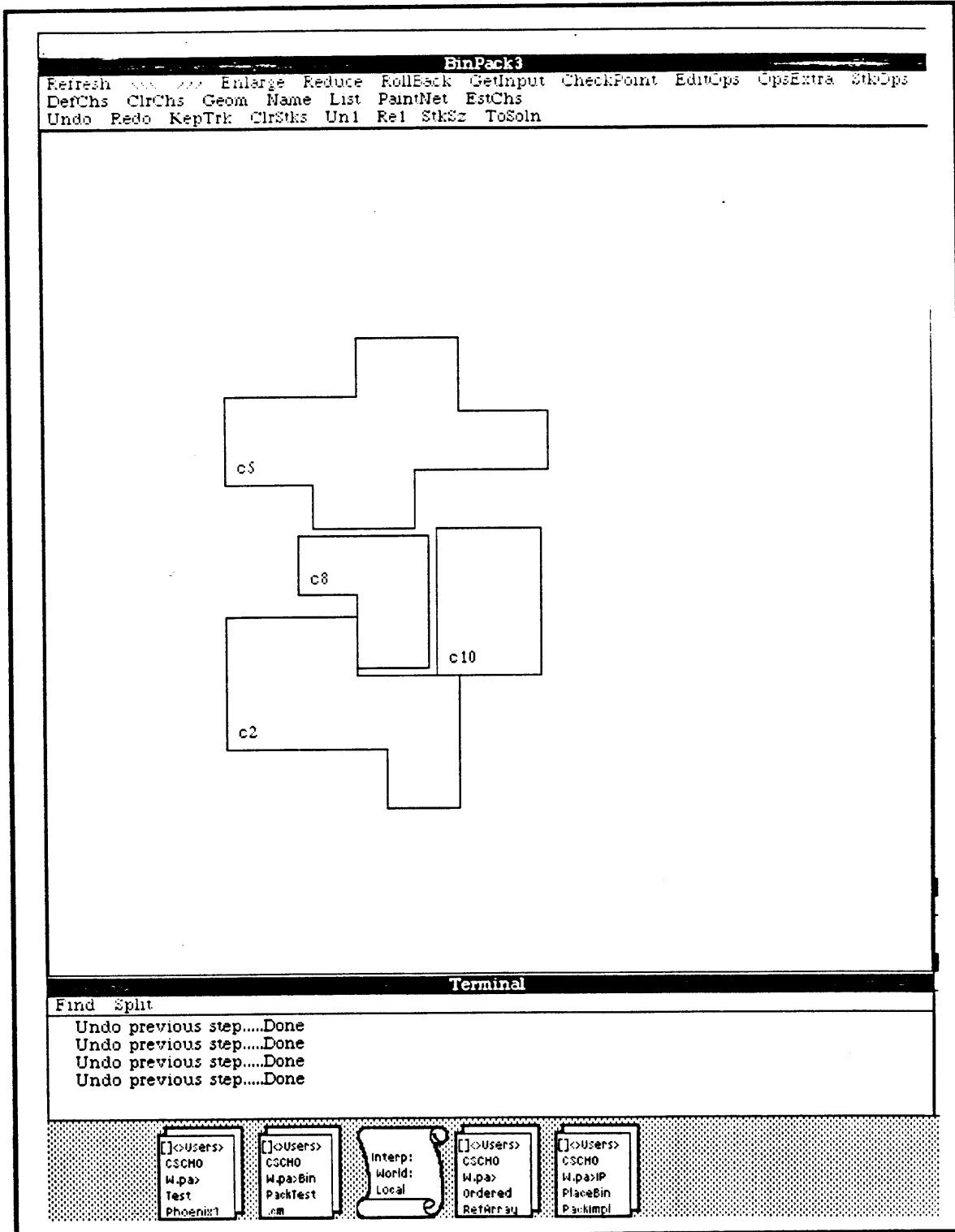


Figure A-17: Block-packing Example Three: Solution #7 (8 steps back)

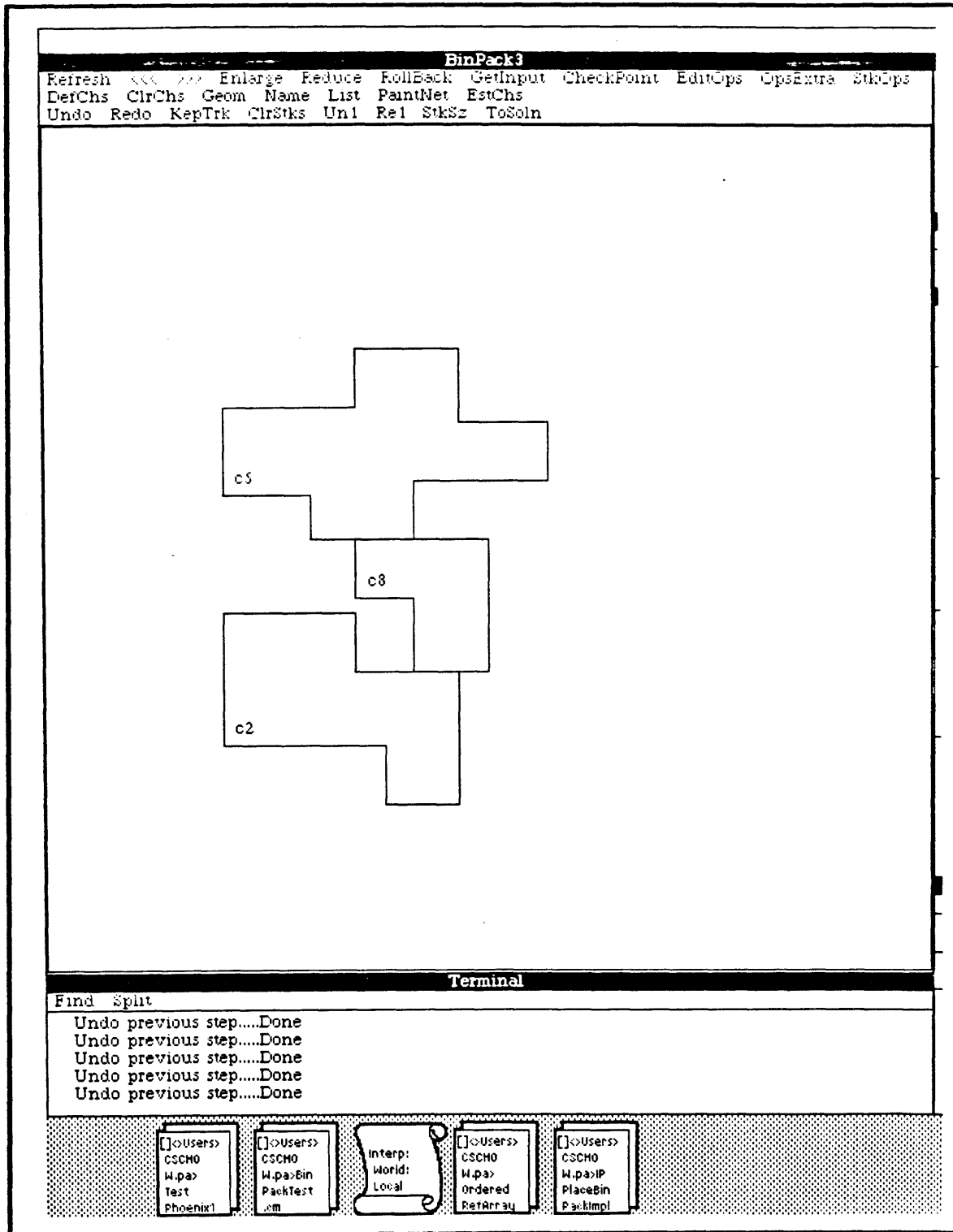


Figure A-18: Block-packing Example Three: Solution #7 (9 steps back)

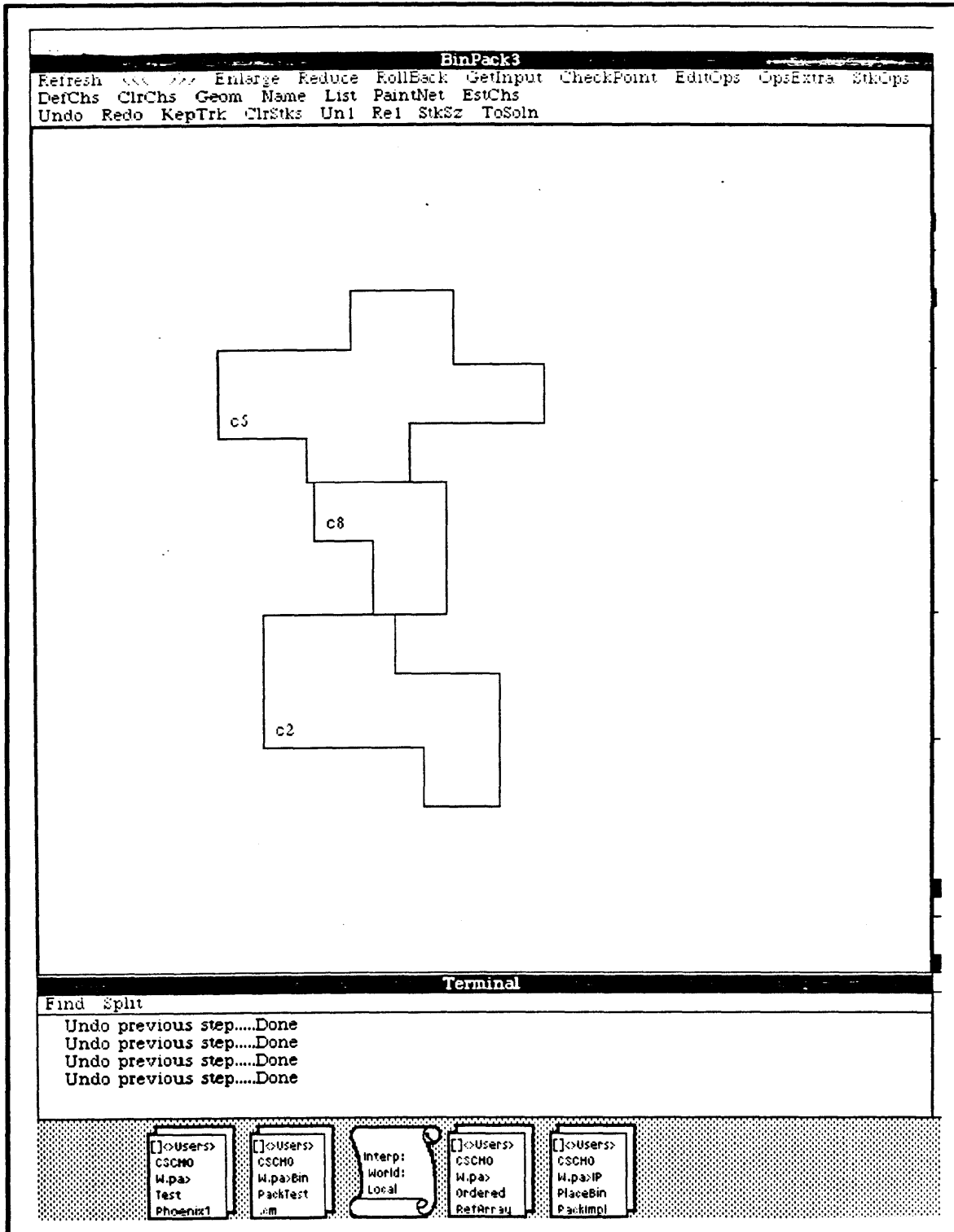


Figure A-19: Block-packing Example Three: Solution #7 (10 steps back)

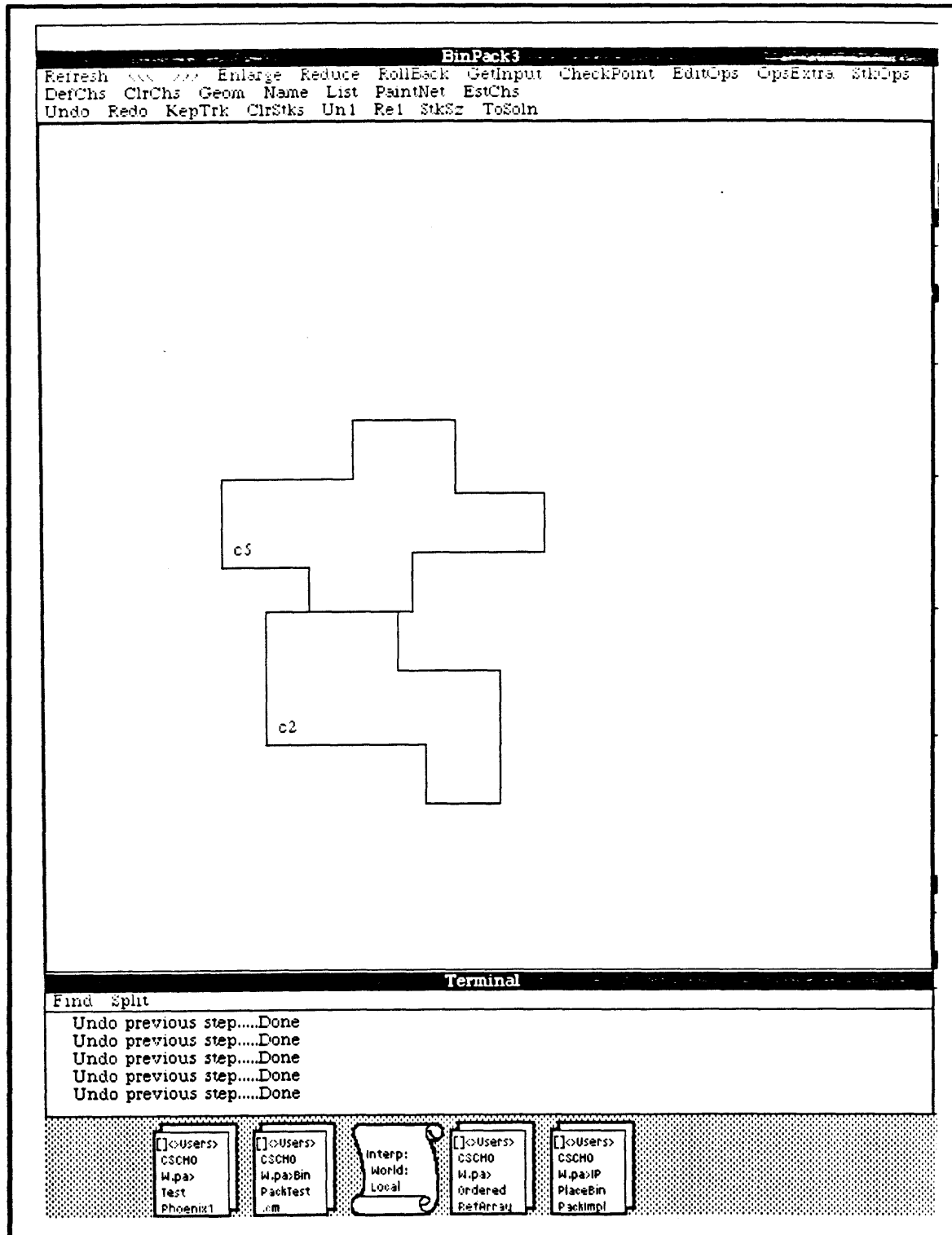


Figure A-20: Block-packing Example Three: Solution #7 (11 steps back)

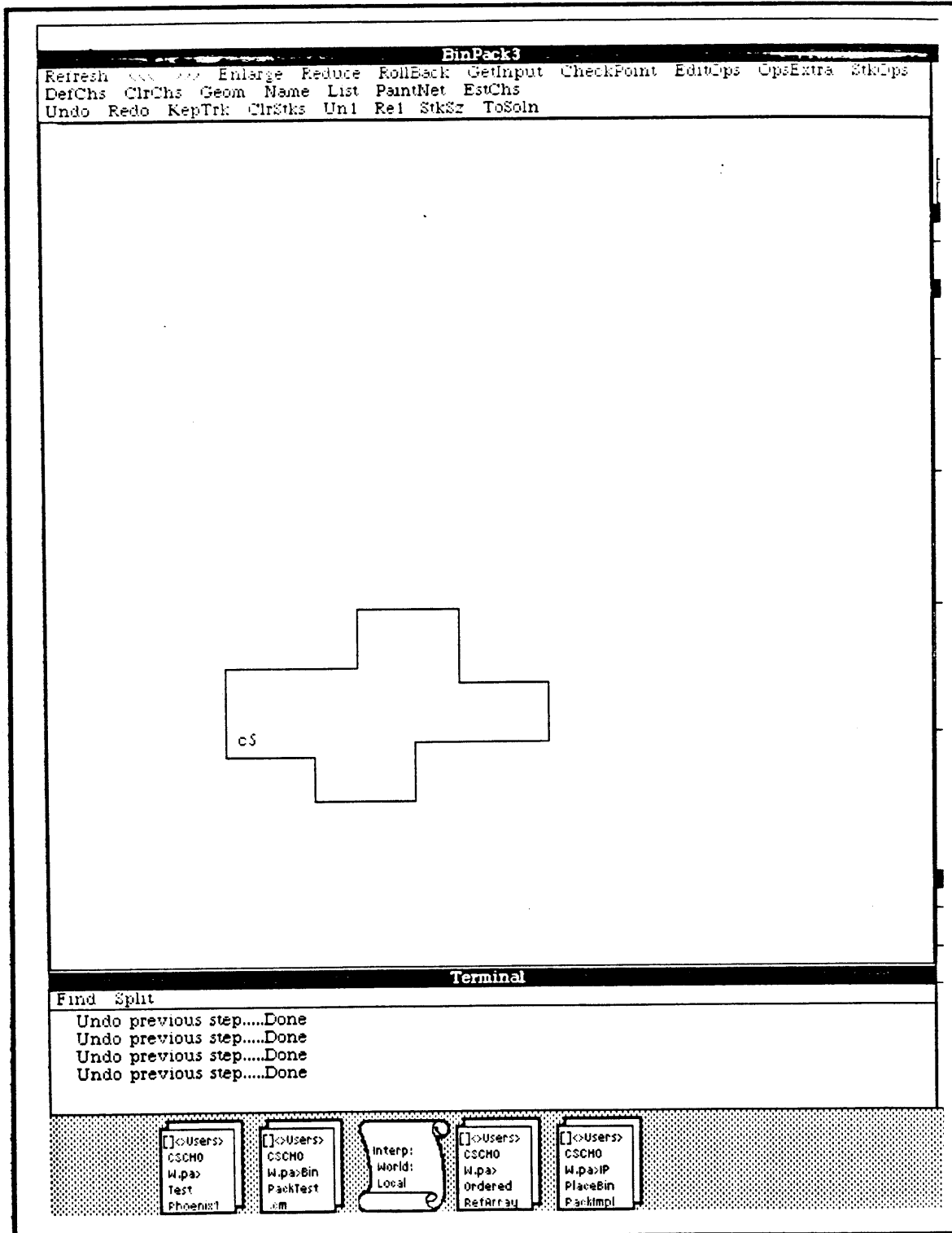


Figure A-21: Block-packing Example Three: Solution #7 (12 steps back)



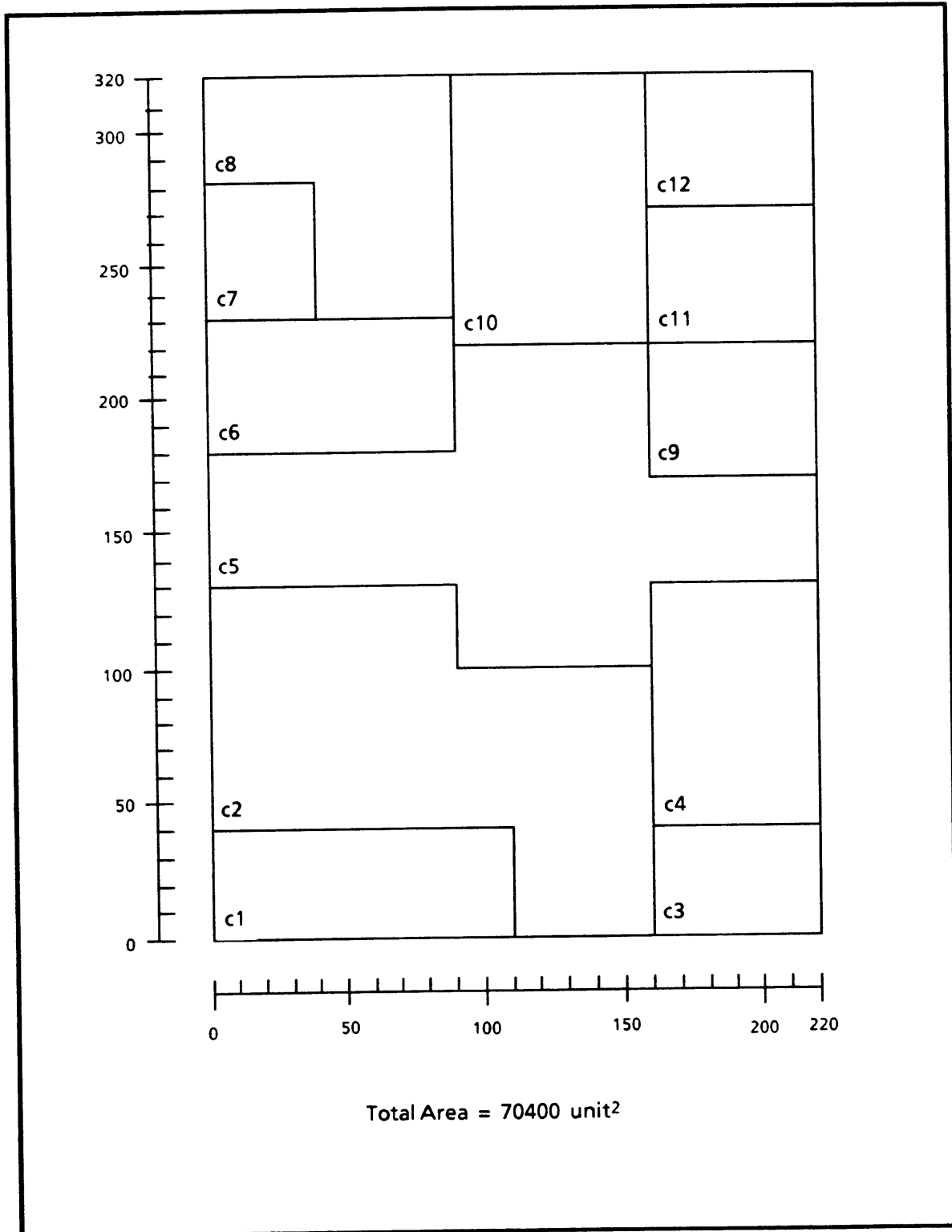


Figure A-23: Original solution to Block-packing Example Three

**This page is intentionally blank**

## **A.6 Floorplanning Results**

### **① Floorplanning Example One (8 blocks).**

Figure A-24 shows how the floorplan is generated.

### **② Floorplanning Example Two (92 blocks).**

Figure A-25 shows the floorplan. All the blocks are fake.

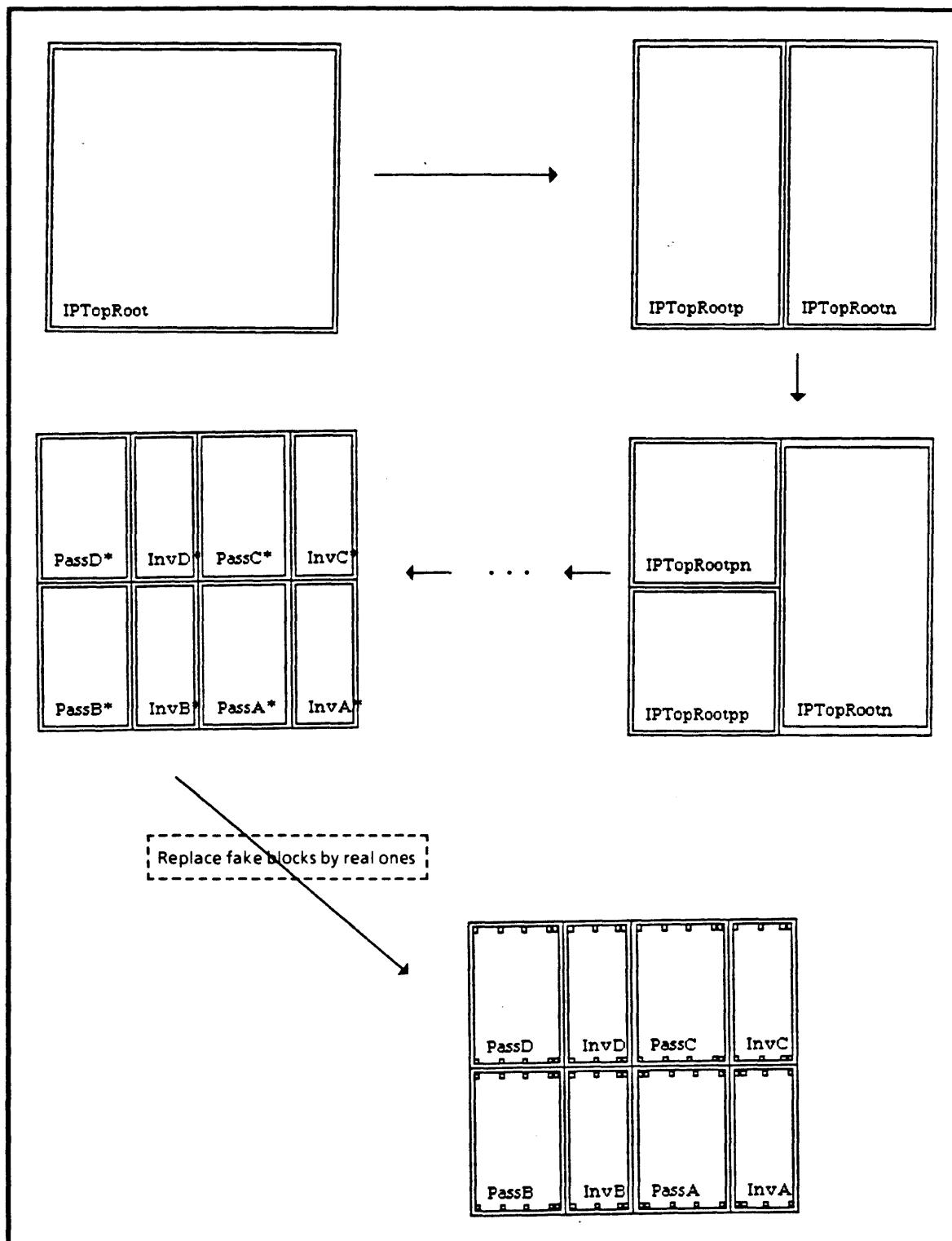


Figure A-24: Floorplanning Example One

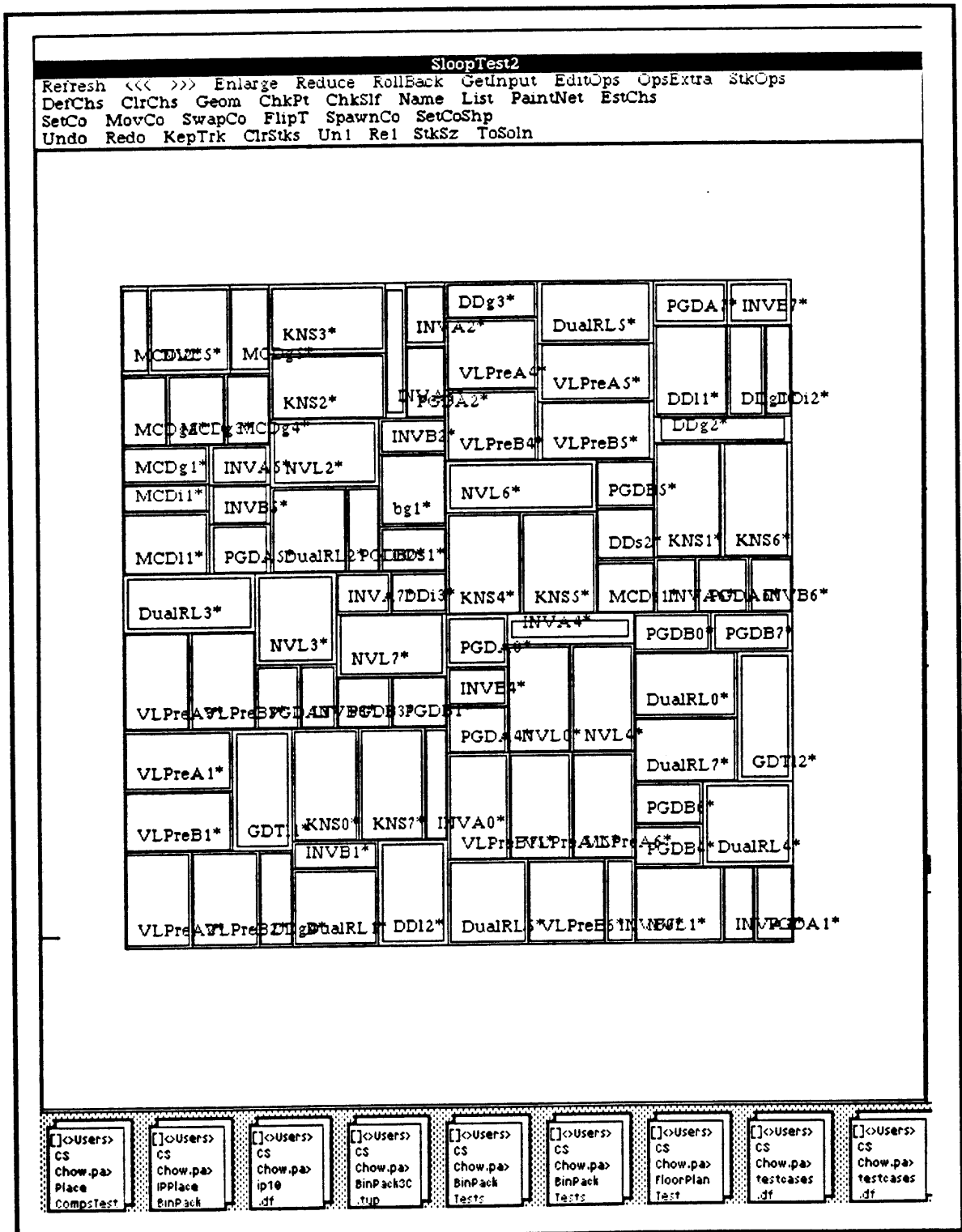


Figure A-25: Floorplanning Example Two

**This page is intentionally blank**

# Appendix B

## Design and Implementation of Phoenix

This appendix discusses implementation issues. It is intended to help interested readers duplicate the work.

### B.1 Background and Programming Environment

The design and implementation of the topological model and the topological operations took eight months: I spent  $2\frac{1}{2}$  months on the paper design,  $\frac{1}{2}$  month learning the programming environment, and five months on the implementation. During the five months, the implementation was revised at least ten times, with three major changes to the underlying data representation of the model. The resultant code for the system was about 250 textual pages.

Phoenix is written in Cedar [1], a highly interactive programming environment designed for building experimental systems, and runs on a Dorado [3], a single-user workstation having the processing power comparable to an IBM 370/168 processor.

### B.2 Interactive Interface Overview

The user interface of Phoenix is built from a number of existing graphics packages. The resultant interface is highly interactive and inputs are accepted from the users through the keyboard and the mouse. The interface has three parts:

- ① A message window
- ② A main display window
- ③ An input-output window

At the top of the screen (see Figure B-1), is the message window which displays error messages to the user. Beneath the message window is the main display window. It has a caption at the top and buttons below the caption. When the buttons are clicked (using the mouse) the corresponding operations are executed. The region beneath the buttons displays a view of the system. This view can be scrolled vertically or horizontally, magnified, and reduced.

The input-output window outputs messages from the system and accepts textual input from the user, it automatically keeps a log of the session.

Five figures are included to illustrate the user interface:

- ① Figure B-1 shows four placed blocks.
- ② Figure B-2 shows the four blocks after **Topologize**.
- ③ Figure B-3 shows the four blocks after **Geometrize**.
- ④ Figure B-4 shows the effect of reshaping the block c2 on the geometry.
- ⑤ Figure B-5 shows only the channels; the display of blocks have been suppressed.



Input Error. Please try again

**BinPack1**

Refresh <<< >>> Enlarge Reduce RollBack GetInput EditOps OpsExtra StkOps  
 DefChs ClrChs Geom ChkPt ChkSlf Name List PaintNet EstChs  
 SetCo MovCo SwapCo FlipT SpawnCo SetCoShp  
 BrkX FomX RemZ FomZ LtoT TtoL FlxKne ExtKne Grw Grw1 Shr Rot Mrr Ornt ClrCr

**Terminal**

```

Find Split
c3 0 0
Moving Component c3.....Done
DefiningChannels.....Done
Geometrizing.....Done
Destroying Channels.....Done
Move Component: <comp> <byX, byY: INT>
c2 30 40
Moving Component c2.....Done
Move Component: <comp> <byX, byY: INT>
c3 -40 50
Moving Component c3.....Done
SetComponent: <comp> <atX, atY: INT + 0> <active: BOOL + TRUE>
c4 10 -100
Setting Componet c4.....Done
  
```

[ ] <Users> cs Chow.pa> Ordered RefArrau	[ ] <Users> CS Chow.pa> ip10 df	[ ] <Users> CS Chow.pa> IPMain Viewerimpl	[ ] <Users> CS Chow.pa> IPMain Viewerimpl	[ ] <Users> CS Chow.pa> IPCoTab mesa
--	---	---	---	--

Figure B-1: View of four blocks

**BinPack1**

Refresh <<< >>> Enlarge Reduce RollBack GetInput EditOps OpsExtra StkOps  
 DefChs CtrChs Geom ChkPt ChkSif Name List PaintNet EstChs  
 SetCo MovCo SwapCo FlptT SpawnCo SetCoShp  
 BrkX FomX RemZ FomZ LtoT TtoL FlxKne ExtKne Grw Grw! Shr Rot Mrr Ornt CtrCr

The diagram shows a rectangular area divided into four blocks labeled c1, c2, c3, and c4. Block c3 is a tall vertical rectangle on the left. Block c2 is a horizontal rectangle on the right. Block c1 is a small square in the center. Block c4 is a small square at the bottom center. Dimensions h1, h2, and h3 are indicated at the top and bottom. Vertical positions v1, v2, v3, and v4 are marked on the left and right sides.

**Terminal**

```
Find Split
DefiningChannels.....Done
Geometrizing.....Done
Destroying Channels.....Done
Move Component: <comp> <byX, byY: INT>
c2 30 40
Moving Component c2.....Done
Move Component: <comp> <byX, byY: INT>
c3 -40 50
Moving Component c3.....Done
SetComponent: <comp> <atX, atY: INT + 0> <active: BOOL + TRUE>
c4 10 -100
Setting Componet c4.....Done
DefiningChannels.....Done
```

<pre style="margin: 0;">[&gt;Users&gt; Cs Chow.pa&gt; Ordered RefArray</pre>	<pre style="margin: 0;">[&gt;Users&gt; cs Chow.pa&gt; ip1@ .df</pre>	<pre style="margin: 0;">[&gt;Users&gt; CS Chow.pa&gt; IPMain Viewerimpl</pre>	<pre style="margin: 0;">[&gt;Users&gt; Cs Chow.pa&gt; IPMain Viewerimpl</pre>	<pre style="margin: 0;">[&gt;Users&gt; cs chow.pa&gt; IPCOTab .mesa</pre>
--	--	---	---	---

Figure B-2: View of the four blocks after Topologize

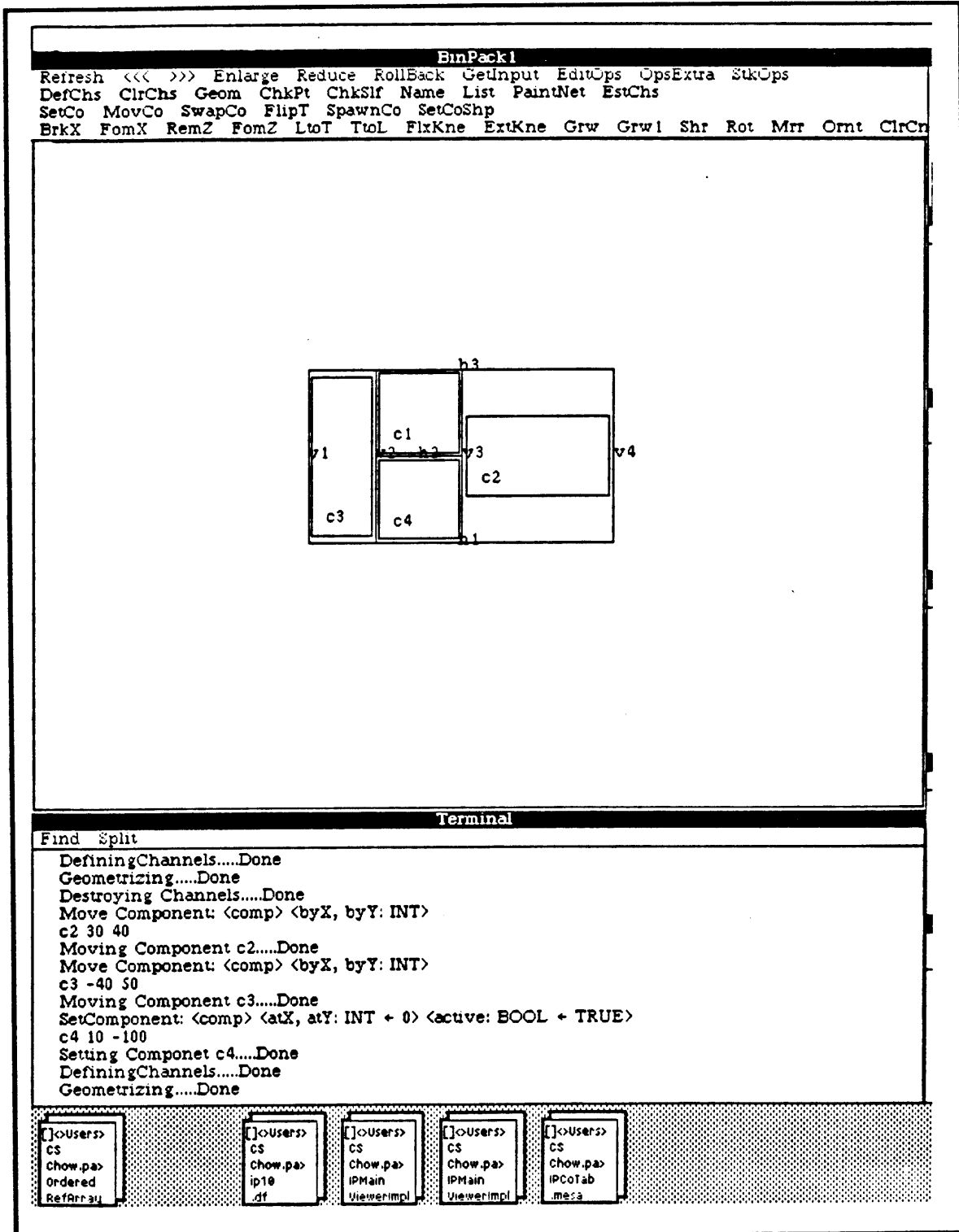


Figure B-3: View of the four blocks after Geometrize

SetCoShp aborted

BinPack1

Refresh <<< >>> Enlarge Reduce RollBack GetInput EditOps OpsExtra StkOps  
 DefChs ClrChs Geom ChkPt ChkSif Name List PaintNet EstChs  
 SetCo MovCo SwapCo FlptT SpawnCo SetCoShp  
 BrkX FomX RemZ FomZ LtoT TtoL FlxKne ExtKne Grw Grw1 Shr Rot Mrr Ornt ClrCr

Terminal

Find Split  
 c2 80 30  
 Input Error...Aborted  
 Set Component Shape: <comp> <shape: {(x y) sw: (x4 y4)}>  
 c2 {80 30}  
 Set Component Shape: <comp> <shape: {(x y) sw: (x4 y4)}>  
 c2 {(80 40)}  
 Setting c2 shape.....Done  
 Set Component Shape: <comp> <shape: {(x y) sw: (x4 y4)}>  
 c2 {(40 80)}  
 Setting c2 shape.....Done

[ ]<Users>  
 cs  
 Chow.pa>  
 Ordered  
 RefArray

[ ]<Users>  
 cs  
 Chow.pa>  
 ip1@

[ ]<Users>  
 cs  
 Chow.pa>  
 IPMain  
 Viewerimpl

[ ]<Users>  
 cs  
 Chow.pa>  
 IPMain  
 Viewerimpl

[ ]<Users>  
 cs  
 Chow.pa>  
 IPCoTab  
 mesa

Figure B-4: View of the four blocks after reshaping the block c2

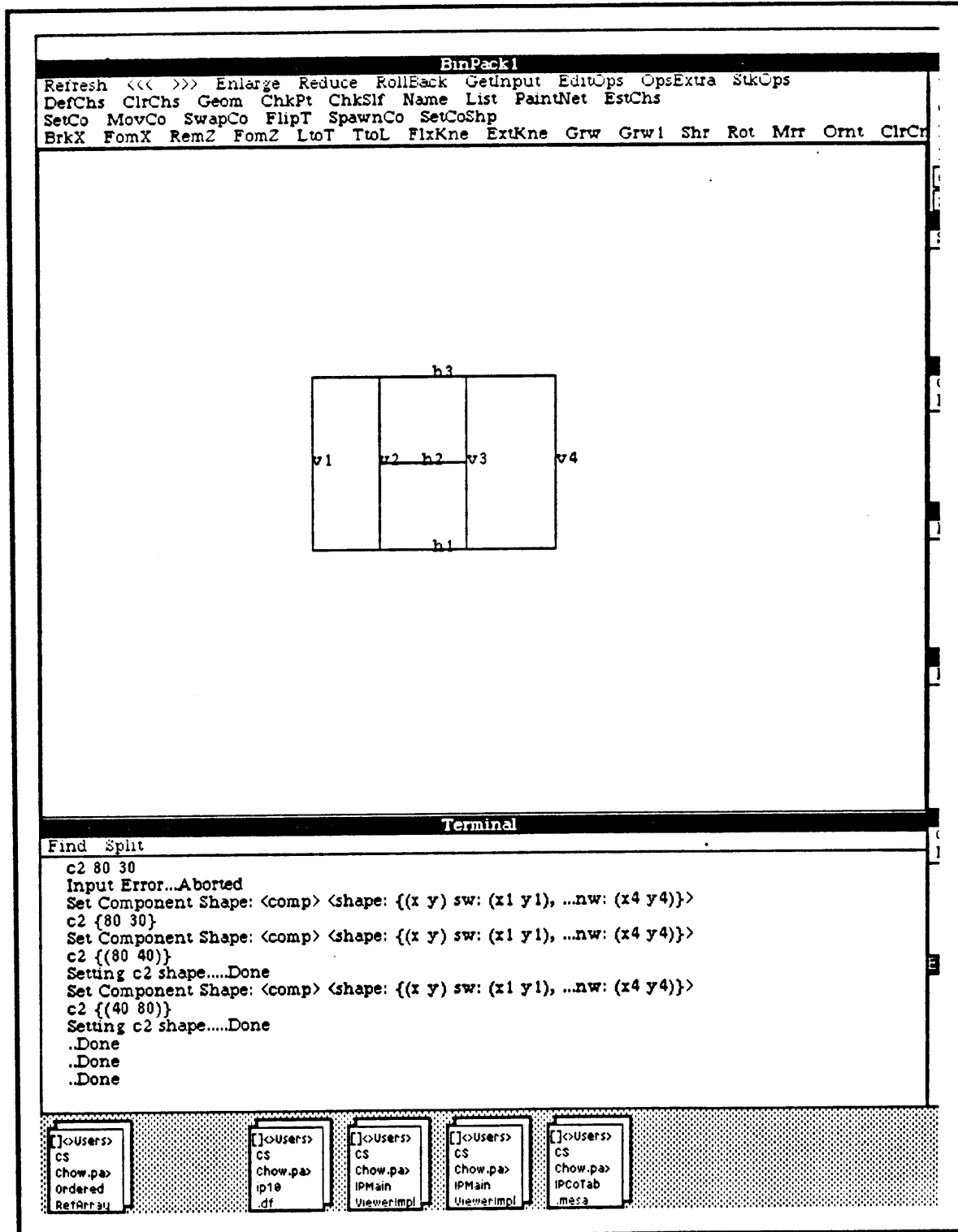


Figure B-5: View of only the channels

## B.3 Implementation Discussions

### B.3.1 Design Principles

Besides using a highly modular design methodology, I found the following principles useful in choosing a proper data structure for the system:

① **Avoid Redundancy.**

Keep in the system the minimal number of structures needed to achieve a certain efficiency. This is because the more information that is kept in the system, the harder it is to update the system, increasing the difficulty of implementing the topological operations.

② **Exploit the symmetry of the problem.**

An example is to use the same representation for both horizontal and vertical channels and differentiating the two kinds of channels by a label. *Negative* and *positive* are used, instead of left or bottom and right or top, to reference the two channel ends. This allows horizontal and vertical channels to be treated uniformly.

Another use of symmetry is as follows. To identify the four principal directions, *south*, *east*, *north*, and *west* are used; and to identify the four corner directions, *southwest*, *southeast*, *northeast*, and *northwest* are used. These values are used consistently throughout the system, such as specifying a topological operation and referencing the corner of a block. Moreover, symmetry operators (such as rotation and reflection) and transformation operators (for resolving a corner direction into two principal directions or converting a principal direction into *negative* or *positive*) are defined for the values. This technique reduces a lot of complexities in implementing the topological operations and enables a more concise implementation.

③ **Use cyclic pointer pairs to piece parts together.**

A large piece of structure can be partitioned into sub-parts without compromising efficiency by linking the sub-parts with cyclic pointer pairs, if necessary.

### **B.3.2 Data Structure**

One of the most difficult tasks in implementing Phoenix was in building the proper data structure to represent the interdependency between the channels and the blocks. This section describes the data structures used to represent the interdependency.

#### **B.3.2.1 Channel**

The boundary information (intersections and surrounding blocks) of a channel is represented as follows:

- ① For each end of the channel, a pointer to the bounding intersection is kept.
- ② For each side of the channel, a list of pointers to the side intersections interspersed with pointers to the side blocks is kept. To access the list quickly, pointers to the beginning and to the end of the list are kept. Alternatively, a doubly linked list can be used instead of a linear list.

#### **B.3.2.2 Intersection**

The intersection of a channel has the following:

- ① A label to indicate the kind of intersection.
- ② A pointer to its owner: the channel.
- ③ A pointer to its dual: the corresponding intersection in the intersecting channel.

For every intersection of a channel the following are true:

- The owner of its dual is the pointer to the intersection channel.
- The dual of its dual is the pointer to itself.

#### **B.3.2.3 Block**

A block is represented by its bounding rectangle minus the rectangles corresponding to the missing corners. For each edge of the block a pointer to the corresponding channel is kept.

#### **B.3.2.4 Data Structure Summary**

Choosing the right data structure is difficult and requires experimentation before a suitable representation can be found. So it is important to keep the implementation modular to limit the effect of changes. Design principles in Section B.3.1 can be used to reduce the complexity of the task.

#### **B.3.3 Implementing Tracking and Backtracking**

The tracking and backtracking in Phoenix is implemented by a redo stack and an undo stack. Both stacks always contain the same number of frames. Each stack frame contains a pointer to a record that contains the arguments of a topological operation and a tag identifying the operations. The undo (or redo) is effected by a small interpreter that discriminates the tag in the record and applies the operation, identified by the tag, to the arguments in the record.

#### **B.3.4 Implementing First Class Operations**

First class operations is implemented using three levels of abstraction:

① **Bottom-level operations.**

These operations interface to the underlying data structures like pointers, channels and intersections. Some examples of operations in this level are operations to create and destroy channels, operations to create and destroy intersections, and operations to manipulate intersections.

② **Intermediate-level operations.**

The operations in this level actually do most of the work in the top-level operations. At this level, the functional inverse of an operation is also its actual inverse. Operations at this level are not suitable for use by a client program because the interface to these operations requires a lot of extra information not needed to specify the topological operations. An example is the operation that implements **BreakCross**, this operation



does not automatically create a new channel but requires a new channel as part of the input. The new channel is created by another intermediate-level operation.

③ Top-level operations.

These are the first class operations. Operations in this level can be thought of as being encapsulations of operations at the intermediate level to provide a more convenient interface for the client. Besides the specific arguments needed, all these operations have the following in their function specifications:

- A boolean argument indicating whether the input arguments should be verified before executing the operation. The default value is true but is set to false when the operation is used to construct another composite operation that involves some invalid intermediate topologies.
- A boolean argument indicating whether the redo and the undo stacks should be pushed. Again the default is true but is set to false when the operation is used to construct a composite operation.
- Returns a pointer to the redo argument record and a pointer to the undo argument record. These are the pointers pushed onto the redo and undo stacks respectively. A composite operation is represented by a list of pointers.

### **B.3.5 Debugging Aids**

Besides the interactive graphical interface, the two debugging aids below were extremely helpful:

① An operation that verifies the consistency in the internal state of the system.

This operation is useful for detecting and localizing system bugs that are not visible immediately.

② An executable debugging log.

Debugging sessions are recorded and added to the debugging log file. After each revision of the system, the log file is executed to check if new bugs are introduced by the revision.

### **B.3.6 Implementation Summary**

Major changes are inevitable in building a system like Phoenix. Design principles are used to manage implementation complexity. These principles can be summarized as follows:

- ① Choose the simplest solution.
- ② Exploit the symmetry in the problem.
- ③ Divide and conquer.

## References

- [1] Deutsch, L. P. and E. A. Taft, "Requirements for an experimental programming environment," *Technical Report CSL-80-10*, Xerox Palo Alto Research Center, 1980.
- [2] Kirkpatrick, S.; C. D. Gelatt Jr., and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, No. 4598, May 1983.
- [3] Lampson, B. W. and K. A. Pier, "A processor for a high performance personal computer," *Proceedings of the 7th IEEE Symposium on Computer Architecture*, May 1980.
- [4] Lauther, U., "A Min-cut Placement Algorithm for General Cell Assemblies Based on a Graph Representation," *Proceedings of the 16th Design Automation Conference*, San Diego, California, June 1979.
- [5] Papadimitriou, C. H. and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, 1982.
- [6] Preas, B. T. and W. M. vanCleave, "Placement Algorithms for Arbitrarily Shaped Blocks," *Proceedings of the 16th Design Automation Conference*, San Diego, California, June 1979.
- [7] Preas, B. T., *Placement and Routing Algorithms for Hierarchical Integrated Circuits Layout*, Stanford University, Department of Electrical Engineering, Ph.D. Dissertation, 1979.
- [8] Preas, B. T. and C. S. Chow, "Placement and Routing Algorithms for Topological Integrated Circuit Layout," *Proceedings of the International Symposium on Circuits and Systems*, Kyoto, Japan, June 1985.
- [9] Rivest, R. L., "The PI (Placement and Interconnect) System," *Proceedings of the 19th Design Automation Conference*, Las Vegas, Nevada, June 1982.
- [10] Slutz, E. A., *Shape Determination and Placement Algorithms for Hierarchical Integrated Circuit Layout*, Stanford University, Department of Electrical Engineering, Ph. D. Dissertation, 1980.
- [11] Supowit, K. J. and E. A. Slutz, "Placement Algorithms for Custom VLSI," *Proceedings 20th Design Automation Conference*, Miami Beach, Florida, June 1983.
- [12] Soukup, J., "Circuit Layout," *Proceedings of The IEEE*, vol. 69, No. 10, October 1981.
- [13] Ullman, J. D., *Computational Aspects VLSI*, Computer Science Press, 1984.
- [14] Mandelbrot, B., *The Fractal Geometry of Nature*, W.H. Freeman, 1983.
- [15] Garey, M and D. Johnson, *Computers and Intractability: A Guide to The Theory of NP-Completeness*, W.H. Freeman, 1979.