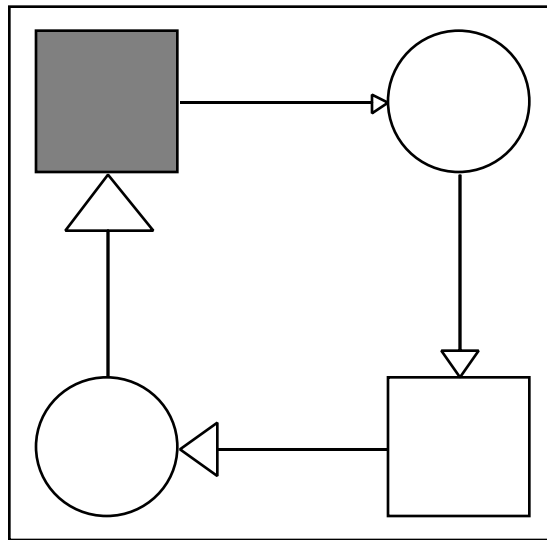


# Design/CPN Tutorial for X-Windows

Version 2.0



## Meta Software Corporation



125 CambridgePark Drive  
Cambridge, MA 02140 U.S.A.  
Tel: (617) 576-6920  
Fax: (617) 661-2008

© 1993 Meta Software

© 1993 Meta Software Corporation

125 CambridgePark Drive

Cambridge, MA 02140

(617) 576-6920

FAX: (617) 661-2008

email: [cpn-tech-support@metasoft.com](mailto:cpn-tech-support@metasoft.com)

Design/CPN is a trademark of Meta Software Corporation.

X-Windows is a trademark of the Massachusetts Institute of Technology.

# Design/CPN Tutorial for X-Windows

Version 2.0

## Table of Contents

### Part 1: CP Net Fundamentals

#### Chapter 1

##### The Design/CPN Tutorial

What Is a Petri Net?.....	1-1
Overview of the Design/CPN Tutorial.....	1-2
Part 1: CP Net Fundamentals.....	1-2
Part 2: Design/CPN Techniques.....	1-2
Appendix A: CPN Hierarchy Techniques.....	1-2
Appendix B: The Sales Order Model.....	1-3
Appendix C: Troubleshooting.....	1-3
Strategy of the Tutorial.....	1-3
How to Use the Tutorial.....	1-4
Proceed Systematically.....	1-4
Ignore the Unexplained.....	1-4
Review Frequently.....	1-5
Build the Models!.....	1-5
Beyond the Tutorial.....	1-5
Request For Feedback.....	1-6

#### Chapter 2

##### Getting Started With Design/CPN

What Is Design/CPN.....	2-1
Prerequisites for This Tutorial.....	2-2
Using This Tutorial With X-Windows.....	2-2

## Chapter 2

### Getting Started With Design/CPN (cont'd)

Design/CPN and X-Windows.....	2-3
Design/CPN Multiprocessing.....	2-3
Design/CPN and the File System.....	2-3
Design/CPN Use of the Mouse.....	2-4
Design/CPN Use of the Keyboard.....	2-4
Establishing a Tutorial Directory.....	2-4
Starting Design/CPN.....	2-5
Opening a Diagram.....	2-5
The Design/CPN User Interface.....	2-5
The Menu Bar.....	2-6
The Status Bar.....	2-6
The Page.....	2-6
Navigating a Diagram.....	2-7
Printing a Diagram.....	2-9
Closing a Diagram.....	2-9
Quitting Design/CPN.....	2-10
Starting and Stopping the Tutorial.....	2-10

## Chapter 3

### Modeling Paradigms

Static Modeling Paradigms.....	3-1
IDEF0 Modeling.....	3-2
Dynamic Modeling Paradigms.....	3-3
CP Net Modeling.....	3-3
IDEF0 Modeling and CP Net Modeling.....	3-4

## Chapter 4

### Using the Design/CPN Editor

The Design/CPN Graphics Editor.....	4-1
Design/CPN Graphical Objects.....	4-2
Graphics Editor Modes.....	4-3
Editing Graphical Objects.....	4-3
Graphics Mode.....	4-3
Text Mode.....	4-4
Creating Graphical Objects.....	4-4
Autoscrolling.....	4-5
Keystroke Shortcuts.....	4-5
Creating a New Diagram.....	4-6
Resetting the Drawing Environment.....	4-6
Caps Lock Under X-Windows.....	4-7

## Chapter 4 Using the Design/CPN Editor (cont'd)

Working With Rectangles.....	4-7
Creating a Rectangle.....	4-7
Enter Rectangle Creation Mode.....	4-7
Specify the First Corner.....	4-8
Specify the Diagonal Corner.....	4-8
Finish the Rectangle.....	4-8
Leave Rectangle Creation Mode.....	4-9
Reshaping a Rectangle.....	4-9
Moving a Rectangle.....	4-9
Deleting a Rectangle.....	4-10
Moving a Rectangle While Creating It.....	4-10
Adding Text to a Rectangle.....	4-11
Creating a Series of Rectangles.....	4-11
Adding Text to a Rectangle While Creating It.....	4-12
Preserving a Rectangle's Aspect Ratio.....	4-12
Working With More Than One Object on a Page.....	4-13
Selecting an Object.....	4-14
Selection After Deletion.....	4-14
Working With Ellipses.....	4-15
Creating an Ellipse.....	4-15
Enter Ellipse Creation Mode.....	4-15
Specify the First Corner.....	4-16
Specify the Diagonal Corner.....	4-16
Finish the Ellipse.....	4-16
Leave Ellipse Creation Mode.....	4-16
Other Operations With Ellipses.....	4-16
Working With More Than One Object Type.....	4-17
Creating Objects From Text Mode.....	4-17
Working With Connectors.....	4-18
Creating a Connector.....	4-18
Routing a Connector.....	4-19
Editing a Connector.....	4-20
Automatic Rerouting of Connectors.....	4-20
Deletion of Dangling Connectors.....	4-20
Working With Labels.....	4-21
Creating a Label.....	4-21
Enter Label Creation Mode.....	4-21
Create the Label.....	4-21
Enter and Edit Text.....	4-21
Create Additional Labels.....	4-22
Other Operations With Labels.....	4-22
Nodes and Regions.....	4-22
Designating a Region.....	4-23
Restoring the Independence of a Region.....	4-24

### **Chapter 4** **Using the Design/CPN Editor (cont'd)**

Nodes and Regions (cont'd)	
Editing Parents and Regions.....	4-24
Moving a Region's Parent.....	4-24
Deleting a Region's Parent.....	4-25
Groups of Objects.....	4-25
Mixed Groups.....	4-26
Selecting a Group.....	4-26
Deselecting a Group.....	4-27
Reconstructing a Group.....	4-27
Operating on Groups.....	4-27
Intermission.....	4-28

### **Chapter 5** **CP Net Components**

The CPN ML Language.....	5-1
A CP Net Example.....	5-2
Nets and Models.....	5-3
CP Net Data.....	5-3
Colorsets.....	5-4
Enumerated Colorsets.....	5-4
String and Integer Colorsets.....	5-6
Duplicate Colorsets.....	5-6
Tokens.....	5-7
Multisets of Tokens.....	5-7
Specifying Multisets.....	5-8
Multiset Addition.....	5-8
Multiset Subtraction.....	5-9
Multiset Subsets.....	5-9
CPN Variables.....	5-10
Places.....	5-11
Place Markings.....	5-12
States and Markings.....	5-12
Initial Marking Regions.....	5-12
Appearance of Markings.....	5-13
Transitions.....	5-13
Arcs.....	5-14
Arc Inscriptions.....	5-15
Guards.....	5-16
CP Net Execution.....	5-16

## Chapter 6

### Creating a Net With Design/CPN

Auxiliary Graphics and CPN Graphics.....	6-1
Setting the Graphical Environment.....	6-2
Object Attributes.....	6-2
Diagram Default Attributes.....	6-3
System Default Attributes.....	6-3
Changing the Display Attributes.....	6-3
Establishing an Environment.....	6-3
Creating the Net.....	6-5
Creating the Transition.....	6-5
Naming the Transition.....	6-7
Creating the Transition's Guard.....	6-8
Creating the Input Place.....	6-9
Naming the Place.....	6-10
Specifying the Place's Colorset and Initial Marking.....	6-11
Creating the Output Place.....	6-13
Creating the Arcs and Arc Inscriptions.....	6-13
Creating the Global Declaration Node.....	6-15
Saving the Diagram.....	6-17
More Efficient Editing Techniques.....	6-17

## Chapter 7

### CP Net Dynamics

Executing CP Nets.....	7-1
The Design/CPN Simulator.....	7-1
Understanding CP Net Execution.....	7-2
When Can a Transition Occur?.....	7-2
Input Arc Inscriptions.....	7-2
Guards.....	7-3
Criteria for Enablement.....	7-3
Examples in This Chapter.....	7-3
Specifying Exact Token Values.....	7-4
Specifying a Single Token.....	7-4
The Simulator's Algorithm.....	7-4
Omitting a Count of One.....	7-5
Specifying More Than One Token Instance.....	7-5
Specifying More Than One Token Value.....	7-5
The General Rule.....	7-6
Specifying Variable Token Values.....	7-7
Binding an Arc Inscription Variable.....	7-7
Constraining Token Values.....	7-8
Guard Syntax.....	7-8
Use of Parentheses.....	7-8
Shortcut for andalso.....	7-9
Constraining a Single Token.....	7-9

## Chapter 7

### CP Net Dynamics (cont'd)

Constraining Token Values (cont'd)	
More Complex Constraints.....	7-10
Constraining More Than One Token.....	7-11
What Happens When a Transition Occurs.....	7-11
A Simple Example.....	7-12
Rebind any CPN Variables Per the Enabling Binding.....	7-12
Evaluate Each Input Arc Inscription.....	7-13
Remove the Enabling Multiset from Each Input Place.....	7-13
Evaluate Each Output Arc Inscription.....	7-13
Put the Output Multiset into the Output Place.....	7-14

## Chapter 8

### Executing a Net With Design/CPN

Loading ML Configuration Information.....	8-2
Performing a Syntax Check.....	8-2
Designating a Prime Page.....	8-4
Entering the Simulator.....	8-7
Simulation Regions.....	8-8
Simulation Regions Indicating Place Markings.....	8-9
Simulation Region Indicating Enablement and Firing.....	8-9
The Sim Menu.....	8-10
Executing the Net.....	8-10
Observing Net Execution.....	8-12
Re-Executing the Net.....	8-13
Order of Net Execution Events.....	8-16
Canceling Net Execution.....	8-18
Leaving the Simulator.....	8-19
Leaving During Execution.....	8-19
Removing Simulation Regions.....	8-20

## Chapter 9

### Handling CP Net Syntax Errors

Opening the Net.....	9-1
Missing Colorset Specification.....	9-2
Locating the Error.....	9-3
Text Pointers.....	9-4
Fixing the Error.....	9-5
Undeclared Variables.....	9-6
Locating the Error.....	9-7
Fixing the Error.....	9-8
Illegal CPN ML Constructs.....	9-9
Conclusion.....	9-10



## Part 2: Design/CPN Techniques

### Chapter 10

#### Extending a CP Net

Building SalesNet.....	10-2
Modifying the Global Declaration Node.....	10-2
Modifying the Guard.....	10-2
Extending the Graphics.....	10-3
Performing a Syntax Check.....	10-3
Discussion of the Model.....	10-3
Description of the System.....	10-3
How SalesNet Represents the System.....	10-4
Entities and Colorsets.....	10-4
Locations for Storing Data.....	10-5
Activities for Transforming Data.....	10-6
Data and Conditions Needed for Activities to Occur.....	10-6
Data That Will Be Produced if an Activity Occurs.....	10-8
What Happens When SalesNet Executes.....	10-9
Rebind Any CPN Variables Per the Enabling Binding.....	10-9
Evaluate Each Input Arc Inscription.....	10-9
Evaluate Each Output Arc Inscription.....	10-10
Remove the Enabling Multiset from Each Input Place.....	10-10
Put the Output Multiset into Each Output Place.....	10-11
Continue Execution.....	10-12
Executing SalesNet.....	10-12
Setting Substep Options.....	10-13
Adjusting Simulation Regions.....	10-13
Key and Popup Regions.....	10-15
Repositioning Simulation Regions.....	10-16
Continuing Execution.....	10-17
Creating a Page for Global Declarations.....	10-17
Creating a New Page.....	10-18
Naming the Page.....	10-18
Improving the Hierarchy Page.....	10-19
Renaming a Page From the Hierarchy Page.....	10-20
Moving the Global Declaration Node.....	10-20
Saving the Net.....	10-21

### Chapter 11

#### Concurrency and Choice

Concurrency Problems.....	11-1
Representing Concurrency.....	11-2
Multiple Enabling Bindings.....	11-2
Concurrent Transition Firing.....	11-3
Identical Enabling Bindings.....	11-3

## Chapter 11 Concurrency and Choice (cont'd)

Concurrent CP Net Execution.....	11-4
Initial State of the Net.....	11-4
Breakpoint 1: Beginning of Substep.....	11-4
Breakpoint 2: End of Substep.....	11-4
Execution Is Complete.....	11-5
Analysis of the Execution.....	11-5
Representing Conflict.....	11-5
Conflicts and Bindings.....	11-6
Concurrent Execution of SalesNet.....	11-7
Loading ML Configuration Information.....	11-8
Adding More Equipment.....	11-8
Executing SalesNet.....	11-9
Breakpoint 1: Beginning of Substep.....	11-10
Breakpoint 2: End of Substep.....	11-11
Execution Is Complete.....	11-11
Analysis of the Execution.....	11-12
Changing a Net in the Simulator.....	11-12
The Simulator's Execution Algorithm.....	11-14
Executing SalesNet With Conflict.....	11-15
Executing SalesNet With Conflict.....	11-15
1: Establish Initial Markings.....	11-15
2: Put All Enabled Transitions on the Enabled List.....	11-15
3A: Construct an Occurrence Set.....	11-15
3B: Execute the Elements in the Occurrence Set.....	11-16
Executing an Occurrence Set.....	11-16
SalesNet's Appearance at Breakpoint 1.....	11-17
SalesNet's Appearance at Breakpoint 2.....	11-18
3c: Update the Enabled List.....	11-20
4: Continue Execution.....	11-20
5: Complete Execution.....	11-20
Experimenting With Concurrency and Conflict.....	11-21

## Chapter 12 CPN Hierarchical Decomposition

Definition of Hierarchical Decomposition.....	12-1
Top-Down and Bottom-Up Development.....	12-2
Creating a Hierarchical Decomposition.....	12-2
Designating the Transition to Decompose.....	12-3
Initiating Subpage Creation.....	12-3
Specifying the Substitution Transition's Location.....	12-3
Naming the Substitution Transition.....	12-5
Improving the Substitution Transition's Appearance.....	12-6
Connecting Superpages to Subpages.....	12-8

## Chapter 12

### CPN Hierarchical Decomposition (cont'd)

Structure of the Subpage.....	12-8
How Design/CPN Creates a Decomposition.....	12-9
Simplifying the Decomposition Page.....	12-10
Substitution and the Hierarchy Page.....	12-11
Improving the Hierarchy Page's Appearance.....	12-12
Renaming the Page.....	12-13
Saving the Net.....	12-14

## Chapter 13

### Understanding a Simple Model

Overview of FirstModel.....	13-1
FirstModel and SalesNet Compared.....	13-2
Structure of FirstModel.....	13-3
Data Declarations in FirstModel.....	13-3
Tuple Colorsets.....	13-4
Tuples in FirstModel.....	13-4
The Superpage in FirstModel.....	13-5
SalesNet.....	13-5
FirstModel Superpage.....	13-6
The Subpage in FirstModel.....	13-6
Function of FirstModel.....	13-7
Enter Order.....	13-8
Tuple Constructors.....	13-8
Example of Tuple Construction.....	13-9
Process Order.....	13-10
Bidirectional Arcs.....	13-11
Tuple Patterns.....	13-12
Enablement of Process Order.....	13-13
Firing of Process Order.....	13-14
Ship Product.....	13-15
Summary of FirstModel.....	13-16
Entering an Order.....	13-17
Processing an Order.....	13-17
Shipping an Order.....	13-17
Concurrency in FirstModel.....	13-18
Locality in CP Nets.....	13-19
Locality and Arc Inscription Variables.....	13-19
Locality and Overview.....	13-19
Emergent Behavior in CP Nets.....	13-20

## Chapter 14

### Building a Simple Model

Adding Global Declarations.....	14-1
Modifying the Superpage.....	14-2
Building FirstModel on the Subpage.....	14-4
The Current Subpage.....	14-4
The Future Subpage.....	14-5
Editing the Subpage.....	14-6
The Starting Point.....	14-7
Rearranging the Ports.....	14-7
Creating the Transitions.....	14-8
Matching the Transition Sizes.....	14-10
Naming the Transitions.....	14-10
Creating and Naming the Places.....	14-12
Give the New Places Their Colorsets.....	14-14
Aligning Net Components.....	14-16
Diagonally Aligning the New Nodes.....	14-16
Horizontal Spread.....	14-16
Vertical Spread.....	14-17
Aligning Nodes Into a Row.....	14-17
Aligning Nodes Into a Column.....	14-18
Other Adjustments.....	14-18
Connect the Net Components With Arcs.....	14-18
Drawing a Bidirectional Arc.....	14-19
Adjusting Arc Appearance.....	14-22
Creating the Arc Inscriptions.....	14-24
Copying and Pasting Text Regions.....	14-26
Creating the Transition Guards.....	14-29
Final Adjustments to the Net.....	14-31
Performing a Syntax Check.....	14-32

## Chapter 15

### Executing a Simple Model

Executing the Net.....	15-1
Analysis of Execution.....	15-3
Subpages and Initial Markings.....	15-4
Experimenting With FirstModel.....	15-6
How to Do Experiments.....	15-6
Analysis of Execution.....	15-7
Complicating FirstModel.....	15-7
Using a Guard to Create a Partial Constraint.....	15-8
Executing the Net.....	15-8
Analysis of Execution.....	15-9

## Chapter 15

### Executing a Simple Model (cont'd)

Controlling the Appearance of Concurrency.....	15-9
Review of Occurrence Sets.....	15-10
Constructing an Occurrence Set.....	15-10
What Is Concurrency?.....	15-11
Occurrence Set Parameters.....	15-11
Scope of Occurrence Set Parameters.....	15-14
Setting Occurrence Set Parameters.....	15-14
Experimenting With Net Execution.....	15-14
Faster Model Execution.....	15-15
Interactive Mode.....	15-16
Automatic Mode.....	15-16
Fair and Fast Execution.....	15-16
Selecting the Execution Mode.....	15-17
Specifying Possible Execution Modes.....	15-17
Specifying the Actual Execution Mode.....	15-18
Specifying Stop Criteria.....	15-19
Automatic Net Execution.....	15-20
Alternating Execution Modes.....	15-21
Saving and Loading Execution States.....	15-22

## Chapter 16

### Simulated Time

The Nature of Simulated Time.....	16-2
Non-Representation of Time in FirstModel.....	16-3
Duration and Causality.....	16-4
Representing Time in a CP Net.....	16-4
How Simulated Time Works.....	16-5
Simulated Time and Transition Enablement.....	16-5
The Simulated Clock.....	16-6
Other Uses for Simulated Time.....	16-6
Specifying Timed Simulation.....	16-7
Declaring a Timed Colorset.....	16-7
Giving a Token a Time Stamp.....	16-7
Delay Expressions in Time Regions.....	16-8
Delay Expressions on Output Arc Inscriptions.....	16-8
Omitting a Time Stamp.....	16-9
Time Stamps and Initial Markings.....	16-10
Time Stamps and Multisets.....	16-11
Changing FirstModel to Assign Time Stamps.....	16-11
Compiling a Timed Net.....	16-14
Executing a Timed Net.....	16-16
Simulation With and Without Time.....	16-18
More Realistic Timed Behavior.....	16-19
Observing Simulation Results.....	16-21

## Appendix A: CPN Hierarchy Techniques

### Chapter A1

#### Introduction to Hierarchy

Files for Use With This Appendix.....	A1-1
CPN Hierarchy.....	A1-1
Fusion Places.....	A1-2
Substitution Transitions.....	A1-2

### Chapter A2

#### Fusion Places

The Resource Use Model.....	A2-1
Description of the Model.....	A2-3
Executing the Model.....	A2-4
Fusion on a Single Page.....	A2-4
Results of Executing the Diagram.....	A2-5
Combining the Resource Pools.....	A2-6
Creating a Fusion Set.....	A2-6
Physical Appearance of a Global Fusion Place.....	A2-8
Adding Places to a Fusion Set.....	A2-9
Initial Markings and Fusion Sets.....	A2-10
Removing Places From a Fusion Set.....	A2-10
Deleting a Fusion Set.....	A2-11
Fusion Across More Than One Page.....	A2-11
Saving and Loading a Subdiagram.....	A2-12
Make the New Page Prime.....	A2-13
Working With Fusion Sets That Span Pages.....	A2-13
Working With More Than One Fusion Set.....	A2-14
Page Fusion Sets.....	A2-17
Creating a Page Fusion Set.....	A2-18
Watching Fusion in Action.....	A2-19
Instance Fusion Sets.....	A2-20
Creating Multiple Page Instances.....	A2-21
Multiplicity and Fusion.....	A2-23
Working With Instance Fusion Sets.....	A2-23
Observing Fusion Across Multiple Instances.....	A2-25

### Chapter A3

#### Substitution Transitions

Structure of a Diagram With Substitution.....	A3-2
The Hierarchy Page.....	A3-2
The Superpage Resmod#1.....	A3-3
The Subpage New#2.....	A3-4

## Chapter A3 Substitution Transitions (cont'd)

Structure of a Diagram With Substitution (cont'd)	
Ports and Sockets.....	A3-6
Jumping Directly to a Superpage.....	A3-6
Overall Structure of the Diagram.....	A3-6
Creating a Substitution Transition.....	A3-7
Designate the Net Components to Move to the Subpage.....	A3-9
Initiate Subpage Creation.....	A3-9
Specify the Substitution Transition's Location.....	A3-9
Name the Substitution Transition (If Desired).....	A3-10
Status of the Diagram.....	A3-11
Improving the Net's Appearance.....	A3-11
Improving the Superpage's Appearance.....	A3-11
Rerouting the Arc.....	A3-12
Moving the Regions.....	A3-13
Improving the Subpage's Appearance.....	A3-14
Improving the Hierarchy Page's Appearance.....	A3-16
Status of the Diagram.....	A3-18
Reversing Substitution Transition Creation.....	A3-18
Status of the Diagram.....	A3-20
Developing on a Subpage.....	A3-21
Create the Substitution Transition and Subpage.....	A3-21
The Modified Hierarchy Page.....	A3-23
The New Subpage.....	A3-23
Relationship of Pages in a Hierarchy.....	A3-24
Deleting a Subpage.....	A3-25
Using a Subpage More Than Once.....	A3-26
Structure of the Diagram.....	A3-31
Substitution Transitions and Multiplicity.....	A3-33
Subpages, Subroutines, and Macros.....	A3-34
Simulating With Hierarchy.....	A3-34
Deleting a Reference to a Subpage.....	A3-35
Manually Assigning Ports to Sockets.....	A3-37

## Appendix B: The Sales Order Model

### Chapter B1

#### Introduction to the Sales Order Model

Files for Use With This Appendix.....	B1-1
Overview of the Sales Order Model.....	B1-1
Entities Represented in the Model.....	B1-2
Action Cycle for Processing Orders.....	B1-2
Inefficiency in the Sales Order System.....	B1-3
Using the Model to Reduce Inefficiency.....	B1-3
Simulation Parameters.....	B1-4
Job Stream Parameters.....	B1-4
Job Value Parameters.....	B1-4
Staff Parameters.....	B1-4
Equipment Parameters.....	B1-5
Gathering and Displaying Statistics.....	B1-5
Revenue Statistics.....	B1-5
Efficiency Statistics.....	B1-6
Using the Sales Order Model.....	B1-6

### Chapter B2

#### Running the Sales Order Model

The Simulation Parameter File.....	B2-1
Restoring the Simulation Parameter File.....	B2-2
System Properties Specified by These Parameters.....	B2-2
Analysis of the Initial Parameters.....	B2-3
Running the Model.....	B2-4
Analyzing and Using Simulation Results.....	B2-5

### Chapter B3

#### Using the Sales Order Model

Interpreting the Results of a Simulation Run.....	B3-1
Examining the Revenue Charts.....	B3-1
Examining the Efficiency Charts.....	B3-3
Experimenting With Possible Improvements.....	B3-6
Changing the Simulation Parameters.....	B3-6
Performing the Experiment.....	B3-8
Interpreting the New Results.....	B3-9
Additional Experiments.....	B3-12
More General Use of the Sales Order Model.....	B3-13
Improving the Sales Order Model.....	B3-13
Analyzing the Problem.....	B3-14
Changing the Model.....	B3-14



## Appendix C: Troubleshooting

### Chapter C1

#### Troubleshooting

CPN Settings File Missing or Obsolete.....	C1-1
Problem Description.....	C1-1
Problem Solution.....	C1-2
ML Configuration Unspecified or Incorrect.....	C1-2
Identifying the Problem.....	C1-2
Copying Diagram Default ML Configuration Options.....	C1-3
Setting ML Configuration Options.....	C1-4
ML Interpreter Cannot Be Started.....	C1-5





# INDEX

## Special Characters

- () (parentheses),**  
guards use of; 7-8
- , (comma),**  
guards use as shorthand for boolean andalso operator;  
7-9
- < (less than),**  
boolean operator used in guards; 7-8
- < > (not equal),**  
boolean operator used in guards; 7-8
- <= (less than or equal),**  
boolean operator used in guards; 7-8
- = (equal),**  
boolean operator used in guards; 7-8
- > (greater than),**  
boolean operator used in guards; 7-8
- >= (greater than or equal),**  
boolean operator used in guards; 7-8
- @ + (delay expression),**  
characteristics and use with time stamps; 16-7
- [] (brackets),**  
guards use as distinguishing characters; 7-8
- ^ (multiset creation operator),**  
creating multisets with; 5-8

## A

- activation rules,**  
as dynamic modeling paradigm component; 3-3
- activities,**  
*See Also* transitions;  
as dynamic modeling paradigm component; 3-3  
as static modeling paradigm component; 3-1  
CP net transitions as representations for; 3-3  
representing with transitions; 10-6
- adding,**  
*See Also* creating;  
declarations to the global declaration nodes,  
to extend FirstNet into SalesNet; 10-2  
multisets; 5-8  
places to fusion sets; A2-9  
text,  
to a rectangle; 4-11  
to a rectangle, while creating it; 4-12
- address,**  
Meta Software; 1-7

- adjusting,**  
*See Also* aligning;  
simulation regions,  
for SalesNet execution; 10-13
- adjustment tool,**  
characteristics and illustration; 4-8
- algorithms,**  
*See Also* bindings;  
constructing occurrence sets,  
issues involved; 15-10  
occurrence set execution; 11-16  
simulator; 11-14  
illustrating with SalesNet model execution with  
conflict; 11-15
- Align menu,**  
Horizontal command,  
aligning nodes in a row with; 14-17  
Horizontal Spread command,  
aligning nodes with; 14-16  
Vertical command,  
aligning nodes in a column with; 14-18  
Vertical Spread command,  
aligning nodes with; 14-17
- aligning,**  
nodes,  
along a diagonal, with Align menu commands;  
14-16  
in a column, with Vertical command (Align menu);  
14-18  
in a row, with Horizontal command (Align menu);  
14-17  
with Align menu commands; 14-16
- Alt-DownArrow keys,**  
navigating to an error with; 9-5
- andalso (boolean AND),**  
boolean operator used in guards; 7-8
- appearance,**  
aligning net components with Align menu  
commands; 14-16  
arcs,  
adjusting; 14-22  
concurrency,  
controlling; 15-9  
effective space use,  
creating a separate page for global declarations;  
10-17  
global fusion place; A2-8  
hierarchical CP nets,  
improving; A3-11  
hierarchy page,  
improving; 12-12, A3-16  
matching transition sizes; 14-10

**appearance (cont'd),**

- subpage,
  - improving; A3-14
- substitution transition,
  - improving with Child Object command (Makeup menu); 12-6
- superpage,
  - improving; A3-11

**Arc (CPN menu),**

- connecting FirstModel nodes with; 14-18
- creating arcs with; 6-13

**arcs,**

- See Also* connectors;
- adjusting the appearance of; 14-22
- arc creation mode,
  - term definition; 6-13
- arc creation tool,
  - term definition and illustration; 6-13
- arc inscription creation mode,
  - term definition; 6-14
- arc inscription region,
  - term definition; 5-15
- as CP net connections; 3-3
- bidirectional,
  - characteristics and use; 13-11
  - drawing, for FirstModel; 14-19
- characteristics as CP net component; 5-1
- connecting FirstModel nodes with; 14-18
- creating; 6-13
- input arc inscriptions,
  - binding variables in; 7-7
  - evaluating during SalesNet execution; 10-9
  - evaluating during transition firing; 7-13
  - role in enabling transitions; 7-2
  - term definition; 3-4
- inscriptions,
  - characteristics and term definition; 5-15
  - CPN variables, locality and; 13-19
  - creating; 6-13
  - creating, for FirstModel; 14-24
- output arc inscriptions,
  - delay expressions on; 16-8
  - evaluating during SalesNet execution; 10-10
  - evaluating during transition firing; 7-13
  - term definition; 3-4
- rerouting; A3-12
- term definition,
  - and characteristics; 5-14

**aspect ratio,**

- rectangle,
  - preserving; 4-12

**assigning,**

- colorsets to FirstModel places; 14-14
- ports to sockets,
  - manually; A3-37
- time stamps,
  - FirstModel; 16-11

**attributes,**

- diagram default,
  - term definition and characteristics; 6-3
- display,
  - changing; 6-3
  - term definition; 6-2
- object,
  - term definition and characteristics; 6-2
- system default,
  - term definition and characteristics; 6-3

**automatic mode,**

- term definition and characteristics; 15-16

**Automatic Run (Sim menu),**

- executing a model with; 15-21
- running the Sales Order Model with; B2-4

**autoscrolling,**

- characteristics; 4-5

**Aux menu,**

- Box command,
  - creating rectangles with; 4-7
- Connector command,
  - creating connectors with; 4-18
- creating auxiliary graphical objects with; 4-4
- Ellipse command,
  - creating ellipses with; 4-15
- Label command,
  - creating rectangles with; 4-21
- Make Region command,
  - creating regions with; 4-23

**auxiliary objects,**

- See Also* graphical objects;
- term definition; 4-1

## B

**backquote (`),**

- as multiset creation operator; 5-8

**bar charts,**

- examining the Sales Order Model,
  - efficiency charts; B3-3
  - revenue charts; B3-1

**behavior,**

- See Also* modeling;
- CP net execution,
  - relation to real-world systems; 13-20

## **behavior (cont'd),**

- CP nets,
  - (chapter); 7-1
- modeling; 3-1
- timed,
  - increasing the realism of; 16-19

## **bidirectional arcs,**

- characteristics and use; 13-11
- drawing,
  - for FirstModel; 14-19

## **bindings,**

- See Also* algorithms; occurrence sets; tuples,
  - constructors; tuples, patterns;
- binding elements,
  - executing; 11-16
  - term definition; 11-14, 15-10
- conflict and; 11-6
- different,
  - setting their representation in an occurrence set; 15-13
- enabling,
  - identical; 11-3
  - multiple; 11-2
- identical,
  - setting their representation in an occurrence set; 15-13
- of CPN variables,
  - in input arc inscriptions; 7-7

## **boolean,**

- See Also* expressions; guards;
- AND (andalso),
  - boolean operator used in guards; 7-8
- NOT (not),
  - boolean operator used in guards; 7-8
- operators,
  - used in guards; 7-8
- OR (orelse),
  - boolean operator used in guards; 7-8
- tests,
  - constraining token values with; 7-8

## **border,**

- page,
  - term definition and characteristics; 2-6

## **bottom-up development,**

- term definition; 12-2

## **bound,**

- term definition; 7-7

## **Box (Aux menu),**

- creating rectangles with; 4-7

## **boxes,**

- See* rectangles;

## **brackets ([ ]),**

- guards use as distinguishing characters; 7-8

## **breakpoints,**

- See Also* simulation;
- beginning of substep,
  - characteristics; 8-12
  - concurrent execution of SalesNet model; 11-10
  - FirstNet model, concurrent execution; 11-4
  - FirstNetDemo model; 8-15
  - SalesNet's appearance; 11-17
- continuing execution after,
  - with Continue command (Sim menu); 8-16
- end of substep,
  - characteristics; 8-13
  - concurrent execution of SalesNet model; 11-11
  - FirstNet model, concurrent execution; 11-4
  - FirstNetDemo model; 8-16
  - SalesNet's appearance; 11-18
- setting,
  - for FirstModel; 15-3
  - for SalesNet execution; 10-13
  - with Interactive Simulation Options command (Set menu); 8-12

# C

## **canceling,**

- CP net execution,
  - with Stop command (Sim menu); 8-18

## **Caps Lock key behavior,**

- preserving rectangle aspect ratio during size change; 4-12
- X-Windows; 4-7

## **causality,**

- See Also* modeling; representation;
- representation in FirstModel; 16-4

## **Change Shape (Makeup menu),**

- changing transition shape with; A3-28
- matching transition sizes with; 14-10

## **changing,**

- CP nets,
  - in the simulator; 11-12
- display attributes; 6-3
- guards,
  - to extend FirstNet into SalesNet; 10-2
- initial markings,
  - to add more equipment to the SalesNet model; 11-8
- rules with guards,
  - FirstModel; 15-8

**changing (cont'd),**

- transition shape,
  - with Change Shape command (Makeup menu); A3-28

**charts,**

- bar,
  - examining the Sales Order Model efficiency charts; B3-3
  - examining the Sales Order Model revenue charts; B3-1

**Child Object (Makeup menu),**

- improving the appearance of a substitution transition with; 12-6
- selecting a region with; A3-13

**choice,**

- See Also* modeling;
- concurrency and,
  - (chapter); 11-1
- term definition; 11-1

**clock,**

- simulated,
  - mechanism characteristics; 16-6
- term definition; 16-5

**Close (File menu),**

- closing diagrams with; 2-9

**closed,**

- page,
  - term definition; 2-7

**closing,**

- diagrams; 2-9

**colored Petri nets,**

- See* CP nets; Design/CPN;

**colorsets,**

- See Also* places;
- assigning to FirstModel places; 14-14
- characteristics; 5-4
- composite,
  - term definition; 13-4
- duplicate,
  - characteristics and term definition; 5-6
- enumerated,
  - characteristics and term definition; 5-4
- integer,
  - characteristics and term definition; 5-6
- missing,
  - detecting and handling; 9-2
- representing entities with; 10-4
- specifying; 6-11
- string,
  - characteristics and term definition; 5-6
- term definition; 5-3

**colorsets (cont'd),**

- timed,
  - declaring; 16-7
- tuple,
  - characteristics and term definition; 13-4

**column,**

- aligning nodes in,
  - with Vertical command (Align menu); 14-18

**comma ( , ),**

- guards use as shorthand for boolean andalso operator; 7-9

**comments,**

- importance of in documenting a model; 14-2
- in models,
  - auxiliary objects use for; 4-2

**committed,**

- term definition; 8-15

**comparison,**

- operators,
  - used in guards; 7-8

**compiling,**

- timed CP nets; 16-14

**components,**

- CP net,
  - (chapter); 5-1

**composite colorset,**

- term definition; 13-4

**concurrency,**

- See Also* time;
- choice and,
  - (chapter); 11-1
- concurrent,
  - activities, term definition; 11-1
  - execution, SalesNet model; 11-7
  - system, term definition; 11-1
- conflict,
  - as limiting factor in; 11-6
  - issues; 15-11
- controlling the appearance of; 15-9
- CP net execution,
  - FirstNet model; 11-4
- experimenting with; 11-21
- firing multiple concurrent transitions; 11-3
- FirstModel; 13-18
- occurrence set construction in relation to; 15-11
- problems with; 11-1
- representing; 11-2
- term definition; 11-1
  - and modeling characteristics; 15-11
- very small occurrence sets and; 15-14

**conditions,**

- representing with guards; 10-6



- conflict,**
  - avoidance by occurrence set elements; 15-10
  - bindings and; 11-6
  - competition for resources,
    - as concurrency problem; 11-1
  - concurrency in relation to; 15-11
  - executing SalesNet model with; 11-15
  - experimenting with; 11-21
  - representing; 11-5
  - resource,
    - modeling with FirstModel; 15-7
  - term definition; 11-1
- connecting,**
  - superpages to subpages; 12-8
- connections,**
  - as dynamic modeling paradigm component; 3-3
  - as static modeling paradigm component; 3-1
  - CP net arcs as representations for; 3-3
- Connector (Aux menu),**
  - creating connectors with; 4-18
- connectors,**
  - See Also* arcs;
  - creating; 4-18
  - deleting,
    - dangling; 4-20
  - editing; 4-20
  - routing,
    - automatic; 4-20
  - routing; 4-19
  - term definition; 4-2, 18
- constants,**
  - See Also* CPN variables;
  - as exact token values; 7-4
- constraining,**
  - See Also* guards;
  - token values; 7-8
    - with more complex guards; 7-10
    - with simple guards; 7-9
- constraints,**
  - partial,
    - adding with guards in FirstModel; 15-8
  - representing with guards; 10-6
  - term definition; 7-8
- constructing,**
  - See Also* creating;
  - occurrence sets,
    - for SalesNet model execution with conflict; 11-15
  - occurrence sets; 15-10
- constructors,**
  - tuple,
    - characteristics; 13-8
    - example of use in First Model; 13-9
- constructors (cont'd),**
  - tuple (cont'd),
    - term definition; 13-9
- Continue (Sim menu),**
  - continuing execution after a breakpoint with; 11-18
  - continuing execution after a breakpoint with; 8-16
- continuing,**
  - CP net execution,
    - with Continue command (Sim menu); 8-16
- Copy Defaults (Set menu),**
  - copying diagram defaults with; 6-4
- copying,**
  - text regions; 14-26
- costs,**
  - Sales Order Model chart depicting elapsed,
    - examining; B3-5
  - Sales Order Model chart depicting incurred; B3-3
- CP nets,**
  - canceling execution,
    - with Stop command (Sim menu); 8-18
  - changing,
    - in the simulator; 11-12
  - compared with IDEF0 modeling; 3-4
  - components,
    - (chapter); 5-1
  - creating; 6-5
    - (chapter); 6-1
  - distributing,
    - across multiple pages; 12-1
  - dynamic modeling paradigm characteristics; 3-3
  - dynamics,
    - (chapter); 7-1
  - example description; 5-2
  - executing; 7-1, 7-14, 8-10
    - (chapter); 8-1
  - FirstNet model, concurrent execution; 11-4
  - order of execution events; 8-16
  - overview; 5-16
  - SalesNet, interactive execution; 10-12
  - SalesNet, simulator actions; 10-9
  - starting execution, with Interactive Run command
    - (Sim menu); 8-14
  - with fusion sets; A2-19
  - with instance fusion sets; A2-25
- extending,
  - FirstNet into SalesNet (chapter); 10-1
- hierarchical,
  - developing on a subpage; A3-21
  - diagram structure; A3-2
  - improving appearance of; A3-11
  - introduction (chapter); 12-1
  - relationship among pages in; A3-24

## **CP nets (cont'd),**

- incremental development,
  - prime page role; 8-4
- locality,
  - term definition; 13-19
- models compared with; 5-3
- modularity,
  - prime page role; 8-4
- moving components between pages; 10-20
- multi-page,
  - interconnecting; 12-1
- observing execution; 8-12
- opening,
  - with Open (File menu); 8-1
- saving; 6-17
- structure,
  - hierarchical, adding; 12-2
- syntax errors,
  - handling, (chapter); 9-1
- term definition; 1-2
- timed,
  - compiling; 16-14
  - executing; 16-16

## **CPN (colored Petri net),**

- hierarchy,
  - term definition; 12-1
- ML language,
  - role in CP nets; 5-1
- model,
  - term definition; 1-2
- objects,
  - term definition; 4-1, 6-2

## **CPN menu,**

- Arc command,
  - connecting FirstModel nodes with; 14-18
  - creating arcs with; 6-13
- CPN Region command,
  - assigning colorsets to FirstModel places with; 14-14
  - assigning time stamps with; 16-12
  - creating arc inscriptions for FirstModel with; 14-24
  - creating arc inscriptions with; 6-14
  - creating guards for FirstModel with; 14-29
  - creating guards with; 6-8
  - fixing a syntax error with; 9-6
  - naming FirstModel places with; 14-12
  - naming FirstModel transitions with; 14-10
  - naming places with; 6-10
  - naming substitution transitions with; 12-6, A3-10
  - naming transitions with; 6-7
  - specifying, colorsets with; 6-11

## **CPN menu (cont'd),**

- CPN Region command (cont'd),
  - specifying, initial markings with; 6-12
- Declaration Node command,
  - creating a global declaration node with; 6-15
- Fusion Place command,
  - adding places to a fusion set with; A2-9
  - creating a fusion set with; A2-6
  - creating instance fusion sets with; A2-24
  - creating multi-page fusion sets with; A2-14
  - creating multiple fusion sets with; A2-15
  - creating page fusion sets with; A2-18
  - deleting fusion sets with; A2-11
  - removing places from a fusion set with; A2-10
- Move to Subpage command,
  - creating subpages with; 12-3, A3-9
  - top-down hierarchical CP net development with; A3-22
- Place command,
  - creating places with; 6-9
- Port Assignment command,
  - manually assigning ports to sockets with; A3-39
- Remove Sim Regions command,
  - cleaning up diagrams during hierarchy simulation; A3-35
  - removing simulation regions with; 8-20
- Replace by Subpage command,
  - reversing substitution transition creation with; A3-19
- Substitution Transition command,
  - converting a transition to a substitution transition with; A3-28
  - creating substitution transitions with manual port assignments; A3-38
- Syntax Check command,
  - performing a syntax check with; 8-2
- Transition command,
  - creating transitions with; 6-5

## **CPN Region (CPN menu),**

- assigning,
  - colorsets to FirstModel places; 14-14
  - time stamps; 16-12
- creating,
  - arc inscriptions; 6-14
  - arc inscriptions for FirstModel; 14-24
  - guards for FirstModel; 14-29
  - guards; 6-8
- fixing a syntax error with; 9-6
- naming,
  - FirstModel places; 14-12
  - FirstModel transitions; 14-10
  - places; 6-10

**CPN Region (CPN menu) (cont'd),**

- naming (cont'd),
  - substitution transitions; 12-6
  - substitution transitions; A3-10
  - transitions; 6-7

- specifying,
  - colorsets with; 6-11
  - initial markings with; 6-12

**CPN variables,**

- arc inscriptions,
  - locality and; 13-19
- binding in input arc inscriptions; 7-7
- characteristics and term definition; 5-10
- multiple bindings resulting from locality; 13-19
- rebinding,
  - during SalesNet execution; 10-9
  - during transition firing; 7-12
- specifying token values with; 7-7
- undeclared,
  - detecting and handling; 9-6

**creating,**

- See Also* constructing; drawing; editing;
- arc inscriptions; 6-13
  - for FirstModel; 14-24
- arcs; 6-13
- connectors; 4-18
- CP nets; 6-5
  - (chapter); 6-1
- decomposition page,
  - Design/CPN actions; 12-9
- diagrams; 4-6
- ellipses; 4-15
- fusion sets,
  - global; A2-6
  - instance; A2-20
  - page; A2-17
- graphical objects; 4-4
- graphical objects,
  - from text mode; 4-17
- guards; 6-8
  - for FirstModel; 14-29
- hierarchical CP nets,
  - by developing on a subpage; A3-21
- labels; 4-21
- models,
  - simple (chapter); 14-1
- multisets; 5-8
- nodes,
  - global declaration; 6-15
- page instances; A2-21
- pages,
  - for global declarations; 10-17

**creating (cont'd),**

- places,
  - for FirstModel subpage; 14-12
  - input; 6-9
  - output; 6-13
- rectangles; 4-7
  - a series of; 4-11
  - adding text while creating; 4-12
- regions; 4-23
- subpages; 12-3, A3-9
- substitution transitions; A3-7
  - with Substitution Transition (CPN menu); A3-38
- transitions; 6-5

**current,**

- marking key region,
  - term definition; 8-9
- marking region,
  - term definition; 8-9
- markings,
  - simulation regions describing; 8-9
  - term definition; 5-12
- object,
  - term definition; 4-13
- state,
  - term definition; 5-12

**cursor keys,**

- navigating to an error with; 9-4

**customer requests (Sales Order Model),**

- characteristics; B1-2

## D

**data,**

- as dynamic modeling paradigm component; 3-3
- characteristics as CP net component; 5-1
- CP net characteristics; 3-3
- declarations,
  - in FirstModel; 13-3
- objects,
  - output,
    - specifying with output arc inscriptions; 10-8

**datatype,**

- See* colorsets;

**DB file,**

- term definition; 2-3

**debugging,**

- controlling the appearance of concurrency; 15-9

**Declaration Node (CPN menu),**

- creating a global declaration node with; 6-15

**declarations,**

*See Also* colorsets; global, declaration node;  
data,  
    in FirstModel; 13-3  
declaration node tool,  
    term definition and illustration; 6-15  
global declaration nodes,  
    adding declarations to extend FirstNet into  
        SalesNet; 10-2  
    creating; 6-15  
    declaring a timed colorset in; 16-7

**declaring,**

timed colorsets; 16-7

**decomposition,**

*See Also* substitution transitions;  
page,  
    creating, Design/CPN actions; 12-9  
    simplifying; 12-10  
term definition; 12-2  
transitions,  
    methods for specifying; 12-2

**deciphering,**

ambiguous syntax error messages; 9-7

**delay expressions,**

*See Also* time;  
term definition and syntax; 16-7

**deleting,**

*See Also* creating;  
connectors,  
    dangling; 4-20  
fusion sets,  
    with Fusion Place (CPN menu); A2-11  
graphical objects; 4-14  
pages; A3-25  
parents; 4-25  
rectangles; 4-10  
subpages,  
    references to, from a hierarchical substitution  
        transition; A3-35  
subpages; A3-25

**descriptions,**

as static modeling paradigm component; 3-2

**deselecting,**

*See Also* selecting;  
groups; 4-27

**Design/CPN,**

*See Also* CP nets;  
characteristics and components; 2-1  
data,  
    characteristics; 5-3  
editor,  
    Design/CPN; 4-1

**Design/CPN (cont'd),**

getting started with (chapter); 2-1  
hierarchical decomposition,  
    substitution transitions, introduction (chapter);  
        12-1  
hierarchy,  
    characteristics; A1-1  
quitting; 2-10  
settings file missing or obsolete,  
    problem symptoms and solutions; C1-1  
simulator,  
starting; 2-5  
tutorial,  
    design strategy; 1-3  
    document components overview; 1-2  
    how to use; 1-4  
    introduction (chapter); 1-1  
    prerequisites; 2-2  
    user interface; 2-5

**designating,**

*See Also* specifying;  
prime pages; 8-4  
    with Mode Attributes (Set menu); 8-5  
transition decomposition; 12-3

**detecting,**

*See Also* troubleshooting;  
errors,  
    ML; 9-9

**diagonal,**

aligning nodes along,  
    with Align menu commands; 14-16

**diagrams,**

closing; 2-9  
creating; 4-6  
default attributes,  
    term definition and characteristics; 6-3  
file,  
    term definition; 2-3  
navigating; 2-7  
opening; 2-5  
printing; 2-9  
saving; 6-17  
term definition; 2-3

**directory,**

establishing a tutorial; 2-4

**display attributes,**

*See Also* attributes;  
changing; 6-3  
graphical objects,  
    term definition; 4-3  
term definition; 6-2

**distributing**,  
     CP net,  
         across multiple pages; 12-1

**documenting**,  
     models,  
         with auxiliary objects; 4-2  
         with comments in the global declaration node;  
             14-1

**Drag (Makeup menu)**,  
     moving a region with; A3-13  
     moving hierarchy key region with; 12-6

**drag mode**,  
     term definition; 12-7

**drag tool**,  
     term definition; 12-7

**drawing**,  
     *See Also* creating;  
     graphics tool used for; 4-3  
     rectangles; 4-7  
     tool,  
         term definition; 4-3

**duplicate colorsets**,  
     term definition and characteristics; 5-6

**dynamics**,  
     CP nets,  
         (chapter); 7-1

## E

**e-mail address**,  
     Meta Software; 1-7

**editing**,  
     *See Also* creating;  
     connectors; 4-20  
     graphical objects; 4-3  
     more efficient techniques; 6-17  
     regions; 4-24  
     text; 4-21

**editor**,  
     Design/CPN,  
         (chapter); 4-1

**efficiency**,  
     in a Sales Order system,  
         examining with simulation charts; B3-3

**Ellipse (Aux menu)**,  
     creating ellipses with; 4-15

**ellipses**,  
     *See Also* graphical objects; places;  
     creating; 4-15  
     creation mode,  
         characteristics; 4-15

**ellipses (cont'd)**,  
     tool,  
         characteristics and illustration; 4-15

**empty multiset**,  
     *See Also* multisets;  
     term definition; 5-7

**enabled**,  
     list,  
         putting all enabled transitions on, for SalesNet  
             model execution with conflict; 11-15  
         scanning, for SalesNet model execution with  
             conflict; 11-15  
         term definition; 7-4, 11-14  
         updating, for SalesNet model execution with  
             conflict; 11-20  
     term definition; 7-2  
     with a binding,  
         term definition; 7-7

**enablement**,  
     *See Also* algorithms; bindings; occurrence sets;  
     criteria for; 7-3  
     factors determining; 7-2  
     identical enabling bindings; 11-3  
     multiple enabling bindings; 11-2  
     simulated time impact on; 16-5  
     simulation region identifying; 8-9  
     term definition; 7-2

**enabling**,  
     bindings,  
         conflict issues; 11-7  
         term definition; 7-7  
     multiset,  
         term definition; 7-3  
     transitions,  
         by binding input arc inscription variables; 7-7

**Enter Editor (File menu)**,  
     leaving the simulator with; 8-19

**Enter Order transition (FirstModel)**,  
     operations performed by; 13-8

**Enter Simulator (File menu)**,  
     entering the simulator with; 8-7

**entering**,  
     *See Also* leaving;  
     simulator,  
         with Enter Simulator (File menu); 8-7  
     text; 4-21

**entities**,  
     *See Also* modeling; representation;  
     represented in the Sales Order Model; B1-2  
     representing with colorsets; 10-4

**enumerated colorsets**,  
     term definition and characteristics; 5-4

**environment,**

- graphical,
  - setting; 6-2
  - term definition; 6-2
- tutorial,
  - establishing; 6-3

**equal (=),**

- boolean operator used in guards; 7-8

**equipment,**

- FirstModel,
  - characteristics; 13-2
- Sales Order Model,
  - characteristics; B1-2
  - simulation parameters; B1-5

**errors,**

- error box,
  - interpreting; 9-4
  - term definition; 9-3
- locating,
  - with text pointers; 9-4
- missing colorset,
  - fixing; 9-5
- ML,
  - detecting; 9-9
- syntax,
  - detecting, with the Syntax Check command (CPN menu); 8-2
  - handling, (chapter); 9-1
  - missing colorset; 9-2
  - undeclared CPN variables; 9-6
- undeclared CPN variables,
  - fixing; 9-6

**establishing,**

- See Also* creating; specifying;
- initial markings,
  - for SalesNet model execution with conflict; 11-15
- tutorial environment; 6-3

**evaluating,**

- input arc inscriptions,
  - during SalesNet execution; 10-9
  - during transition firing; 7-13
- output arc inscriptions,
  - during SalesNet execution; 10-10
  - during transition firing; 7-13

**executing,**

- binding elements; 11-16
- CP nets; 7-1, 7-14, 8-10
  - (chapter); 8-1
- FirstNet model, concurrent execution; 11-4
- order of execution events; 8-16
- overview; 5-16
- SalesNet, interactive execution; 10-12

**executing (cont'd),**

- CP nets (cont'd),
  - SalesNet, simulator actions; 10-9
  - starting execution, with Interactive Run command (Sim menu); 8-14
  - with fusion sets; A2-19
  - with instance fusion sets; A2-25
- models,
  - automatic mode; 15-20
  - FirstModel (chapter); 15-1
  - interactive vs. fast mode; 15-15
- occurrence sets,
  - algorithm for; 11-16
  - elements; 11-16
- Resource Use Model; A2-4
- SalesNet model with conflict; 11-15
- timed CP nets; 16-16

**execution,**

- concurrent,
  - SalesNet model; 11-7
- modes,
  - alternating between automatic and interactive; 15-21
  - automatic, executing a model with; 15-21
  - fair automatic compared with fast automatic mode; 15-16
  - specifying, actual; 15-18
  - specifying, possible; 15-17
- specifying stop criteria for; 15-19
- states,
  - loading; 15-23
  - saving; 15-22
  - starting with a saved; 15-24

**experimenting,**

- See Also* modeling;
- with Sales Order Model improvements; B3-6

**expressions,**

- boolean,
  - used in guards; 7-8
- delay,
  - syntax and characteristics; 16-7

**extending,**

- CP nets,
  - FirstNet into SalesNet (chapter); 10-1

---

## F

**fair automatic mode,**

term definition and comparison with fast automatic mode; 15-16

**fast automatic mode,**

term definition and comparison with fair automatic mode; 15-16

**fax number,**

Meta Software; 1-7

**feedback region,**

term definition; 8-9

**File menu,**

Close command,

closing diagrams with; 2-9

Enter Editor command,

leaving the simulator with; 8-19

Enter Simulator command,

entering the simulator with; 8-7

Load State command,

saving an execution state with; 15-23

Load Subdiagram command,

loading a subdiagram with; A2-12

New command,

creating a CP net with; 6-5

creating new diagrams with; 4-6

Open command,

opening a diagram with; 2-5

opening CP nets with; 8-1

starting the Sales Order Model with; B2-3

Quit command,

exiting Design/CPN Sales Order Model

experiments with; B3-14

quitting Design/CPN with; 2-10

Save As command,

saving CP nets with; 6-17

Save State command,

saving an execution state with; 15-22

Save Subdiagram command,

saving a subdiagram with; A2-12

**files,**

Design/CPN files,

characteristics, contents, and use; 2-3

Sales Order Model; B1-1

simulation parameters,

Sales Order Model, contents and use; B2-1

**fire,**

term definition; 7-12

**firing,**

concurrent transitions; 11-3

**firing (cont'd),**

simulation region identifying; 8-9

**FirstModel model,**

characteristics and components; 13-1

concurrency; 13-18

entering an order; 13-17

operations; 13-7

processing an order; 13-17

SalesNet compared with; 13-2

shipping an order; 13-17

structure of; 13-3

summary of operations; 13-16

superpage,

compared with SalesNet; 13-5

**FirstNet model,**

concurrent execution of; 11-4

extending,

into SalesNet (chapter); 10-1

**FirstNetDemo model,**

breakpoints,

beginning of substep; 8-15

end of substep; 8-16

**Fusion Place (CPN menu),**

adding places to a fusion set with; A2-9

creating,

a fusion set with; A2-6

instance fusion sets with; A2-24

multi-page fusion sets with; A2-14

multiple fusion sets with; A2-15

page fusion sets with; A2-18

deleting,

fusion sets with; A2-11

places from a fusion set with; A2-10

**fusion,**

key region,

term definition and illustration; A2-8

places,

(chapter); A2-1

characteristics; A1-2

global, physical appearance; A2-8

global, term definition; A2-8

instance, term definition; A2-24

multiple pages; A2-11

page, term definition; A2-17

ports as a type of; 12-8

single page; A2-4

sockets as a type of; 12-8

term definition; 12-1, A1-2, A2-1

region,

term definition; A2-8

sets,

adding places to; A2-9

## **fusion (cont'd),**

- sets (cont'd),
  - deleting with Fusion Place (CPN menu); A2-11
  - executing CP nets with; A2-19
  - global, creating; A2-6
  - global, term definition; A2-7
  - initial markings and; A2-10
  - instance, creating; A2-20
  - instance, executing CP nets with; A2-25
  - multiple, working with; A2-14
  - multiplicity and; A2-23
  - page, creating; A2-17
  - page, term definition; A2-17
  - page-spanning, working with; A2-13
  - removing places from; A2-10
  - term definition; A1-2, A2-1
- subsets,
  - instance, term definition; A2-24
  - page, term definition; A2-17

## **G**

## **General Simulation Options (Set menu),**

- simulating with and without; 16-18
- specifying actual executions modes with; 15-18

## **generating,**

- See Also* creating;
- output multisets,
  - during SalesNet execution; 10-11

## **global,**

- declaration node,
  - adding declarations to extend FirstNet into SalesNet; 10-2
  - creating; 6-15
  - creation mode, term definition; 6-15
  - declaring a timed colorset in; 16-7
  - FirstModel; 13-3
  - moving to a new page; 10-20
  - term definition; 5-3
- declarations,
  - converting SalesNet into FirstModel; 14-1
- fusion place,
  - physical appearance; A2-8
  - term definition; A2-8
- fusion set,
  - term definition; A2-7

## **graphical objects,**

- See Also* ellipses;
- characteristics; 4-2
- creating; 4-4
  - from text mode; 4-17

## **graphical objects (cont'd),**

- deleting; 4-14
- editing; 4-3
- multiple,
  - working with different types; 4-17
  - working with; 4-13
- selecting; 4-14
- term definition; 4-1

## **graphics,**

- auxiliary compared with CPN; 6-1
- Design/CPN editor,
  - (chapter); 4-1
- graphical environment,
  - term definition; 6-2
- graphics mode,
  - characteristics; 4-3
  - term definition; 4-3
- graphics tool,
  - term definition; 4-3

## **greater than (>),**

- boolean operator used in guards; 7-8

## **greater than or equal (>=),**

- boolean operator used in guards; 7-8

## **Group menu,**

- Regroup command,
  - reconstructing groups with; 4-27
- selecting groups with; 4-26
- Ungroup command,
  - deselecting groups with; 4-27

## **groups,**

- deselecting; 4-27
- group mode,
  - term definition; 4-25
- group tool,
  - characteristics and illustration; 4-25
- mixed,
  - restrictions; 4-26
- operations on; 4-26
- reconstructing; 4-27
- selecting; 4-26
- term definition; 4-25

## **guards,**

- See Also* input arc inscriptions;
- changing rules with,
  - FirstModel; 15-8
- characteristics,
  - and term definition; 5-16
  - as CP net component; 5-1
- creating; 6-8
  - for FirstModel; 14-29
- guard region creation mode,
  - term definition; 6-8



**guards (cont'd),**

- modifying to extend FirstNet into SalesNet; 10-2
- representing constraints and conditions with; 10-6
- role in enabling transitions; 7-3
- syntax; 7-8
- term definition; 3-4, 7-3

**H****handles,**

- See Also* arcs; connectors;
- term definition; 4-9

**handling,**

- syntax errors,
- (chapter); 9-1

**hierarchy,**

- See Also* fusion; substitution transitions;
- hierarchical CP nets,
  - developing on a subpage; A3-21
  - relationship among pages in; A3-24
  - term definition; 1-2
- hierarchical decomposition,
  - substitution transitions, introduction (chapter); 12-1
  - term definition; 12-2
- introduction,
  - (chapter); A1-1
- key region,
  - moving with Drag command (Makeup menu); 12-6
  - term definition; 12-5, A3-10
- page,
  - changes when a substitution transition is created; 12-11
  - deleting a page from; A3-25
  - error box location on; 9-3
  - for a hierarchical CP net diagram; A3-2
  - hierarchical CP net for top-down net development; A3-23
  - improving appearance; 12-12
  - improving appearance; A3-16
  - improving the appearance of; 10-19
  - term definition and characteristics; 2-7
- region,
  - term definition; 12-5, A3-10
  - term definition; A1-1

**Horizontal (Align menu),**

- aligning nodes in a row with; 14-17

**horizontal spread,**

- aligning nodes along,
- with Horizontal Spread command (Align menu); 14-16

**Horizontal Spread (Align menu),**

- aligning nodes with; 14-16

**I****IDEF0 modeling paradigm,**

- characteristics; 3-2
- compared with CP net modeling; 3-4

**incremental net development,**

- prime page role; 8-4

**initial markings,**

- See Also* places;
- changing,
  - to add more equipment to the SalesNet model; 11-8
- establishing,
  - for SalesNet model execution with conflict; 11-15
- fusion sets and; A2-10
- location of in a hierarchical net,
  - modeling considerations; 15-4
- region,
  - characteristics and term definition; 5-12
  - initializing with Initial State command (Sim menu); 11-13
- specifying; 6-11
- term definition; 5-12
- time stamps and; 16-10

**initial state,**

- term definition; 5-12

**Initial State (Sim menu),**

- experimenting with changed simulation parameters
  - for the Sales Order Model; B3-8
- initializing SalesNet after changing in the simulator; 11-13
- initializing state after changing initial markings in FirstModel; 15-5
- initializing the CP net state with; 8-13

**initializing,**

- CP net state with Initial State (Sim menu); 8-13
- initial marking region,
  - with Initial State command (Sim menu); 11-13
- Sales Order Model; B3-8
- SalesNet after changing in the simulator; 11-13
- state after changing initial markings in FirstModel; 15-5

**input,**

- arcs,
  - term definition; 5-14
- places,
  - creating; 6-9

**input (cont'd),**

- places (cont'd),
  - removing enabling multisets from, during transition firing; 7-13
  - term definition; 5-14
- token key region,
  - term definition; 8-15
- token region,
  - term definition; 8-15
- tokens,
  - term definition; 8-15

**input arc inscriptions,**

- See Also* arcs; guards; inscriptions;
- binding variables in; 7-7
- characteristics as CP net component; 5-1
- constants used as; 7-4
- evaluating,
  - during SalesNet execution; 10-9
  - during transition firing; 7-13
- role in enabling transitions; 7-2
- term definition; 3-4, 5-15

**inscriptions,**

- arc,
  - characteristics and term definition; 5-15
  - creating; 6-13
  - creating, for FirstModel; 14-24
  - detecting errors in; 9-9
- input arc,
  - binding variables in; 7-7
  - constants used as; 7-4
  - evaluating during SalesNet execution; 10-9
  - evaluating during transition firing; 7-13
  - role in enabling transitions; 7-2
  - term definition; 3-4
- output arc,
  - delay expressions on; 16-8
  - evaluating during SalesNet execution; 10-10
  - evaluating during transition firing; 7-13
  - term definition; 3-4

**Instance Switch dialog,**

- switching among page instances with; A3-35

**instance,**

- fusion place,
  - term definition; A2-24
- fusion sets,
  - creating; A2-20
- fusion subsets,
  - term definition; A2-24
- page,
  - creating; A2-21
  - switching among, with the Instance Switch dialog; A3-35

**instance (cont'd),**

- page (cont'd),
  - term definition; A2-20

**integer colorsets,**

- characteristics and term definition; 5-6

**interactive mode,**

- term definition and characteristics; 15-16

**Interactive Run (Sim menu),**

- constructing an occurrence set with; 11-15
- executing,
  - CP nets with; 8-10
  - FirstModel with; 15-3
  - SalesNet with; 10-13
- starting execution with; 8-14

**Interactive Simulation Options (Set menu),**

- setting breakpoints; 8-12
  - for FirstModel; 15-3
  - for SalesNet; 10-13

**interpreting,**

- Sales Order Model run results; B3-1

## J

**Jensen, Kurt,**

- bibliographic reference; 1-1

**job,**

- stream (Sales Order Model),
  - simulation parameters; B1-4
- value (Sales Order Model),
  - simulation parameters; B1-4

## K

**key region,**

- term definition and characteristics; 10-15

**keyboard,**

- Design/CPN use of; 2-4

**keystroke,**

- shortcuts; 4-5

## L

**Label (Aux menu),**

- creating rectangles with; 4-21

**labels,**

- creating; 4-21
- differences between graphical objects and; 4-22

**labels (cont'd),**  
 label creation mode,  
   characteristics; 4-21  
 label tool,  
   characteristics and illustration; 4-21  
 term definition; 4-2

**leaving,**  
*See Also* entering;  
 simulator,  
   with Enter Editor command (File menu); 8-19

**less than (<),**  
 boolean operator used in guards; 7-8

**less than or equal (<=),**  
 boolean operator used in guards; 7-8

**literals,**  
 as exact token values; 7-4

**Load State (File menu),**  
 saving an execution state with; 15-23

**Load Subdiagram (File menu),**  
 loading a subdiagram with; A2-12

**loading,**  
*See Also* saving;  
 execution states; 15-23  
 subdiagrams; A2-12

**locality,**  
*See Also* concurrency; modeling;  
 CP net,  
   term definition; 13-19

**locating,**  
 errors,  
   with text pointers; 9-4  
 errors; 9-3

**locations,**  
 as dynamic modeling paradigm component; 3-3  
 CP net places as representations for; 3-3  
 representing with places; 10-5

## M

**macros,**  
 subpages compared to; A3-24, A3-34

**Make Region (Aux menu),**  
 creating regions with; 4-23

**Makeup menu,**  
 Change Shape command,  
   changing transition shape with; A3-28  
   matching transition sizes with; 14-10  
 Child Object command,  
   improving the appearance of a substitution  
     transition with; 12-6  
   selecting a region with; A3-13

**Makeup menu (cont'd),**  
 Drag command,  
   moving a region with; A3-13  
   moving hierarchy key region with; 12-6

**mapping,**  
 inputs to output,  
   role of the simulator in; 11-3

**markings,**  
 appearance of; 5-13  
 characteristics and term definition; 5-12  
 current marking key region,  
   term definition; 8-9  
 current marking region,  
   term definition; 8-9  
 current simulation regions,  
   describing; 8-9  
 initial marking region,  
   characteristics and term definition; 5-12  
 initial,  
   fusion sets and; A2-10  
   specifying; 6-11  
   term definition; 5-12

**mathematical introduction,**  
 Petri nets,  
   bibliographic reference; 1-1

**menu bar,**  
 characteristics; 2-6

**Meta Software,**  
 addresses and telephone numbers; 1-7

**ML Configuration Options (Set menu),**  
 loading ML configuration information with; 8-2  
 loading SalesNet configuration information with;  
   11-8  
 preserving ML configuration options with; 6-4  
 setting options for the Resource Use Model; A2-4

**ML language,**  
 configuration information,  
   loading; 8-2  
   loading for SalesNet model; 11-8  
 configuration options,  
   preserving; 6-4  
 errors,  
   detecting; 9-9  
 file,  
   term definition; 2-3  
 process,  
   term definition; 2-3  
 role in CP nets; 5-1  
 window,  
   term definition and illustration; 8-3

**Mode Attributes (Set menu),**  
 creating multiple page instances with; A2-21

**Mode Attributes (Set menu) (cont'd),**

designating prime pages with; 8-5  
specifying prime pages with; A2-13

**model time,**

term definition; 16-5

**modeling,**

*See Also* behavior; choice; representation;  
activities,

with transitions; 10-6

analyzing FirstModel execution; 15-3

under heavy load; 15-7

with different occurrence set parameter settings;  
15-14

with partial constraints; 15-9

behavior resulting from net execution; 13-20

concurrency issues; 15-11

conditions,

with guards; 10-6

documenting the model through comments; 14-2

effective space use,

creating a separate page for global declarations;  
10-17

improving the appearance of the hierarchy page;  
10-19

entities,

with colorsets; 10-4

experimenting with a model,

guidelines for; 15-6

experimenting with Sales Order Model

improvements; B3-6

extending a small net into a high-level model,

(chapter); 10-1

initial marking location considerations; 15-4

interpreting the results of Sales Order Model run;  
B3-1

locality in CP nets; 13-19

locations,

with places; 10-5

mutual independence of transitions in CP nets; 13-19

occurrence set construction issues; 15-10

paradigms,

(chapter); 3-1

dynamic, term definition and characteristics; 3-3

static, term definition and characteristics; 3-1

term definition; 3-1

power of locality for real-world representation; 13-19

representation,

SalesNet model relationship to the system it  
represents; 10-4

requirements for skill in; 1-6

rules,

with guards; 10-6

**modeling (cont'd),**

specifying how and when to stop; 15-19

submodels; 15-4

understanding a simple model (chapter); 13-1

**models,**

*See Also* FirstModel model; FirstNet model;

FirstNetDemo model; Resource Use model;

Sales Order model; SalesNet model;

building a simple (chapter); 14-1

CP nets compared with; 5-3

executing,

automatic mode; 15-20

FirstModel (chapter); 15-1

interactive vs. fast mode; 15-15

**modes,**

execution,

alternating between automatic and interactive;  
15-21

specifying, actual; 15-18

specifying, possible; 15-17

graphics,

term definition; 4-3

graphics editor,

term definition; 4-3

text,

term definition; 4-3

**modularity,**

*See Also* hierarchy;

CP nets,

prime page role; 8-4

**mouse,**

Design/CPN use of; 2-4

**Move to Subpage (CPN menu),**

creating subpages with; 12-3, A3-9

top-down hierarchical CP net development with;  
A3-22

**moving,**

*See Also* navigating;

CP net components between pages; 10-20

hierarchy key region,

with Drag command (Makeup menu); 12-6

parents; 4-24

rectangles; 4-9

during creation; 4-10

regions; A3-13

**multiplicity,**

fusion and; A2-23

substitution transitions and; A3-33

term definition; A2-20

**multiprocessing,**

Design/CPN use of; 2-3

**multisets,**  
 adding; 5-8  
 designator,  
   term definition and characteristics; 5-8  
 empty,  
   term definition; 5-7  
 enabling,  
   removing during SalesNet execution; 10-10  
 multiset creation operator ({}),  
   creating multisets with; 5-8  
 output,  
   generating during SalesNet execution; 10-11  
   putting into output place during transition firing;  
     7-14  
   term definition; 7-13  
 regions,  
   term definition; 8-15  
 removing enabling from each input place,  
   during transition firing; 7-13  
 specifying; 5-8  
 subsetting; 5-9  
 subtracting; 5-9  
 term definition and characteristics; 5-7  
 time stamps and; 16-11

## N

**naming,**  
 name region creation mode,  
   term definition; 6-7  
 pages,  
   with Page Attributes command (Set menu); 10-18  
 places; 6-10  
   for FirstModel subpage; 14-12  
 substitution transitions; 12-5, A3-10  
 transitions; 6-7  
**navigating,**  
*See Also* moving;  
 diagrams; 2-7  
 from subpage to superpage,  
   by double-clicking on a port; A3-6  
 to an error with cursor keys; 9-4  
**New (File menu),**  
   creating a CP net with; 6-5  
   creating new diagrams with; 4-6  
**New Page (Page menu),**  
   creating a page for global declarations with; 10-18  
**nodes,**  
   *See Also* regions;

**nodes (cont'd),**  
 aligning,  
   along a diagonal, with Align menu commands;  
     14-16  
   with Align menu commands; 14-16  
 CPN,  
   term definition; 6-2  
 global declaration,  
   adding declarations to extend FirstNet into  
     SalesNet; 10-2  
   creating; 6-15  
   declaring a timed colorset in; 16-7  
 page,  
   term definition and characteristics; 2-8  
 reference,  
   term definition; 14-16  
 regions and; 4-22  
 term definition; 4-2, 4-18  
**not (boolean NOT),**  
   boolean operator used in guards; 7-8  
**not equal (<>),**  
   boolean operator used in guards; 7-8

## O

**object attributes,**  
   term definition and characteristics; 6-2  
**objects,**  
   CPN compared with auxiliary; 4-2  
   data,  
**occlusion order,**  
   term definition; 4-14  
**occur,**  
   term definition; 7-2  
**occurrence,**  
   what happens when a transition occurs; 7-11  
**occurrence sets,**  
   *See Also* algorithms; bindings;  
   characteristics; 15-10  
   concurrency in relation to; 15-11  
   constructing; 15-10  
     for SalesNet model execution with conflict; 11-15  
   controlling the appearance of concurrency with;  
     15-10  
   executing,  
     algorithm for; 11-16  
     the elements in; 11-16  
   parameters controlling their construction,  
     scope of; 15-14  
     setting; 15-14  
   term definition; 15-11

**occurrence sets (cont'd),**

- setting parameters,
  - with Occurrence Set Options (Set menu); 15-12
- term definition; 11-14, 15-10

**Occurrence Set Options (Set menu),**

- setting occurrence set parameters with; 15-12

**omitting,**

- See Also* deleting;
- time stamp; 16-9

**Open (File menu),**

- opening a diagram with; 2-5
- opening CP nets with; 8-1
- starting the Sales Order Model with; B2-3

**open page,**

- term definition; 2-7

**Open Page (Page menu),**

- accessing hierarchy page with; 10-19
- examining bar chart page with; B3-3
- opening pages with; 2-7
- selecting a page with; 8-5

**opening,**

- CP nets,
  - with Open (File menu); 8-1
- diagram; 2-5
- pages,
  - with Open Page (Page menu); 2-7, 8-5

**operators,**

- See Also* expressions;
- backquote (`); 5-8
- boolean,
  - used in guards; 7-8
- comparison,
  - used in guards; 7-8

**order processing,**

- FirstModel,
  - characteristics; 13-2

**orders,**

- FirstModel,
  - characteristics; 13-1

**or else (boolean OR),**

- boolean operator used in guards; 7-8

**output,**

- arcs,
  - term definition; 5-14
- multisets,
  - generating during SalesNet execution; 10-11
  - putting into output place during transition firing;  
7-14
  - term definition; 7-13
- places,
  - creating; 6-13

**output (cont'd),**

- places (cont'd),
  - putting the output multiset into during transition  
firing; 7-14
  - term definition; 5-14
- representing with output arcs and output arc  
inscriptions; 10-8
- token key region,
  - term definition; 8-15
- token region,
  - term definition; 8-15
- tokens,
  - term definition; 8-15

**output arc inscriptions,**

- characteristics as CP net component; 5-1
- delay expressions on; 16-8
- evaluating,
  - during SalesNet execution; 10-10
  - during transition firing; 7-13
- specifying output with; 10-8
- term definition; 3-4, 5-15

## P

**page,**

- mode key region,
  - term definition; 8-6
- mode region,
  - term definition; 8-6
- node,
  - term definition and characteristics; 2-8
- numbers,
  - printing use of; 2-9

**Page Attributes (Set menu),**

- Page Height component,
  - page border specified by; 2-6
- Page Width component,
  - page border specified by; 2-6
- renaming pages with; 10-20
- renaming pages with; 12-13

**page border,**

- term definition and characteristics; 2-6

**Page Height component - Page Attributes  
(Set menu),**

- page border specified by; 2-6

**page instances,**

- multiple,
  - setting their representation in an occurrence set;  
15-12

## **Page menu,**

- New Page command,
  - creating a page for global declarations with; 10-18
- Open Page command,
  - accessing hierarchy page with; 10-19
  - examining bar chart page with; B3-3
  - opening pages with; 2-7
  - selecting a page; 8-5
- Redraw Hierarchy command,
  - redrawing the hierarchy page with; 12-12, A3-17

## **Page Width component - Page Attributes (Set menu),**

- page border specified by; 2-6

## **pages,**

- closed,
  - term definition; 2-7
- creating,
  - for global declarations; 10-17
- decomposition,
  - creating, Design/CPN actions; 12-9
  - simplifying; 12-10
- deleting; A3-25
- fusion place,
  - term definition; A2-17
- fusion sets,
  - creating; A2-17
  - term definition; A2-17
- fusion subsets,
  - term definition; A2-17
- hierarchy,
  - for a hierarchical CP net diagram; A3-2
  - hierarchical CP net for top-down net development; A3-23
  - improving the appearance of; 10-19, 12-12, A3-16
  - term definition and characteristics; 2-7
- instances,
  - creating; A2-21
  - term definition; A2-20
- moving CP net components between; 10-20
- multiple,
  - distributing a CP net across; 12-1
  - setting their representation in an occurrence set; 15-12
- naming,
  - with Page Attributes command (Set menu); 10-18
- open,
  - term definition; 2-7
- opening,
  - with Open Page (Page menu); 2-7, 8-5
- prime,
  - designating; 8-4

## **pages (cont'd),**

- prime (cont'd),
  - designating with Mode Attributes (Set menu); 8-5
  - specifying; A2-13
  - term definition; 8-4
- relationship among in a hierarchical CP net; A3-24
- renaming; 12-13
  - from the hierarchy page; 10-20
- term definition and characteristics; 2-6

## **parameters,**

- occurrence set,
  - setting with Occurrence Set Options (Set menu); 15-12
- simulation,
  - changing, for Sales Order Model; B3-6
  - Sales Order Model; B1-4
  - Sales Order Model, contents and use of the file containing; B2-1

## **parentheses (()),**

- guards use of; 7-8

## **parents,**

- See Also* regions;
- deleting; 4-25
- moving; 4-24

## **pasting,**

- text regions; 14-26

## **patterns,**

- tuple,
  - characteristics; 13-12
  - term definition; 13-12

## **performance,**

- fast model execution vs. interactive model execution; 15-15

## **performing,**

- See Also* executing;
- syntax check; 8-2

## **Petri nets,**

- See Also* CP nets; Design/CPN;
- characteristics and use; 1-1
- hierarchical colored,
  - term definition; 1-2
- history; 1-1
- mathematical introduction,
  - bibliographic reference; 1-1
- term definition; 1-1

## **Place (CPN menu),**

- creating places with; 6-9

## **place creation mode,**

- term definition; 6-9

## **place tool,**

- term definition and illustration; 6-9

## **places,**

*See Also* colorsets; ellipses; initial markings;  
adding to fusion sets; A2-9  
as CP net locations; 3-3  
characteristics and term definition; 5-11  
characteristics as CP net component; 5-1  
creating,  
    for FirstModel subpage; 14-12  
current marking simulation regions,  
    characteristics; 8-9  
fusion,  
    multiple pages; A2-11  
    page, term definition; A2-17  
    ports as a type of; 12-8  
    sockets as a type of; 12-8  
global fusion,  
    physical appearance; A2-8  
    term definition; A2-8  
input,  
    creating; 6-9  
    removing enabling multisets from, during  
        transition firing; 7-13  
markings,  
    characteristics and term definition; 5-12  
naming; 6-10  
    for FirstModel subpage; 14-12  
output,  
    creating; 6-13  
    putting the output multiset into, during transition  
        firing; 7-14  
removing from fusion sets; A2-10  
representing locations with; 10-5

## **pointer tool,**

characteristics and illustration; 4-23

## **popup regions,**

term definition and characteristics; 10-15

## **Port Assignment (CPN menu),**

manually assigning ports to sockets with; A3-39

## **ports,**

as a type of fusion place; 12-8  
key region,  
    term definition; 12-9  
manually assigning to sockets; A3-37  
navigating from a subpage to a superpage by double-  
    clicking on; A3-6  
port key region,  
    term definition; A3-5  
port region,  
    term definition; A3-5  
rearranging,  
    for FirstModel; 14-7

## **ports (cont'd),**

region,  
    term definition; 12-9  
relationship to sockets; 14-5  
socket relationship to; A3-6  
subpage,  
    FirstModel appearance before modification; 14-4  
term definition; 12-8, A3-1

## **preserving,**

*See Also* aligning;  
aspect ratio; 4-12

## **prime pages,**

*See Also* occurrence sets; simulation;  
designating; 8-4  
    with Mode Attributes (Set menu); 8-5  
specifying; A2-13  
term definition; 8-4

## **printing,**

diagrams; 2-9

## **Process Order transition (FirstModel),**

enablement of; 13-13  
firing; 13-14  
operations performed by; 13-10

## **process,**

modeling; 3-1

## **products,**

FirstModel,  
    characteristics; 13-2

# Q

## **Quit (File menu),**

exiting Design/CPN Sales Order Model experiments  
    with; B3-14  
quitting Design/CPN with; 2-10

## **quitting,**

*See Also* entering;  
Design/CPN; 2-10

# R

## **rebinding,**

*See Also* bindings; CPN variables;  
CPN variables,  
    during transition firing; 7-12

## **reconstructing,**

groups; 4-27

## **rectangles,**

*See Also* transitions;



**rectangles (cont'd),**

- adding text to; 4-11
- creating; 4-7
  - a series of; 4-11
  - adding text while; 4-12
- deleting; 4-10
- moving,
  - during creation; 4-10
- moving; 4-9
- preserving the aspect ratio during size change; 4-12
- rectangle creation mode,
  - characteristics; 4-7
- rectangle tool,
  - characteristics and illustration; 4-7
- reshaping; 4-9

**Redraw Hierarchy (Page menu),**

- redrawing the hierarchy page with; 12-12, A3-17

**reference nodes,**

- term definition; 14-16

**region,**

- key,
  - term definition and characteristics; 10-15

**regions,**

- See Also* nodes;
- CPN,
  - term definition; 6-2
- creating; 4-23
- current marking key,
  - term definition; 8-9
- current marking,
  - term definition; 8-9
- editing; 4-24
- feedback,
  - term definition; 8-9
- fusion key,
  - term definition and illustration; A2-8
- hierarchy key,
  - moving with Drag command (Makeup menu); 12-6
  - term definition; A3-10
- hierarchy,
  - term definition; A3-10
- initial marking,
  - characteristics and term definition; 5-12
  - initializing with Initial State command (Sim menu); 11-13
- moving; A3-13
  - the parent of; 4-24
- popup,
  - term definition and characteristics; 10-15
- port,
  - term definition; 12-9, A3-5

**regions (cont'd),**

- port key,
  - term definition; 12-9, A3-5
- region tool,
  - term definition and illustration; 6-7
- restoring the independence of a; 4-24
- selecting; A3-13
- simulation,
  - characteristics and purpose; 8-8
  - current marking; 8-9
  - feedback; 8-9
  - removing with Remove Sim Regions command (CPN menu); 8-20
  - term definition; 8-9
  - transition feedback; 8-9
- substitution tag,
  - term definition; 12-12
- term definition and characteristics; 4-22
- text,
  - copying; 14-26
  - pasting; 14-26
- time,
  - term definition; 16-8

**Regroup (Group menu),**

- reconstructing groups with; 4-27

**Remove Sim Regions (CPN menu),**

- cleaning up diagrams during hierarchy simulation; A3-35
- removing simulation regions with; 8-20

**removing,**

- See Also* creating;
- enabling multisets from each input place,
  - during SalesNet execution; 10-10
  - during transition firing; 7-13
- places from fusion sets; A2-10
- simulation regions,
  - with Remove Sim Regions command (CPN menu); 8-20

**renaming,**

- pages; 12-13
  - from the hierarchy page; 10-20

**Replace by Subpage (CPN menu),**

- reversing substitution transition creation with; A3-19

**representation,**

- activities,
  - modeling with transitions; 10-6
- causality,
  - in FirstModel; 16-4
- concurrency; 11-2
- conditions,
  - modeling with guards; 10-6

**representation (cont'd),**

- conflict; 11-5
- entities,
  - modeling with colorsets; 10-4
- locations,
  - modeling with places; 10-5
- modeling,
  - SalesNet model relationship to the system it represents; 10-4
- modeling paradigms,
  - dynamic, term definition and characteristics; 3-3
  - static, term definition and characteristics; 3-1
- rules,
  - modeling with guards; 10-6
- Sales Order Model entities; B1-2
- state,
  - transition use as; 11-3
- time,
  - simulated (chapter); 16-1

**representing,**

- data output with output arcs and output arc inscriptions; 10-8

**rerouting,**

- See Also* aligning;
- arcs; A3-12

**resetting,**

- drawing environment; 4-6

**reshaping,**

- See Also* aligning; creating;
- rectangles; 4-9

**Resource Use Model,**

- characteristics and files; A2-1
- description; A2-3
- executing; A2-4
- hierarchy page; A3-2
- overall structure; A3-6
- subpage; A3-4
- superpage; A3-3

**resources,**

- allocation of; 11-1
- competition for,
  - as concurrency problem; 11-1
- modeling with FirstModel; 15-7

**restoring,**

- drawing environment; 4-6

**Reswitch (Sim menu),**

- reswitching after changing initial markings in FirstModel; 15-5
- reswitching SalesNet after changing in the simulator; 11-13

**reswitching,**

- after changing a CP net in the simulator; 11-13

**reversing,**

- substitution transition creation; A3-18

**routing,**

- See Also* aligning;
- connectors; 4-19
  - automatic; 4-20

**row,**

- aligning nodes in,
  - with Horizontal command (Align menu); 14-17

**rules,**

- as source of system inefficiency,
  - detecting through simulation; B3-14
- representing with guards; 10-6

**running,**

- See Also* executing;
- Sales Order Model; B2-4

## S

**Sales Order Model,**

- detecting inefficiency in the system described by;
  - B1-3
- efficiency charts,
  - examining; B3-3
- experimenting with changed simulation parameters;
  - B3-8
- inefficiency in the system described by; B1-3
- interpreting and using (chapter); B3-1
- introduction to (chapter); B1-1
- overview; B1-1
- revenue charts,
  - examining; B3-1
- running; B2-4
  - (chapter); B2-1
- starting,
  - with Open (File menu); B2-3

**SalesNet model,**

- changing in the simulator; 11-12
- concurrent execution of; 11-7
- executing with conflict; 11-15
- extending,
  - with substitution transitions; 12-1
- FirstModel compared with; 13-2
- Order Processing System modeled by; 10-3
- superpage,
  - compared with FirstModel; 13-5

**satisfy,**

- term definition; 7-10

**Save As (File menu),**

- saving CP nets with; 6-17

## **Save State (File menu),**

saving an execution state with; 15-22

## **Save Subdiagram (File menu),**

saving a subdiagram with; A2-12

## **saving,**

*See Also* loading;

CP nets; 6-17

diagram changes,

with Save Changes dialog (Close command); 2-9

diagrams; 6-17

execution states; 15-22

subdiagrams; A2-12

## **scrolling,**

autoscrolling characteristics; 4-5

## **segments,**

term definition; 4-19

## **selecting,**

graphical objects; 4-14

groups; 4-26

pages,

with Open Page (Page menu); 8-5

regions; A3-13

## **Set menu,**

Copy Defaults command,

copying diagram defaults with; 6-4

General Simulation Options command,

simulating with and without; 16-18

specifying actual executions modes with; 15-18

Interactive Simulation Options command,

setting breakpoints with; 8-12

setting FirstModel breakpoints with; 15-3

setting SalesNet breakpoints with; 10-13

ML Configuration Options command,

loading ML configuration information with; 8-2

loading SalesNet configuration information with;  
11-8

preserving ML configuration options with; 6-4

setting options for the Resource Use Model; A2-4

Mode Attributes command,

creating multiple page instances with; A2-21

designating prime pages with; 8-5

specifying prime pages with; A2-13

Occurrence Set Options command,

setting occurrence set parameters with; 15-12

Page Attributes command,

Page Height component, page border specified by;  
2-6

Page Width component, page border specified by;  
2-6

renaming pages with; 10-20, 12-13

Page Height component - Page Attributes command,

page border specified by; 2-6

## **Set menu (cont'd),**

Page Width component - Page Attributes command,  
page border specified by; 2-6

Shape Attributes command,

specifying bidirectional arcs with; 14-20

Simulation Code Options command,

compiling timed CP nets with; 16-14

specifying possible execution modes with; 15-17

## **sets,**

*See* multisets;

## **setting,**

breakpoints,

for FirstModel; 15-3

for SalesNet execution; 10-13

with Interactive Simulation Options command (Set  
menu); 8-12

graphical environment; 6-2

occurrence set parameters,

with Occurrence Set Options (Set menu); 15-12

## **shape,**

transition,

changing with Change Shape command (Makeup  
menu); A3-28

## **Shape Attributes (Set menu),**

specifying bidirectional arcs with; 14-20

## **Ship Product transition (FirstModel),**

operations performed by; 13-15

## **shortcuts,**

keystroke; 4-5

## **Sim menu,**

Automatic Run command,

executing a model with; 15-21

running the Sales Order Model with; B2-4

characteristics and purpose; 8-10

Continue command,

continuing execution after a breakpoint with;  
8-16, 11-18

Initial State command,

experimenting with changed simulation

parameters for the Sales Order Model; B3-8

initializing SalesNet after changing in the  
simulator; 11-13

initializing state after changing initial markings  
in FirstModel; 15-5

initializing the CP net state with; 8-13

Interactive Run command,

constructing an occurrence set with; 11-15

executing CP nets with; 8-10

executing FirstModel with; 15-3

executing SalesNet with; 10-13

starting execution with; 8-14

## **Sim menu (cont'd),**

- Reswitch command,
  - reswitching FirstModel after changing initial markings; 15-5
  - reswitching SalesNet after changing in the simulator; 11-13
- Stop command,
  - canceling CP net execution with; 8-18

## **simplifying,**

- See Also* aligning;
- decomposition page; 12-10

## **Simulation Code Options (Set menu),**

- compiling timed CP nets with; 16-14
- specifying possible execution modes with; 15-17

## **simulation,**

- See Also* breakpoints;
- interpreting the results of Sales Order Model run;
  - B3-1
- parameters,
  - changing, for Sales Order Model; B3-6
  - Sales Order Model; B1-4
- regions,
  - adjusting for SalesNet execution; 10-13
  - characteristics and purpose; 8-8
  - current marking; 8-9
  - feedback; 8-9
  - removing with Remove Sim Regions command (CPN menu); 8-20
  - term definition; 8-9
  - transition feedback; 8-9
- simulated time,
  - mechanism; 16-5
- specifying time for; 16-7
- with and without time; 16-18
- with fusion sets; A2-19
- with hierarchy; A3-34
- with instance fusion sets; A2-25

## **simulator,**

- actions,
  - at the beginning of substep breakpoint; 8-15
  - when executing SalesNet; 10-9
- algorithm,
  - determining enablement; 7-4
- changing a net in; 11-12
- characteristics; 7-1
- entering,
  - with Enter Simulator (File menu); 8-7
- executing CP nets with; 7-14
  - (chapter); 8-1
- execution algorithm; 11-14
  - illustrating with SalesNet model execution with conflict; 11-15

## **simulator (cont'd),**

- leaving,
  - with Enter Editor command (File menu); 8-19
- role in mapping inputs to outputs; 11-3
- term definition; 7-1

## **size,**

- rectangle,
  - preserving the aspect ratio while changing; 4-12

## **sizes,**

- matching,
  - for transitions; 14-10

## **sockets,**

- as a type of fusion place; 12-8
- manually assigning ports to; A3-37
- port relationship to; A3-6
- relationship to ports; 14-5
- term definition; 12-8, A3-1

## **spacebar,**

- adjusting arc appearance with; 14-22

## **specifying,**

- colorsets; 6-11
- exact values for tokens; 7-4
- execution modes,
  - actual; 15-18
  - possible; 15-17
- initial marking; 6-11
- multisets; 5-8
- prime pages; 8-4, A2-13
- stop criteria; 15-19
- substitution transition location; A3-9
- time for a simulation; 16-7
- token values with CPN variables; 7-7
- transition decomposition,
  - methods for; 12-2

## **spread,**

- See Also* aligning;
- horizontal,
  - aligning nodes along with Horizontal Spread command (Align menu); 14-16
- vertical,
  - aligning nodes along with Vertical Spread command (Align menu); 14-17

## **staff members,**

- FirstModel,
  - characteristics; 13-2
- Sales Order Model,
  - characteristics; B1-2
  - simulation parameters; B1-4

## **starting,**

- See Also* entering; quitting; stopping;
- Design/CPN; 2-5

- starting (cont'd),**
  - Sales Order Model,
    - with Open (File menu); B2-3
  - tutorial; 2-10
- states,**
  - characteristics and term definition; 5-12
  - CP net,
    - initializing with Initial State command (Sim menu); 8-13
  - execution,
    - loading; 15-23
    - saving; 15-22
    - starting with a saved; 15-24
  - initial,
    - FirstNet model, concurrent execution; 11-4
    - term definition; 5-12
  - time in relation to; 16-3
  - transitions as representation of; 11-3
- static modeling paradigm,**
  - IDEF0,
    - characteristics; 3-2
  - term definition and characteristics; 3-1
- statistics,**
  - efficiency,
    - in Sales Order Model; B1-5
  - gathering and displaying,
    - in Sales Order Model; B1-5
  - revenue,
    - in Sales Order Model; B1-5
- status bar,**
  - characteristics; 2-6
  - information displayed during syntax checking; 8-3
  - messages generating during switching; 8-8
- steps,**
  - setting a limit,
    - as a stop criteria; 15-20
  - term definition; 11-14
- Stop (Sim menu),**
  - canceling CP net execution with; 8-18
- stop criteria,**
  - specifying; 15-19
- stopping,**
  - tutorial; 2-10
- string colorsets,**
  - characteristics and term definition; 5-6
- structure,**
  - hierarchical CP nets; A3-2
  - modeling; 3-1
- subdiagrams,**
  - loading; A2-12
  - saving; A2-12
- submodels,**
  - subpages use as; 15-4
  - term definition; 12-1, A1-2, A3-1
- subnet,**
  - term definition; A1-2
- subpages,**
  - breaking the connection between the substitution transition and its; A3-35
  - building FirstModel on; 14-4
  - connecting superpages to; 12-8
  - creating; 12-3, A3-9
  - deleting; A3-25
  - FirstModel; 13-6
  - editing; 14-6
  - final appearance; 14-5
  - improving the appearance; A3-14
  - navigating to superpage from a,
    - by double-clicking on a port; A3-6
  - similarity to subroutines and macros; A3-34
  - structure of; 12-8
  - term definition; 12-2, A3-1
  - use as submodels; 15-4
  - using more than once; A3-26
- subroutines,**
  - subpages compared to; A3-24, A3-34
- subsetting,**
  - multisets; 5-9
- substep,**
  - term definition; 11-14
- Substitution Transition (CPN menu),**
  - converting a transition to a substitution transition
    - with; A3-28
  - creating substitution transitions with manual port assignments; A3-38
- substitution transitions,**
  - (chapter); A3-1
  - breaking the connection to its subpage; A3-35
  - characteristics; A1-2
  - creating; A3-7
    - with Substitution Transition (CPN menu); A3-38
  - hierarchical decomposition with,
    - introduction (chapter); 12-1
  - hierarchy page changes; 12-11
  - improving the appearance of; 12-6
  - multiplicity and; A3-33
  - naming; 12-5, A3-10
  - reversing the creation of; A3-18
  - simulating with; A3-34
  - specifying the location of; 12-3, A3-9
  - substitution tag region,
    - term definition; 12-12

**substitution transitions (cont'd),**

- substitution transition creation mode,
- term definition; 12-3
- term definition; 12-1, A1-2, A3-1

**subtracting,**

- multisets; 5-9

**superpages,**

- connecting subpages to; 12-8
- converting SalesNet into FirstModel; 14-2
- FirstModel compared with SalesNet; 13-5
- improving the appearance; A3-11
- navigating to from a subpage,
  - by double-clicking on a port; A3-6
- Resource User Model; A3-3
- term definition; 12-2, A3-1

**switching,**

- among instances,
  - with the Instance Switch dialog; A3-35
- status bar messages generated during; 8-8
- term definition; 8-8

**syntax,**

- check,
  - performing; 8-2
- errors,
  - deciphering ambiguous messages; 9-7
  - handling (chapter); 9-1
  - locating; 9-3
  - missing colorset; 9-2
  - undeclared CPN variables; 9-6
- guards; 7-8

**Syntax Check (CPN menu),**

- performing a syntax check with; 8-2

**system default attributes,**

- term definition and characteristics; 6-3

## T

**telephone number,**

- Meta Software; 1-7

**text,**

- adding to,
  - a rectangle; 4-11
  - a rectangle, while creating it; 4-12
- editing; 4-21
- entering; 4-21
  - into a global declaration node; 6-16
- mode,
  - characteristics; 4-4
  - creating objects from; 4-17
  - term definition; 4-3

**text (cont'd),**

- pointers,
  - locating an error with; 9-4
  - term definition; 9-4
- regions,
  - copying; 14-26
  - pasting; 14-26
- text tool,
  - term definition; 4-4

**Text menu,**

- Turn Off Text command,
  - adding text to rectangles with; 4-11
- Turn On Text command,
  - adding text to rectangles with; 4-11

**time,**

- See Also* concurrency;
- increasing the realism of simulation with; 16-19
- modeling; 3-1
- non-representation in FirstModel; 16-3
- output arc inscriptions; 16-8
- real,
  - term definition; 16-2
- regions,
  - term definition; 16-8
- representation methods in CP nets; 16-4
- Sales Order Model chart depicting elapsed,
  - examining; B3-3
- Sales Order Model chart depicting elapsed; B3-3
- simulated,
  - (chapter); 16-1
  - characteristics; 16-2
  - term definition; 16-2
  - uses for; 16-6
- specifying for a simulation; 16-7
- stamps,
  - assigning, FirstModel; 16-11
  - giving to a token; 16-7
  - initial markings and; 16-10
  - omitting; 16-9
  - term definition; 16-4
- state in relation to; 16-3
- timed,
  - colorsets, declaring; 16-7
  - colorsets, term definition; 16-4
  - CP nets, compiling; 16-14
  - CP nets, executing; 16-16
  - token, term definition; 16-4

**tokens,**

- See Also* multisets;
- characteristics and term definition; 5-7
- constraining,
  - with more complex guards; 7-10

## **tokens (cont'd),**

- constraining (cont'd),
  - with simple guards; 7-9
- term definition; 5-3
- timed,
  - delay expression syntax; 16-7
- values,
  - constraining; 7-8
  - specifying exact; 7-4
  - specifying multiple constant; 7-5
  - specifying multiple instances of the same constant; 7-5
  - specifying one constant value; 7-4
  - specifying variable; 7-7

## **top-down development,**

- hierarchical CP nets; A3-21
- term definition; 12-2

## **transformation rules,**

- as dynamic modeling paradigm component; 3-3

## **transformations,**

- representing with transitions; 10-6

## **Transition (CPN menu),**

- creating transitions with; 6-5

## **transitions,**

- See Also* rectangles;
- as CP net activities; 3-3
- as state representation; 11-3
- characteristics and term definition; 5-13
- characteristics as CP net component; 5-1
- concurrent,
  - firing; 11-3
- creating; 6-5
  - for FirstModel subpage; 14-8
- decomposition of,
  - designating; 12-3
  - methods for specifying; 12-2
- enabled,
  - putting on enabled list, for SalesNet model execution with conflict; 11-15
  - setting their representation in an occurrence set; 15-13
- factors controlling,
  - enablement; 7-2
  - occurrence; 7-2
- naming,
  - for FirstModel; 14-10
- naming; 6-7
- occurrence of; 7-11
- representing activities with; 10-6
- shape-changing,
  - with Change Shape command (Makeup menu); A3-28

## **transitions (cont'd),**

- substitution,
  - breaking the connection to its subpage; A3-35
  - creating with Substitution Transition (CPN menu); A3-7, A3-38
  - hierarchy page changes; 12-11
  - improving the appearance of; 12-6
  - naming; 12-5, A3-10
  - reversing the creation of; A3-18
  - simulating with; A3-34
  - specifying the location of; A3-9
  - term definition; 12-1
- time regions,
  - characteristics and syntax; 16-8
- transition feedback region,
  - term definition; 8-9

## **troubleshooting,**

- common problem symptoms and solutions, (chapter); C1-1
- memory problems,
  - problem symptoms and solutions; C1-5
- ML,
  - configuration not specified, problem symptoms and solutions; C1-2
  - interpreter cannot be started, problem symptoms and solutions; C1-5
- resetting the tutorial environment; 6-5
- settings file missing or obsolete,
  - problem symptoms and solutions; C1-1

## **tuples,**

- colorsets,
  - characteristics and term definition; 13-4
- constructors,
  - characteristics; 13-8
  - example of use in FirstModel execution; 13-9
  - term definition; 13-9
- patterns,
  - characteristics; 13-12
  - term definition; 13-12
- term definition; 13-4
- value,
  - term definition; 13-4

## **Turn Off Text (Text menu),**

- adding text to rectangles with; 4-11

## **Turn On Text (Text menu),**

- adding text to rectangles with; 4-11

## **tutorial,**

- environment,
  - establishing; 6-3
- starting; 2-10
- stopping; 2-10

## U

**unbound,**  
term definition; 5-10  
term definition; 7-7  
**Ungroup (Group menu),**  
deselecting groups with; 4-27  
**updating,**  
enabled list,  
for SalesNet model execution with conflict; 11-20  
**user interface,**  
design,  
*See* appearance;  
Design/CPN; 2-5

## V, W

**values,**  
exact,  
specifying for tokens; 7-4  
tokens,  
constraining; 7-8  
tuple,  
term definition; 13-4  
variable,  
specifying for tokens; 7-7  
**variables (CPN),**  
characteristics and term definition; 5-10  
rebinding,  
during SalesNet execution; 10-9  
during transition firing; 7-12  
specifying token values with; 7-7  
**vertexes,**  
term definition; 4-19  
**Vertical (Align menu),**  
aligning nodes in a column with; 14-18  
**vertical spread,**  
aligning nodes along,  
with Vertical Spread command (Align menu);  
14-17  
**Vertical Spread (Align menu),**  
aligning nodes with; 14-17

## X, Y, Z

**X-Windows,**  
Caps Lock key behavior; 4-7  
Design/CPN use of; 2-3  
tutorial use with; 2-2





# **PART 1**

## **CP Net Fundamentals**

# Chapter 1

## The Design/CPN Tutorial

This is the Design/CPN Tutorial. Its goal is to teach you how to use Design/CPN to do modeling and simulation with Petri nets.

This tutorial assumes that you are familiar with computers and computer programming, and know how to use the particular computer on which you will be working with Design/CPN. It requires no prior familiarity with Petri nets, system design and analysis, modeling, simulation, or any particular computer language.

This tutorial does not attempt to teach the general theory of system design and analysis, or the mathematical formalism that underlies Petri nets. Its emphasis is on the practical, hands-on use of Design/CPN to build and execute Petri net models. For an accessible but mathematically rigorous introduction to Petri nets, see *Colored Petri Nets*, by Kurt Jensen (Springer-Verlag, 1992).

### What Is a Petri Net?

A *Petri net* is a network of interconnected locations and activities, with rules that determine when an activity can occur, and specify how its occurrence changes the states of the locations associated with it. Petri nets originated in the work of C. A. Petri in 1962, and have since been developed by many researchers in many countries.

Petri nets can be used to model and simulate systems of any type. They are particularly useful in facilitating the design and analysis of complex distributed systems that handle discrete flows of objects and/or information.

There is an extensive mathematical formalism associated with Petri nets. This formalism completely defines what a Petri net is and how it behaves. Although Petri nets are typically represented as graphs drawn on paper or on a computer screen, a Petri net is actually a mathematical object that exists independently of any physical representation.

There is no need to understand the mathematics of Petri nets in order to use them. Just as an engineer can use scientific theories to build useful devices without having to become a scientist, so a system

designer can use Petri nets to build useful models without having to become a mathematician.

Petri nets have been developed over the years from a simple yet universally applicable paradigm to a more complex but far more convenient methodology, the *hierarchical colored Petri net*. Such nets are hierarchical in that they contain facilities for representing a model as a hierarchical structure, and are “colored” in that they allow data to have different types and values (“colors”); earlier varieties of Petri nets allowed only boolean (“black and white”) data.

For brevity, hierarchical colored Petri nets are usually called *CP nets*. A CP net that models a system is called a *CPN model*. This tutorial is concerned only with CP nets and CPN modeling. For information on other types of Petri nets, see Jensen’s *Colored Petri Nets*, referenced above..

## Overview of the Design/CPN Tutorial

This tutorial is divided into five parts:

- Part 1: CP Net Fundamentals
- Part 2: Design/CPN Techniques
- Appendix A: CPN Hierarchy Techniques
- Appendix B: The Sales Order Model
- Appendix C: Troubleshooting

### Part 1: CP Net Fundamentals

Part 1 introduces and defines CP nets, and shows how such nets can be used for modeling and simulation. The general concepts are presented in the context of a small net that you build and execute using Design/CPN.

### Part 2: Design/CPN Techniques

This part provides the essential information you need in order to use Design/CPN to do realistic modeling and simulation with CP nets.

### Appendix A: CPN Hierarchy Techniques

CP net modeling is similar in many ways to ordinary programming. Many of the same techniques are useful in both enterprises, such as

the capabilities provided by global data and by reusable subroutines. Hierarchical nets provide these capabilities. Appendix A shows you how to use them.

### **Appendix B: The Sales Order Model**

The Sales Order Model illustrates a variety of advanced CPN techniques. Appendix B shows you how to run it and experiment with it.

### **Appendix C: Troubleshooting**

Design/CPN requires a particular computer environment in order to run, as described in the installation notes that accompany the product. If the computer environment is incorrect, problems may occur. Appendix C describes these problems and gives their solutions.

## **Strategy of the Tutorial**

You don't need to know everything, or even most things, about CP nets or Design/CPN in order to use them. All you need is a core of essential information, techniques, and intuitions that together are sufficient to get you going. Therefore this tutorial does not try to teach everything about any of the topics that it covers. Its goal is to teach the essentials of all of them, and the relationships among them, with particular attention given to matters that tend to be difficult when first encountered.

The information you need in order to use CP nets and Design/CPN is not particularly complex or difficult, but it is highly interconnected. There are many cases where you must have all of a set of concepts in order to fully understand any of them. It is difficult to present such material sequentially, and still define every concept before it is used.

This tutorial copes with the problem by taking an iterative path. Most topics are covered several times, with each treatment taking advantage of intervening material to achieve a greater depth. Details about mutually dependent topics are often postponed until they can all be presented together, so they can provide for each other the context needed to make them all meaningful.

Once you have mastered the concepts and skills that are covered in this tutorial, you should have little trouble acquiring additional information as needed from appropriate reference manuals and textbooks, and from your experiences with Design/CPN itself.

### How to Use the Tutorial

Four principles, if rigorously followed, will greatly facilitate your use of this tutorial:

#### Proceed Systematically

CP nets are based on a relatively small number of fundamental principles. On these is built a methodology of very great generality and power. Don't let the simplicity of the fundamentals deceive you into rushing through them. Every detail deserves careful examination and complete understanding before you proceed to the next.

As you go through the tutorial, it is essential that you take the time to master each exercise before you go on, even if the ultimate purpose of the exercise is not entirely obvious. All of the information that the tutorial presents is there for a reason, and all of it will eventually be used.

If you proceed systematically through the tutorial you will always be on solid ground. If you do not, you will soon find yourself lost in a jungle of undefined terms, meaningless concepts, and unusable techniques. Even if you believe you already know about some topic being covered, you should read each chapter completely to verify that your understanding corresponds to the material being presented.

#### Ignore the Unexplained

This tutorial is organized so that at each point you have the information you need to understand the material currently being presented. But providing the necessary prerequisite information at each point is not the same as answering every question that might be asked at that point. When mutually dependent topics are studied sequentially, as in this tutorial, questions frequently arise that cannot be answered immediately. They cannot be answered until additional information has been presented that is necessary in order for the answer to make sense.

As you proceed through this tutorial, you will probably have questions that are not answered at the point where it becomes possible to ask them, and you will often see features of Design/CPN that have not yet been explained. Please be patient: the answers to the questions should become clear as you proceed, and the Design/CPN features will be covered once you have the information you need in order to use them.

### Review Frequently

There is not much repetition in this tutorial: it assumes at every point that you are familiar with all terms, methods, and concepts presented up to that point. In iterating through topics to gain greater depth, it does not recapitulate what was covered on the previous round, but continues from where it left off.

The reason this tutorial does not repeat material is that doing so would make the tutorial enormous, yet most of the repetition would be wasted on any particular reader, serving only to dilute new material in a flood of redundant explanation. But no-one can learn all of a body of information in a single pass: there is always a need to repeat some points.

If at any time you are unclear on what was said about a topic that the tutorial previously discussed, or on how to perform some operation that the tutorial previously described, you should immediately review the relevant sections of the tutorial. If you do this you will maintain the firmness of the foundation on which you are building. If you do not, you will leave it full of holes, and it will eventually collapse.

### Build the Models!

Modeling and simulation are not primarily something to think about; they are something to do. It is all too easy to understand them in principle without being able to apply the understanding to real situations.

This tutorial includes a variety of models that are to be built using Design/CPN on the computer. You should take the time to actually build these models, and to thoroughly understand each one before proceeding to the next. In learning to model and simulate with CP nets, there is no substitute for hands-on experience.

### Beyond the Tutorial

When you have completed this tutorial, you will know many things about modeling and simulation generally, and about CP nets and Design/CPN specifically. But you will not be at the end of the process of learning about these things—you will be at the beginning. There are four reasons for this:

The first is that this tutorial deals almost exclusively with modeling and simulation. But these activities do not exist in isolation. They are embedded in the much larger realm of system design and analysis, about which the tutorial says little. Numerous books that deal with system design and analysis are currently in print.

The second is that this tutorial covers only a portion of the information that is available about any topic that it deals with. There is more, often much more, to every subject that the tutorial mentions: the tutorial's goal is only to get you started. For more information about CP nets, see Jensen. For more about Design/CPN, see *The Design/CPN Reference Manual*.

The third reason is that becoming a skilled modeler is not just a matter of acquiring knowledge, skills, and techniques. These are necessary but not sufficient. You would not expect to become an expert programmer, engineer, scientist, or artist on the basis of knowledge alone. It is the same with modeling, which overlaps all four of these areas. Real proficiency in modeling comes only with time and experience, which no document can provide.

Fourth, useful modeling is not only a matter of becoming skilled at modeling per se. There is also the matter of learning to model systems of the particular type that is of interest to you. No one way of thinking applies to the modeling of every type of system, and the ways of thinking that are customary for envisioning systems of various types do not necessarily lead to effective modeling.

For these reasons, completing this tutorial is only the first phase of the task of learning to use Design/CPN effectively. If you use the tutorial correctly, it will provide you with a solid foundation for using Design/CPN to model and simulate with CP nets. On that foundation you will be able to build whatever structure of expertise you need in order to solve particular problems using Design/CPN. But keep in mind that this tutorial can provide a foundation only: the rest is up to you.

## Request For Feedback

The Design/CPN Tutorial cannot succeed without detailed commentary and criticism from its users. Meta Software is interested in feedback at all levels, from the most general to the most detailed. Some general questions:

- What do you think of the overall approach and organization of the tutorial?
- Is the tutorial too elementary? Too advanced? Is it consistent in its level?
- Where would more information be useful? Where has too much been provided?
- How would you have done things differently in order to better meet the needs of a reader such as yourself?



- What other questions should we be asking in order to get better feedback from users of the tutorial.

Please mail your comments to Meta Software, CPN Tutorial, 125 Cambridge Park Drive, Cambridge, MA, 02140; fax them to the attention of CPN Tutorial at 617-661-2008; or email them to [tutorial@metasoft.com](mailto:tutorial@metasoft.com). Every comment will be carefully considered, and the next version of the tutorial will reflect all comments to the greatest achievable extent.



# Chapter 2

## Getting Started With Design/CPN

CP nets allow system designers and analysts to move the often difficult and sometimes impossible task of working directly with real systems into the more tractable and inexpensive realm of computerized modeling, simulation, and analysis. Such a move solves many problems, but it does not solve them all.

A model of a complex system may itself be complex. Methods for dealing with this complexity are required in order for the model to be created and used effectively. Furthermore, a computer model is a form of computer program, and as such can encounter the same types of problems that more conventional computer programs face. Syntactic errors, semantic errors, and design errors all can occur. These must be detected and corrected if the model is to successfully perform its function.

### What Is Design/CPN

Design/CPN is an interactive computer tool for performing modeling and simulation with CP nets. Design/CPN provides:

- An editor for creating and manipulating CP nets.
- Syntax checkers for validating CP nets.
- A simulator for executing CP nets.
- Interactive monitoring and debugging capabilities.
- Facilities for organizing a net into a hierarchy of modules.
- Animation and charting facilities for displaying simulation results.

These capabilities allow CP net models to be conveniently created, modified, organized, executed, debugged, examined, and validated.

Design/CPN thereby makes the theoretical power of CP nets available in practice for modeling arbitrarily large and complex systems.

This tutorial shows you how to use Design/CPN to build and execute CP nets. Once you have mastered the material in this tutorial, you will be able to use CP nets and Design/CPN to do useful modeling. You should then have little difficulty picking up additional techniques as needed to perform any modeling task for which CP nets are appropriate.

## Prerequisites for This Tutorial

This tutorial assumes that you:

1. Know how to start X-Windows programs.
2. Are familiar with the X-Windows user interface generally.

If you are unfamiliar with any of these matters, you should be sure to acquire the necessary information before you continue with this tutorial.

Before you proceed, be sure that Design/CPN has been correctly installed on your machine. The Design/CPN Installation Procedure is included in the documentation shipped with the product.

In order to use this tutorial you will need to know where on the computer to find:

1. The Design/CPN program.
2. The directory named TutorialDiagrams that is included with each release of Design/CPN.
3. The directory named examples that is included with each release.

Be sure that you know where these are before you go on.

## Using This Tutorial With X-Windows

This version of the Design/CPN Tutorial shows Macintosh-style dialog boxes rather than X-Windows-style boxes. In almost every case, the Macintosh and X-Windows boxes are functionally identical, differing only in the details of their appearance. Where there is a functional difference that bears on a topic under discussion in the tutorial, the difference is explicitly described.

Other than dialog box appearance, there should be no differences between what this tutorial describes and what happens under X-Windows.

## Design/CPN and X-Windows

Design/CPN is a standard X-Windows program. All standard X-Windows user interface techniques are available, and work as they do in X-Windows programs generally.

### Design/CPN Multiprocessing

When Design/CPN is active, it sometimes accesses a second process, called the *ML process*, that interprets and compiles computer code. The two processes communicate and cooperate with each other to execute CP nets.

In order to be executed, a net must contain some information that Design/CPN uses to access the ML process. This information is automatically included in a new net when it is created. When a pre-existing net is imported from another system, as with the example nets supplied with this tutorial, the necessary information must be explicitly loaded into the net before it can be executed. Instructions for loading it appear where needed in this tutorial.

### Design/CPN and the File System

A *diagram* is a CP net and optional additional graphics created using Design/CPN. Design/CPN stores a diagram as a group of three files. Multiple files are used rather than one because a single file would often be inconveniently large. The files are:

1. The *diagram file*. This contains data that represents the diagram in a form suitable for displaying and editing.
2. The *DB file*. This contains a database describing the diagram. It has the same name as the diagram file, with the addition of the suffix “.DB”.
3. The *ML file*. This contains code that represents the diagram in executable form. It has the same name as the diagram file, with the addition of the suffix “.ML”.

Thus the diagram AirlineModel would be stored in the files:

AirlineModel  
AirlineModel.DB  
AirlineModel.ML

## Design/CPN Tutorial

---

The diagram and DB files contain a complete description of the diagram. The ML file is not generated until it is needed, and can be re-generated if necessary using the information in the diagram and DB files. Therefore a diagram may not have an ML file, either because the file was never created by Design/CPN, or because it was subsequently deleted. ML files are quite large, and are often deleted to save disk space.

DB and ML files are for internal use only by Design/CPN. When you want to open a particular diagram, open the diagram file; DB and ML files cannot be opened directly.

The only time you need to think about DB and ML files is when you copy, move, or rename a diagram via file system commands. Be sure to treat all of the files that constitute the diagram in the same way.

- Navigate to the directory examples.
- List its contents.

You can now see the multi-part form in which diagrams are kept. (There may not be any ML files.)

### Design/CPN Use of the Mouse

When you manipulate windows and scroll bars while using Design/CPN, the three mouse buttons have their standard functions in the X-Windows user interface. When you perform a mouse operation that is specific to Design/CPN rather than to X-Windows in general, use the left button. Design/CPN uses only one mouse button (to allow for porting to one-button mouse systems such as the Macintosh), so it ignores the middle and right buttons.

### Design/CPN Use of the Keyboard

Design/CPN documentation specifies the use of the ALT key in various keystroke shortcuts. Some keyboards use a DIAMOND ( ) key instead; occasionally other keys are used. If ALT does not produce the described results, check your terminal configuration.

## Establishing a Tutorial Directory

As you go through this tutorial, you will create several diagrams. Some will be based on existing diagrams that are supplied with the tutorial, in the TutorialDiagrams directory, and others will be entirely new.

It is best not to save diagrams you create into the TutorialDiagrams directory: the directory should be kept in exactly its original condition, so that it will always match the assumptions made about it in the tutorial. Therefore:

- Create a directory somewhere to hold diagrams you create while working with this tutorial. Call the directory NewTTDiagrams.

## Starting Design/CPN

- Start Design/CPN as you would any X-Windows program.

If you see a dialog that mentions a problem of any kind, see Appendix C before you proceed.

You are now in Design/CPN, and the editor is active.

## Opening a Diagram

To open a diagram:

- Choose **Open** from the **File** menu.

The **Open File** dialog appears. We will need a particular diagram to be open during the next few sections. To open it:

- Navigate to the directory TutorialDiagrams.
- Open the diagram SalesNetDemo.

The SalesNetDemo diagram opens, and a window named Sales#1 appears.

## The Design/CPN User Interface

You should now be looking at a screen that contains three things:

1. A *menu bar* at the top of a small window.
2. A *status bar*, immediately below the menu bar in the same window.
3. A *page* in a second window.

# Design/CPN Tutorial

---

## The Menu Bar

This is an ordinary pulldown menu bar. Take a moment to pull down each of the menus and briefly examine it.

New users of Design/CPN are sometimes daunted by the number of commands in the Design/CPN menu, and the unfamiliarity of the operations described in their somewhat cryptic names. But don't worry. By the time you have completed this tutorial, you will know what most of these commands are for, why they exist, and how to use them effectively to assist you in working with CP nets. Complete information on all Design/CPN commands is contained in *The Design/CPN Reference Manual*.

## The Status Bar

The status bar is used by Design/CPN to post brief messages that describe diagram components, editor and simulator states and actions, and various other things that you may want to know about as you work with Design/CPN. The messages currently posted are Type: Place on the left, Text: Off in the center, and Page Scale: 100% on the right. The meanings of these and many other status bar messages will be explained as we proceed.

If you are ever in doubt about where you are in Design/CPN, or want to see if anything is going on that you should know about but do not, check the status bar. It can be extremely informative.

## The Page

A page is a “blank slate” on which you can create CP net structure, as described in the following chapters. A diagram may contain any number of pages.

Every page is displayed in a separate window. A newly created page is centered under its window, rather than extending down and to the right as a file in a text editor would. It is centered because CP nets, being graphical rather than textual, tend to grow radially rather than linearly, so that empty drawing space is as likely to be needed in one direction as in another.

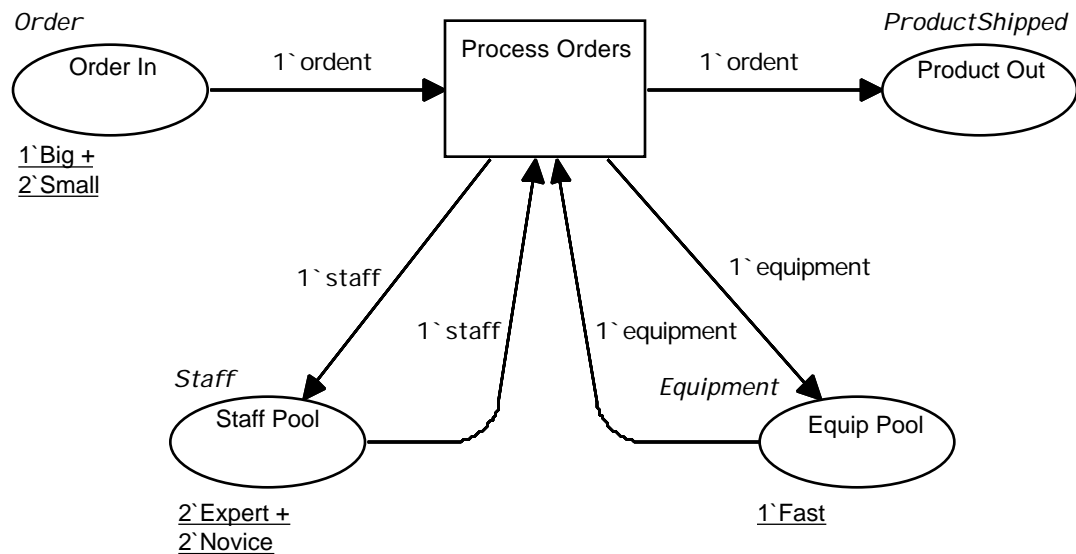
The center of a page is marked by a rectangle, called the *page border*. The dimensions of this rectangle are the **Page Width** and **Page Height** specified in the **Page Attributes** dialog (**Set** menu). If you print a page and do not specify any other behavior (via **Page Setup** from the **File** menu), the area inside the page border will be printed.

The page currently on display is named Sales#1. This name appears in the title bar of the window that holds the page. The page contains



a small net that represents a high-level view of a simple Sales Order Processing system. This net is called SalesNet:

[if ordent = Big then staff = Expert else staff = Novice]



You don't need to worry about the details of this net now. You will soon understand everything that is in it, and know how to build and execute it.

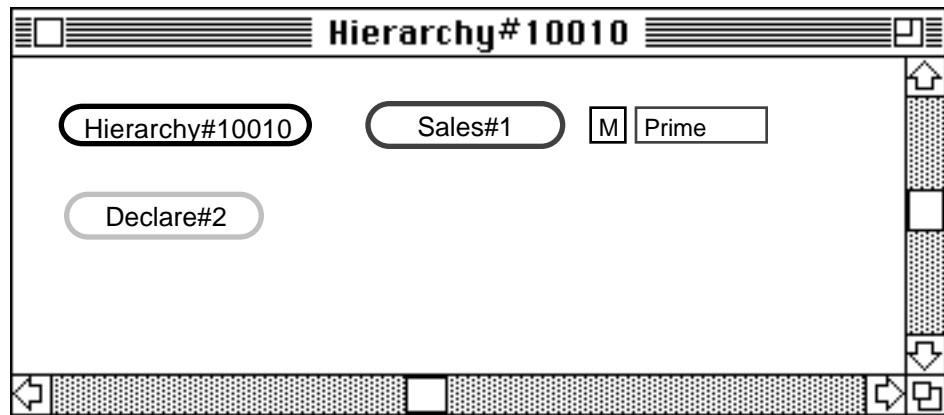
## Navigating a Diagram

A Design/CPN diagram typically consists of several pages. A given page may be *open*, in which case it is visible in a window (as Sales#1 now is), or *closed*, in which case it is not visible.

When a diagram contains more than one page, a method is necessary for keeping track of the pages and their relationship to each other. To provide this capability, every diagram contains a special page called a *hierarchy page*. Let's look at SalesNet's hierarchy page.

- Choose **Open Page** from the **Page** menu.

The hierarchy page opens:



(The window will be larger on your screen, but will contain the information shown here.)

A diagram's hierarchy page contains a small oval, called a *page node*, for every page in the diagram, including itself. Each page node contains the name of the corresponding page. This hierarchy page includes three page nodes:

- Hierarchy#10010, representing the hierarchy page itself.
- Sales#1, representing the page you were looking at before opening the hierarchy page.
- Declare#2, a page containing data declarations. We'll look at it in a moment.

A page node may be accompanied by additional information about the page. The designations "M" and "Prime", in two boxes next to the Sales#1 page node, are examples of such information. Their meaning will be explained later in this tutorial.

This hierarchy page is not particularly interesting, since there are so few pages to keep track of. When there are many pages linked into a hierarchical net, the hierarchy page becomes a rich source of information about the composition and structure of the net.

You can use the hierarchy page to navigate to any other page in a diagram. To illustrate the technique, let's look at the page Declare#2:

- Select the page node for Declare#2 by clicking the mouse on it.
- Choose **Open Page** from the **Page** menu.

Declare#2 opens. It contains a small box of computer code:

```
color Order = with Big | Small;  
color ProductShipped = Order;  
color Staff = with Expert | Novice;  
color Equipment = with Fast | Slow;  
  
var ordent : Order;  
var staff : Staff;  
var equipment : Equipment;
```

This computer code is in a language called CPN ML. The code defines the datatypes and variables that are used in the net on page [Sales#1](#). As before, don't worry about the details now.

Let's go back to [Sales#1](#), using a slightly different method.

- Choose **Open Page** from the **Page** menu.

The hierarchy page reappears.

- Double-click the page node for [Sales#1](#).

[Sales#1](#) is again the current page.

## Printing a Diagram

Printing a diagram is similar to printing any other file. The details of the printer dialog depend on the particular printer you are using, so they cannot be covered here.

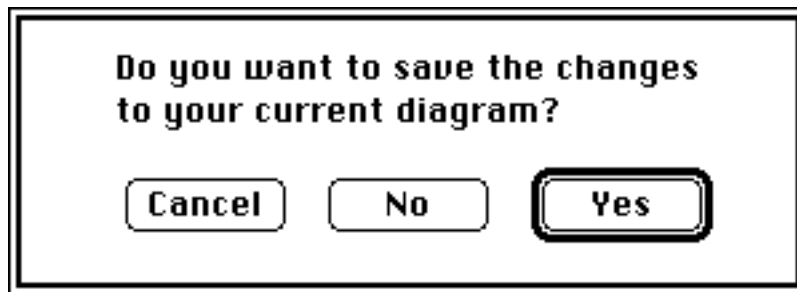
Every Design/CPN page name includes a page number, e.g. [Sales#1](#) and [Declare#2](#). Page numbers can be used to tell the printer to print a particular page or range of pages. Diagram pages have no intrinsic order, so Design/CPN numbers them by assigning each new page the lowest number not currently in use in the diagram.

## Closing a Diagram

To close a diagram:

- Choose **Close** from the **File** menu.

The **Save Changes** dialog appears:



Design/CPN displays this dialog whenever you have made changes to a diagram's contents, or have opened, closed, or changed any windows that hold diagram pages. There is no need to save changes now, so:

- Click **N**o.

The dialog disappears, and the diagram closes.

## Quitting Design/CPN

To quit Design/CPN:

- Choose **Quit** from the **File** menu.

Design/CPN quits. You are back at Operating System level.

## Starting and Stopping the Tutorial

Different users work through a tutorial like this on very different schedules. One person might work through small pieces of chapters at irregular intervals, while another might work through several chapters in a single session.

There is no way for this tutorial to anticipate how you will use it. Therefore, from this point onwards, the tutorial generally does not mention starting or quitting Design/CPN. It assumes that you will perform these actions as needed, using the techniques given in this chapter, before you begin working with Design/CPN and after you are done.

# Chapter 3

## Modeling Paradigms

In order to make a model of a system, we need some way to represent the components that a system consists of. These representations do not have to be similar in detail to the components of real systems. What we need is a set of abstractions that will allow us to capture the essence of any system that we wish to model. Such a set of abstractions is often called a *modeling paradigm*.

There is no one modeling paradigm that is best in all cases. The choice of paradigm depends on the nature of the system being modeled, and on the purpose of the model. For systems such as the weather, where there is no flow of information but only a succession of states, systems of partial differential equations are used. For modeling physical objects, such as the components of a machine, CAD/CAM methods are best.

Humans frequently design systems to perform complex tasks that are distributed over space and time, and that involve discrete flows of objects and/or information. The social systems in which people live are also of this nature. Such systems can often be improved by modeling them.

Where only the structure of such a system is to be modeled, static modeling paradigms are useful. Where structure and behavior must both be modeled, dynamic modeling paradigms are needed. Lets take a brief look at each of these types of modeling.

### Static Modeling Paradigms

A *static modeling paradigm* is one that can represent the structure of a system, but not its behavior over time. When structural representation is all that is needed, a static modeling paradigm is sufficient. At the minimum, a static modeling paradigm must represent:

- **Activities:** These represent the constituent actions of the modeled system.
- **Connections:** These show the relationships between activities.

- **Descriptions:** These describe the activities and their relationships

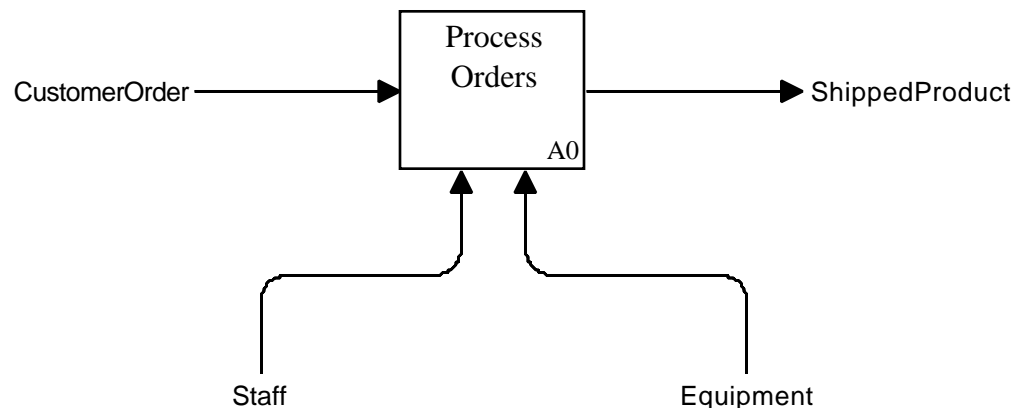
The components of a static modeling paradigm do not necessarily have a one-to-one relationship with these capabilities, but the paradigm must provide at least this much in order to be useful. Frequently these are all that is needed for static modeling.

### IDEF0 Modeling

One of the most widely used static modeling paradigms is called *IDEF0*. An IDEF0 model is a graphical structure consisting primarily of:

- **Activity Boxes** to represent activities.
- **Arrows** to represent connections between activities.
- **Labels** to describe the activities and their relationships.

For example:



You may notice a resemblance between this model and the CPN model you saw briefly in the diagram [SalesNetDemo](#) in Chapter 2. That model is in fact a CPN version of the IDEF0 model shown here.

This tutorial does not cover IDEF0 modeling, but it mentions it occasionally, because there is a close relationship between IDEF0 models and CPN models. Meta Software provides an interactive tool, Design/IDEF, for performing IDEF0 modeling.

### Dynamic Modeling Paradigms

Static modeling paradigms are insufficient for representing system behavior over time. The reason is that they provide no way to represent a particular state of a system, or for specifying how the system's state will change as time passes. Lacking such provisions, static models cannot execute.

A *dynamic modeling paradigm* is one that can represent both the structure and the behavior of a system. At the minimum, a dynamic modeling paradigm must provide:

- **Data:** Data of appropriate types and values must be available to represent the objects and information that the system manipulates.
- **Locations:** The data must be stored somewhere, so we know where to find it when we need it.
- **Activities:** These transform data, and thereby move the model from one state to the next.
- **Connections:** Locations and activities must be linked so as to represent the flow of data through the model.
- **Activation Rules:** These determine when an activity can take place.
- **Transformation Rules:** These determine the effect of an activity taking place.

The components of a dynamic modeling paradigm do not necessarily have a one-to-one relationship with these capabilities, but one way or another the paradigm must provide them all.

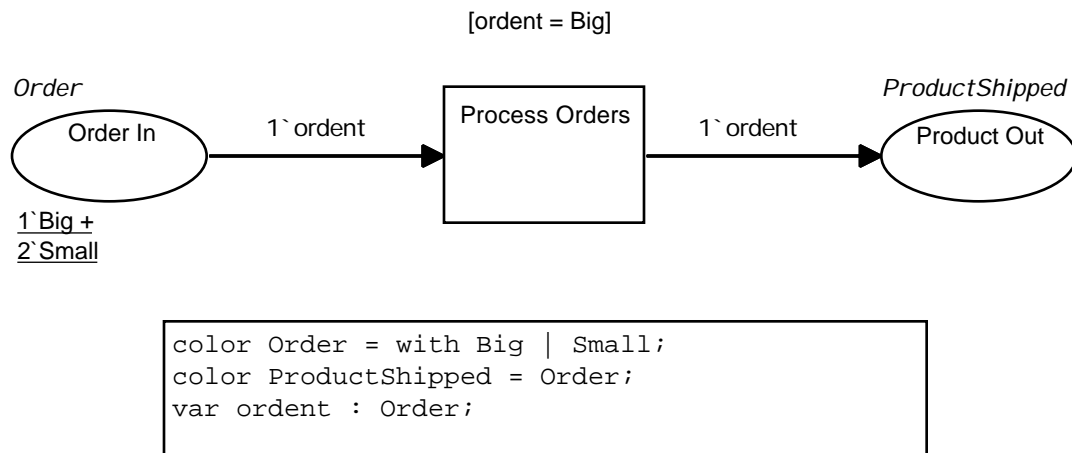
### CP Net Modeling

CP nets provide an extremely effective dynamic modeling paradigm. A CP net is a graphical structure with associated computer language statements. The primary components of a CP net are:

- **Data:** CP nets define datatypes, data objects, and variables that hold data values.
- **Places:** Locations for holding data objects.
- **Transitions:** Activities that transform data.
- **Arcs:** Connect places to transitions.

- **Input Arc Inscriptions:** Specify data that must exist in order for an activity to occur.
- **Guards:** Boolean expressions that further define conditions that must be true for an activity to occur.
- **Output Arc Inscriptions:** Specify data that will be produced if an activity occurs.

For example:



This example probably looks somewhat cryptic now, but don't worry: you will soon understand it completely, and be able to create and execute it. Small though this net is, it demonstrates most of the techniques needed to create even the most complex CP net.

## IDEF0 Modeling and CP Net Modeling

For brevity, the CP net modeling paradigm is generally referred to as *CPN*.

IDEF0 and CPN are closely related. They are based on similar ideas, and use similar components and graphical conventions. IDEF0 activities correspond to CPN transitions, IDEF0 arrows correspond to CPN arcs, and IDEF0 labels correspond to CPN datatypes.

These correspondences are not perfect, but they are close enough to make CPN effectively a superset of IDEF0. Everything contained in an IDEF0 model of a system is needed in a CPN model of that system, plus additional information to add the details necessary to provide executability.



It is therefore a common practice to use IDEF0 and CPN together to do system modeling. The general method is to use IDEF0 to explore the general structure of a system without taking the time to specify every detail. Such exploration can often identify with relatively little effort the parts of a system in which problems are likely to arise. Then those parts of the model that relate to potential problem areas are converted to CPN and executed.

This method provides the best of both worlds. IDEF0 is used to locate possible problems without putting a lot of effort into detailed representation, and CPN is used to fill in details only where there is reason to believe that the additional effort of doing so will be worthwhile. Executing the CPN model can then provide a wealth of information that could never have been derived using IDEF0 alone.



# Chapter 4

## Using the Design/CPN Editor

If you are an experienced user of Meta Software's Design/IDEF or MetaDesign, you already know almost all of the material in this chapter. You should nevertheless skim the chapter and note any details that differ from your understanding.

If you are familiar with any object-oriented graphics application (one that stores graphics as drawing instructions rather than as bitmaps), you already know some of what is in this chapter. Nevertheless you should read the chapter and do all exercises that do not seem completely obvious.

If you are not experienced with object-oriented graphics editing, you should work through this chapter very carefully, and do every exercise completely.

Whatever your current level of expertise, be sure you have mastered the techniques described in this chapter before you proceed through the tutorial. If you do, you will find that before long you can create and modify CP net structure almost as easily as you can enter and edit text using an ordinary text editor.

Most modeling tasks do not require the full power of the editor. This chapter presents a subset of the editor that provides the minimum capabilities that are required for working effectively with CP nets. For information about the many additional capabilities of the Design/CPN editor, see *The Design/CPN Reference Manual*.

### The Design/CPN Graphics Editor

The Design/CPN editor provides facilities for dealing with both geometric forms and textual information. It represents such information as *graphical objects*. Each graphical object represents one graphical entity: a rectangle, ellipse, text string, etc. A graphical object is stored as a set of instructions for drawing the particular object on the screen. Graphical objects can be modified after they are created by changing the instructions that constitute them.

Design/CPN graphical objects are of two kinds: auxiliary objects and CPN objects. An *auxiliary object* is just an ordinary graphical

object, such as any graphics editor can produce. A *CPN object* is a graphical object that is part of a CP net.

The difference is not one of appearance, but one of function. An auxiliary object is just itself; there is nothing to it but its physical form. A CPN object is more than its physical form: it also has a meaning as a CP net component.

Auxiliary objects are used primarily to add commentary to nets. Objects can be converted between auxiliary and CPN status as needed. This permits objects to be deactivated as net components without having to be deleted, just as otherwise executable lines are often “commented out” for various reasons during development of an ordinary program.

You could start learning about the editor by immediately creating CPN objects, but this approach is not the best. The overhead of dealing with the specific details of CP net syntax would interfere with the task of learning general editor techniques.

Our approach will be to first study methods for creating graphical and textual objects without trying to use them for anything. Therefore all the objects you work with in this chapter will be auxiliary objects. In later chapters you will use exactly the same techniques to work with CP net components.

## Design/CPN Graphical Objects

The Design/CPN editor provides many types of graphical objects, including rectangles, ellipses, connectors, labels, and many more. All of these are created and edited in essentially the same way; the details differ slightly from type to type, to suit the individual characteristics of each.

Every Design/CPN graphical object is either a node or a connector. A *node* is any object that can exist in its own right, such as a rectangle or an ellipse. A *connector* is an arrow that runs between one node and another. A connector cannot exist independently: it can exist only when there are two nodes for it to connect.

Every Design/CPN graphical object consists of one or both of:

1. A geometric form.
2. A text field.

One object type, the *label*, has no geometric form and a required text field. Every other object type has a geometric form and an optional text field.

A graphical object's geometric form and/or text field define its fundamental properties, but do not specify the details of its physical appearance: its line thickness, fill pattern, color, font, etc. All aspects of an object's appearance can be controlled by setting various graphic and textual attributes of the object. Such attributes are called *display attributes*.

## Graphics Editor Modes

It is often useful to perform the same operation repeatedly while using the editor: to create a series of rectangles, for example, or to edit text in a series of locations. To facilitate such repeated operations, the editor offers a variety of *editor modes*. When the editor is in a particular mode, you can perform the mode's characteristic action repeatedly using only the mouse and keyboard, without the need to respecify each time what you want to do.

In each editor mode, Design/CPN displays a distinctive mouse pointer that indicates the mode. Such a pointer is known as a *drawing tool*. Specific drawing tools are described later in this chapter.

## Editing Graphical Objects

You can use the editor to modify an existing graphical object at any time. The modification may involve moving, reshaping, and deleting geometric forms, such as rectangles or ellipses, or entering, moving, and deleting text.

The capabilities needed for editing geometric forms differ from those needed for editing text. The editor therefore has two *editing modes*: graphics mode and text mode. In *graphics mode*, you can move and reshape an object's geometric form using the mouse. In *text mode*, you can use both the mouse and the keyboard to edit an object's text field.

### Graphics Mode

When the editor is in graphics mode, the status bar displays Text: Off. If you are not performing any particular activity, the mouse pointer is the *graphics tool*:



(The tool is shown larger than actual size.)

The graphics tool simply indicates a location: the location it indicates is the pixel under the dot at the center of the tool. All drawing tools that indicate a specific pixel location include such a dot.

As you perform particular activities in graphics mode, the mouse pointer changes to various other tools to indicate the activity currently underway, then reverts to the graphics tool when the activity is complete.

### Text Mode

When the editor is in text mode, the status bar displays Text: On. If you are not performing any particular activity, the mouse pointer is the *text tool*:



This tool indicates a location, as the graphics tool does, and contains a “T” to indicate text mode. As you perform particular activities in text mode, the mouse pointer changes to various other tools to indicate the activity currently underway, then reverts to the text tool when the activity is complete.

## Creating Graphical Objects

When you are in the editor, you can create new graphical objects at any time. The general sequence for creating a new object is:

1. Use the **Aux** menu to choose the type of object you want to draw.
2. Use the mouse to specify the location and shape of the object.
3. Create more objects of the specified type if desired.

When you choose an object type from the **Aux** menu, the editor switches from whichever editing mode it is in (graphics or text) to a *creation mode*. Mouse movement then creates objects of the chosen type. When you are done creating objects of that type, you exit the creation mode. There are two ways to exit a creation mode:

1. Press ESC.
2. Click the mouse anywhere in the menu bar.

When you do one of these, the editor leaves the creation mode, and returns to whichever editing mode it was in before you began creating. For brevity, the rest of this tutorial specifies using ESC to exit a creation mode. If you prefer, you can click in the menu bar instead.

When the editor is in a creation mode, and you are not performing any particular activity, the mouse pointer is a tool that indicates the mode. As you go through the various steps that create an object, the mouse pointer changes to various other tools to indicate the current step, then reverts to the relevant creation tool when the step is complete. Specific drawing tools used during object creation are described in the following sections.

## Autoscrolling

Usually you can use the mouse and the scroll bars to move to any part of a page. During some operations for creating and editing graphical objects, the mouse becomes dedicated to a particular purpose, and could not be used for scrolling without disrupting the operation's progress.

If you begin creating or editing a graphical object, then discover that you want to extend the operation to a location not currently visible in the window, move the mouse pointer off the edge of the window in the direction of the location, and the window will scroll automatically. When the desired location is in view, move the mouse pointer back onto the window, and scrolling will stop.

If you move the mouse pointer off the window without intending to autoscroll, but the window begins autoscrolling, the reason is that you have inadvertently begun some editor operation but have not completed it. Complete or cancel the operation, and the autoscrolling will cease to occur.

## Keystroke Shortcuts

Many of the menu options described in the following sections have associated keystrokes that can be used to execute the command without pulling down a menu and making a selection. When a command has an associated keystroke, the keystroke appears next to the menu entry for the command.

Keystroke shortcuts are often very convenient, but due to their brevity they do little to describe the commands they represent, and not all commands have one. In the interest of clarity and consistency, this tutorial always describes explicitly the menu command that performs a particular action, even when a keystroke shortcut is available. Once you have become familiar with the keystroke for a

particular command, feel free to use it rather than pulling down a menu whenever this tutorial asks you to execute that command.

### Creating a New Diagram

So much for theory. Let's start putting some of these ideas into practice.

- Start Design/CPN.
- Choose **N**ew from the **F**ile menu.

A new diagram appears, and displays a blank page called New#1. Take a look at the left side of the status bar: it displays None, because there are currently no graphical objects on the page.

In the rest of this chapter you will create a variety of graphical objects on this page. As you create and manipulate them, the status bar will change in various ways. Some of the objects you will create need to be linked to previously created objects. Therefore, if you have to break off during this chapter, save the diagram, then reopen and continue using it when you return to the tutorial.

### Resetting the Drawing Environment

If at any time you find yourself in a state that does not match the tutorial's instructions and that you don't know how to get out of, reset the drawing environment by doing the following:

1. Press ESC. This cancels any object creation sequence that is in progress.
2. Pull down the **T**ext menu. If its first item is **T**urn Off **T**ext, choose that item. Repeat until the first item in the menu is **T**urn On **T**ext, then leave the menu without choosing anything. This cancels any text-editing operation that may be in progress.
3. Press the DELETE key until a warning appears that "This command cannot be applied when the active page is empty." This deletes any leftover objects.

These steps leave the editor in graphics mode with no operations in progress and an empty page on display. You can now give the instructions that failed another try. Once you understand the information in this chapter, you won't need to clear everything if something goes off course: it will be obvious what went wrong and how to correct it.



- Reset the drawing environment now, to be sure that it is in the state that the following sections assume.

### CAPS LOCK Under X-Windows

Under X-Windows, setting CAPS LOCK changes the behavior of the editing environment in various ways, as described later in this chapter. Except where noted, the instructions in this chapter assume that CAPS LOCK is off. If you suddenly find that the results of following the instructions differ from those that are described, check that CAPS LOCK is still off: a finger slip may have set it.

- Check to be sure that CAPS LOCK is off.

## Working With Rectangles

This section shows you how to create, reshape, move, add text to, and delete rectangles.

### Creating a Rectangle

To create a single rectangle, execute the following steps:

#### Enter Rectangle Creation Mode

- Choose **B**ox from the **A**ux menu
- Move the mouse pointer over the page on which you will draw the rectangle.

The editor enters *rectangle creation mode*; the mouse pointer changes to the *rectangle tool*:



The dot in the tool indicates the specific pixel that the tool points to.

#### Specify the First Corner

- Position the rectangle tool anywhere over the page, then depress and hold the mouse button.

(Generic references to “the mouse button” in this tutorial refer to the left button.)

When you depress the mouse button, the editor does three things:

1. Draws a small square whose upper right corner is at the location of the rectangle tool.
2. Changes the rectangle tool to the *adjustment tool*:



3. Positions the adjustment tool to point at the lower left corner of the square.

The purpose of the adjustment tool is to set or change the shape of a graphical object.

### **Specify the Diagonal Corner**

- Holding the mouse button depressed, move the adjustment tool around on the page.

As the tool moves, the corner at which it points moves with it. This allows you to give the rectangle whatever dimensions you like. You can position the corner in a location that is not currently visible in the window by moving the tool off the window in the direction of the location. The window will autoscroll until you move the tool back onto the window.

- Using the adjustment tool, move the rectangle's corner around the page, then position the rectangle's corner so that the rectangle is a few inches high and wide.

### **Finish the Rectangle**

To finish the rectangle:

- Release the mouse button.

A small editing box appears that just covers the rectangle. The box has scroll bars and a text insertion cursor. You can use this box to write text inside the rectangle. Ignore it for now; such editing boxes will be discussed later in this section.

### **Leave Rectangle Creation Mode**

To leave rectangle creation mode:

- Press ESC.

The rectangle tool is replaced by the graphics tool, and eight small black squares appear on the rectangle, one at each corner and one at the center of each side.

Look at the left side of the status bar. It now displays Auxiliary Node, indicating that the rectangle you just created is not a CPN object, and is a node.

### Reshaping a Rectangle

The black squares on the rectangle you just created can be used to modify the rectangle's shape. They are called *handles*.

- Position the graphics tool over one of the rectangle's handles.
- Depress and hold the mouse button.

The adjustment tool appears next to the handle.

- Move the adjustment tool around on the page.

The handle tracks the tool, and the rectangle changes shape appropriately.

- Release the mouse button.

The graphics tool reappears.

### Moving a Rectangle

A rectangle once created may be moved anywhere on the page:

- Move the graphics tool to the interior of the rectangle.
- Depress and hold the mouse button.
- Move the mouse pointer around on the page.

The rectangle tracks the mouse pointer, keeping the same position relative to it that it had when you depressed the mouse button.

- Release the mouse button.

The rectangle remains in the position to which you have moved it.

### Deleting a Rectangle

To delete the rectangle, just:

- Press the DELETE key.

The rectangle disappears.

### Moving a Rectangle While Creating It

It might be that the point at which you place the initial corner of a rectangle turns out not to be the correct location. You could finish drawing the rectangle, and then move it; or you can move it while you are still drawing it, by using the SHIFT key.

If you press and hold SHIFT while you are using the adjustment tool to define the shape of a rectangle, the rectangle will stop changing shape as you move the tool. Instead it will track the tool by moving as a whole, as if it were tracking the drag tool. When you release SHIFT, the rectangle will go back to changing shape as the tool moves.

- Create a rectangle as described above.
- While setting the rectangle's shape, press and release SHIFT several times, and note how the rectangle responds.
- Make the rectangle a few inches wide and high, for use in the next exercise.
- Release the mouse button.
- Leave rectangle creation mode (ESC).

The graphics tool reappears. The rectangle is now complete, and the handles are visible.

### Adding Text to a Rectangle

You can add textual information to a rectangle by switching from graphics mode to text mode:

- Choose **Turn On Text** from the **Text** menu.

Four things happen:

1. The mouse pointer becomes the text tool.
2. The handles disappear from the edges of the rectangle.

3. A small editing box appears that just covers the rectangle. The box has scroll bars and a text insertion cursor.
4. The status bar displays Text: On.

You can now perform simple text editing operations. You can enter characters via the keyboard, and delete them by pressing DELETE. If you move the text tool onto the rectangle, it will change to an I-beam. The I-beam can be used to position the insertion cursor, and to select sections of text; these sections can be copied, cut, and pasted.

The editing box is part of the X-Windows environment; it is not specific to Design/CPN. Therefore you can use all three mouse buttons to manipulate its scroll bars and edit its text, using the techniques characteristic of the X-Windows user interface.

- Enter and edit text until you are comfortable with the process.
- Choose **Turn Off Text** from the **Text** menu.

The editing box disappears, handles reappear, the graphics tool reappears, and the status bar displays Text: Off. The editor is back in graphics mode. The text you created using the editing box is in the rectangle's text field.

### Creating a Series of Rectangles

You don't have to enter and leave rectangle creation mode once for each rectangle you create. Once you are in it, you can create as many rectangles as you like, by repeating the sequence:

1. Depress and hold the mouse button.
2. Use the adjustment tool to position the diagonal corner.
3. Release the mouse button.

once for each rectangle.

- Enter rectangle creation mode via the **Aux** menu.
- Create several rectangles at various locations.
- Leave rectangle creation mode.

The graphics tool reappears, and handles appear on the rectangle you created last.

### Adding Text to a Rectangle While Creating It

When you create a rectangle, an editing box with scroll bars and a cursor appears over the rectangle after you specify the second corner. This facility exists as a convenience to allow immediate text entry.

- Create a rectangle
- Remain in rectangle creation mode, and enter and edit some text in the editing box.

Examine the status bar while you are entering the text. Note that it displays **Text: On** even though you did not explicitly enter text mode. The editor has automatically entered a special form of text mode, called *creation text mode*, to give you the opportunity to enter text during the rectangle creation process. It will leave creation text mode as soon as you do anything other than enter or edit text, so you don't have to leave it explicitly.

- Leave rectangle creation mode.

Creation text mode ends automatically, because you have done something other than enter or edit text. The editor returns to graphics mode.

### Preserving a Rectangle's Aspect Ratio

A rectangle's aspect ratio is the ratio of its height to its width. It is sometimes useful to preserve the aspect ratio while adjusting a rectangle. For example, suppose the rectangle is a square, and you want to change its size while maintaining its squareness: it would then be inconvenient to have the height and width vary independently during the adjustment process.

To preserve a rectangle's aspect ratio during either the adjustment phase of rectangle creation, or later reshaping of the rectangle, set CAPS LOCK before you begin adjusting its shape, or at any time during the adjustment process. While CAPS LOCK is set, the rectangle's aspect ratio is fixed, and motion of the adjustment tool changes only the rectangle's size. When you release CAPS LOCK, the aspect ratio ceases to be fixed, and again varies with motion of the adjustment tool. You can set and release CAPS LOCK as desired during the shape-adjustment process.

- Set CAPS LOCK.
- Enter rectangle creation mode.
- Create a rectangle.

The rectangle retains its initial square shape as you adjust its shape.

- Release CAPS LOCK.

The aspect ratio now varies with motion of the adjustment tool.

- Complete the rectangle and leave the creation mode.
- Set CAPS LOCK.
- Reshape the rectangle.

The rectangle's aspect ratio is preserved; only the size changes.

- Set and release CAPS LOCK several times as you continue reshaping.

The aspect ratio is preserved whenever CAPS LOCK is set.

## Working With More Than One Object on a Page

When there is more than one graphical object on a page, and you want to edit one of them, you must tell the editor which one, or it will not know where to apply your instructions. The way to give the editor this information is to *select* the object.

The object that is currently selected is called the *current object*. A description of the current object (e.g. Auxiliary Node) is displayed on the left side of the status bar. If the editor is in graphics mode, handles appear around the current object, so you can change its shape. In text mode, an editing box appears on top of it, so you can edit its text field.

When you create a new object, it automatically becomes the current object.

### Selecting an Object

When a page contains only one object, that object is always selected. When there is more than one object, and the object you wish to edit is not selected, click the mouse anywhere on or inside the object of interest. That object is thereby selected, and the object that was selected previously ceases to be selected.

You can select an object whenever the editor is in graphics or text mode. Its appearance (with handles or covered by an editing box) will become the one appropriate to the mode. When you change modes, the current object will change its appearance accordingly.

- Click on various rectangles, and notice what happens when one becomes selected. Move or reshape some of the rectangles.
- Enter text mode, select various rectangles, and notice what happens. Edit some text in some of the rectangles.
- Return to graphics mode.

### Selection After Deletion

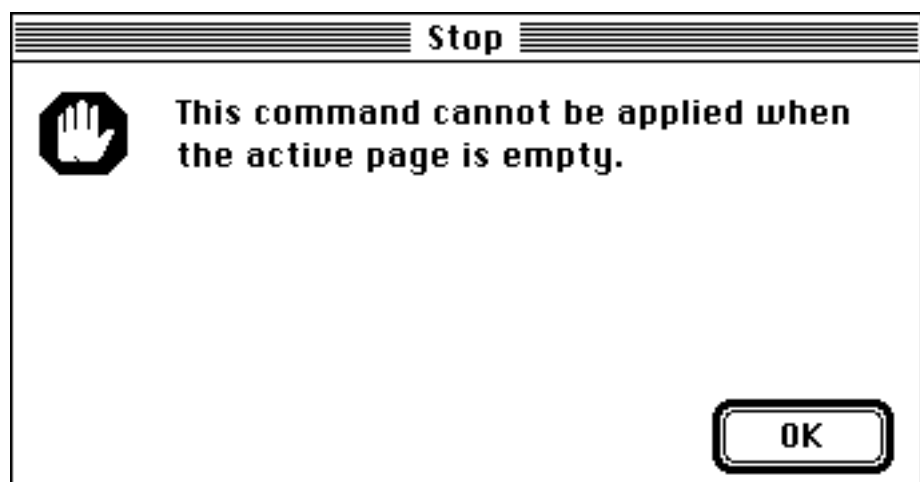
When there is more than one object on a page, the editor keeps them in an order, called the *occlusion order*. When an object is created, it is put at the front of the occlusion order.

The occlusion order is used primarily to control what happens when objects with opaque interiors are drawn on top of each other. The objects you create in this chapter are all transparent, so the occlusion order does not affect their display appearance.

Another use of the occlusion order is to determine what object will become selected when the current object is deleted. The rule is: when an object is deleted, the object at the front of the occlusion order becomes current. This behavior can be useful when you have created several objects in sequence, but then decide to delete them all and try a different approach.

- Press DELETE until no rectangles remain on the page, then press it once more.

A dialog appears:



This response is typical of Design/CPN generally: if you attempt anything that is meaningless or impossible (such as trying to delete



an object that is not there), it refuses to carry out the attempt, and instead warns you of the problem.

## Working With Ellipses

Working with ellipses is very similar to working with rectangles. This section shows you how to create, reshape, move, and delete ellipses.

### Creating an Ellipse

To create a single ellipse, execute the following steps:

#### Enter Ellipse Creation Mode

- Choose **Ellipse** from the **Aux** menu.
- Move the mouse pointer over the page on which you will draw the ellipse.

The editor enters *ellipse creation mode*; the mouse pointer changes to the *ellipse tool*:



#### Specify the First Corner

- Position the ellipse tool anywhere over the page, and depress and hold the mouse button.

The editor draws a small circle that fills an invisible square whose upper right corner is at the location of the ellipse tool, switches to the adjustment tool, and positions it to point at the lower left corner of the invisible square.

#### Specify the Diagonal Corner

- Holding the mouse button depressed, move the adjustment tool around on the page.

As the tool moves, the invisible corner at which it points moves with it, and the ellipse changes shape so that it always fills the invisible rectangle. Autscrolling works as with rectangles.

- Using the adjustment tool, give the ellipse whatever appearance you like.

### **Finish the Ellipse**

To finish the ellipse:

- Release the mouse button.

The ellipse tool reappears, and an editing box appears over the ellipse. It can be used to write text inside the ellipse.

### **Leave Ellipse Creation Mode**

- Press ESC.

The ellipse tool is replaced by the graphics tool, and eight handles appear on the invisible rectangle that the ellipse fills. These can be used to reshape the ellipse.

## **Other Operations With Ellipses**

All the operations that work with rectangles work with ellipses in exactly the same way. The reason ellipses and rectangles are so similar is that an ellipse is completely defined by the rectangle that contains it.

CAPS LOCK preserves an ellipse's aspect ratio just as it does that of a rectangle.

- Perform with ellipses all of the operations that you performed with rectangles.

## **Working With More Than One Object Type**

When a page contains objects of more than one type, they are all kept in a single occlusion order.

You don't have to leave one object creation mode before you enter another. You can go directly from one to another by choosing the new mode while you are still in the old one.

- Create a mixture of rectangles and ellipses by going directly from one creation mode to the other and back again.

### Creating Objects From Text Mode

In the exercises so far, the editor was always in graphics mode when you entered a creation mode, and therefore returned to graphics mode when you exited the creation mode.

You can also enter a creation mode from text mode. While the editor is in the creation mode, it makes no difference that it got there from text mode. The only difference is that when you leave the creation mode, the editor will return to text mode, since that is where it started. The editor's automatic activation and deactivation of creation text mode during object creation is orthogonal to the underlying text mode.

- Enter text mode.
- Enter ellipse creation mode.
- Create an ellipse.

The editor has automatically entered creation text mode.

- Enter some text.
- Exit the creation mode.

The editor leaves creation text mode, because you have done something other than enter or edit text, and returns to text mode. The editor is now in the same state that it would have been in if you had created the object from graphics mode, returned to graphics mode, then entered text mode from graphics mode.

- Edit the text you entered in creation text mode.
- Return to graphics mode.

### Working With Connectors

Objects of the types we have worked with so far can exist independently of other objects. In Design/CPN parlance, such objects are called *nodes*.

A *connector* is an arrow that runs between one node and another, establishing a directed link between them. A connector cannot exist independently: it can exist only when there are two nodes for it to connect. Every Design/CPN auxiliary object is either a node or a connector that links two nodes.

This section shows you how to create, reroute, edit, and delete connectors. In general, the methods are the same as for other objects.

You should have various rectangles and ellipses (nodes) scattered around on the page. If you do not, create some now. Be sure the editor is in graphics mode before you proceed.

### Creating a Connector

To create a connector:

- Choose **Connector** from the **Aux** menu.

The editor is now in *connector creation mode*, and the mouse pointer is the *connector tool*:



- Position the connector tool in the interior of some node.
- Depress and hold the mouse button.
- Move the connector tool to the interior of some other node.
- Release the mouse button.

A connector now points from the first node to the second. The editor is still in connector creation mode, so you can create additional connectors.

Look at the left side of the status bar: it displays Auxiliary Connector, because that is the type of the object you have just created.

### Routing a Connector

When there are many objects on a page, the best route for a connector may not be a straight line. You can make a connector follow any path you want.

- Position the connector tool inside a node not used so far, and depress and hold the mouse button.
- Move the tool to a location not inside any node, and release the mouse button.
- Move the tool around without depressing the mouse button.

The location at which you released the mouse button becomes a fixed point. As you move the tool, the connector tracks it, extending between the fixed point and the current connector tool location.

- Click the mouse somewhere outside any node.

Another fixed point results. You can repeat this sequence as often as you like, creating an arbitrarily complex connector. The fixed points are called *vertexes*. The lines between them are called *segments*.

- Create several more vertexes and segments.
- Click the mouse inside some node other than, and not connected to, the node where you began.

A convoluted connector now points from the first node to the second.

- Create several more connectors.
- Leave connector creation mode.

The editor is back in graphics mode. The last connector you created is the current object, as with any newly created object. Handles appear at each vertex, in the center of each segment, and at each end of the connector you just created.

### Editing a Connector

To reroute a connector, drag its handles. If you drag the point at the center of a segment, resulting in a new vertex and two new segments, new handles appear on the two new segments.

You can use the handles at either end of a connector to detach it from one node and attach it to another: just drag the handle to the edge or interior of the node you want to connect to.

- Reroute and reattach connectors until you are familiar with the process.

You can edit connectors in most of the ways that you can edit objects generally. You can make a connector current in text mode and add text to it, delete it with DELETE, etc.

- Perform some object editing operations on connectors.

### Automatic Rerouting of Connectors

If you move a node that has an attached connector, the connector automatically reroutes itself so that it remains connected to the node.

- Move some nodes that have connectors, and note how the connectors change.

No algorithm for automatic rerouting can give ideal results in all cases, so you will sometimes need to manually adjust a connector's route after automatic rerouting has occurred.

### Deletion of Dangling Connectors

If you delete a node that has any attached connector, the connector no longer has a node at each end. Such a connector is illegal; the editor deletes it automatically.

- Delete a few nodes that have connectors.

### Working With Labels

A label is a node that consists entirely of text; it has no associated geometric form. For editing purposes, a label is no different from the text field that is associated with every rectangle or ellipse. Its lack of an accompanying geometric form is its only unusual property.

### Creating a Label

To create a label, execute the following steps:

#### Enter Label Creation Mode

- Choose **Label** from the **Aux** menu.
- Move the mouse pointer over the page on which you will draw the label.

The editor enters *label creation mode*; the mouse pointer changes to the *label tool*:



#### Create the Label

- Position the label tool anywhere over the page, then click the mouse.

A text insertion cursor appears at the location where you clicked the mouse.

### Enter and Edit Text

You can now enter and edit text just as if the editor were in text mode, and you were editing the text field of a rectangle or ellipse.

- Type and edit several short lines of text.

Note that there are no scroll bars. Since a label consists only of its text, there is no frame in which the text must fit, and hence no need for scrolling.

### Create Additional Labels

- Create a few more labels at various locations on the page.
- Leave label creation mode.

The editor is back in graphics mode.

### Other Operations With Labels

As with other objects, you can create as many labels in succession as you like. To edit a label's text after the label has been created, select the label in text mode. You can either select it first or enter text mode first.

For most purposes, labels are the same as rectangles and ellipses, and can be treated in the same way. The only differences are:

1. You can't move a label while you are creating it, because its creation is a one-step process.
2. You can't reshape a label in graphics mode, since it has no geometric form: its shape is just the shape of the text it consists of, and changes only when you edit that text.
3. An empty label would serve no purpose, so the editor deletes any label that contains no text.

With these three points in mind:

- Create various labels and do various things with them until you feel comfortable with them. Intermix them with ellipses and rectangles, and be sure that you are clear on the similarities and differences among the various types.

### Nodes and Regions

It is frequently useful to define a node as being logically subordinate to some other object. Such a subordinate node is called a *region*. A node that is a region does not cease to be a node. It just takes on some additional properties that make it a region as well.

An object may have any number of regions. These may in turn have regions, sometimes called subregions, and so on indefinitely. An object and any associated regions always form a tree; non-tree-structured relationships may not be defined. Objects and their regions are referred to as parents and children, as is customary with tree structures.

When a node is a region, it is affected not only by operations performed on itself, but also by operations performed on its parent. For example, when a region's parent is deleted, the region is automatically deleted as well. Details appear later in this section.

Both nodes and connectors can have regions, but only a node can be a region: a connector cannot be. The reason is that allowing a connector to be a region would lead to unresolvable contradictions between its role as a link and its role as a subordinate object.

### Designating a Region

You should have a variety of nodes and connectors on the page left over from previous exercises. If not, create some now.

You can designate a node to be a region of an object whenever the editor is in graphics mode or text mode. The operation of designating a region does not affect the editing mode.

To designate a node as a region:

- Select the node that will be the region.
- Choose **Make Region** from the **Aux** menu.

The status bar displays Select Parent, and the mouse pointer becomes the *pointer tool*:



- Move the tool so that it is on or inside the object that is to be the parent of the region. (Any node or connector will do at this point.)



A dark border flashes on and off around the prospective parent. An object that could not legally become the parent, because the result would not be a tree, would not show such a border.

- Click on the object that is to be the parent.

The selected node is now a region of the object you designated as its parent.

- Create some more regions. Make some of them be subregions of other regions.

You can execute **Make Region** on a node that is already a region. It will cease to be a region of its current parent, and become instead a region of the newly designated parent.

- Transfer a region from one parent to another.

### Restoring the Independence of a Region

You can disconnect a region from its parent, restoring its independent status, whenever the editor is in graphics or text mode. If the region has subregions, their status relative to the region is not affected by its disconnection.

- Select a region that is to be returned to independent status.
- Choose **Make Node** from the **Aux** menu.

The region is returned to independent status (note the change in its status bar description). It has the same properties that it would have if it had never been a region.

### Editing Parents and Regions

Editing a region does not affect its parent, but editing the parent can affect the region in a number of ways:

#### Moving a Region's Parent

When a region's parent is a node, and the parent is moved, the region moves with it so that they retain the same relative positions.

- Establish a chain or tree of regions.
- Experiment with moving parent nodes.

When a region's parent is a connector, the situation is complicated by the fact that a connector (unless it is a straight line) has no center

point that can be taken as its position. Consequently there is no way to define how rerouting a connector would move a region. Rerouting a connector therefore does not affect region position.

The position of a connector is defined as the midpoint of a straight line connecting its endpoints. When that point moves, because one of the connected nodes moves, any regions will move also.

- Establish a region whose parent is a connector.
- Experiment with rerouting the connector and with moving the nodes it connects.

### Deleting a Region's Parent

When a region's parent is deleted, the region is automatically deleted along with it.

- Delete a parent object and note the results.
- Delete a node that is linked to some other by a connector that has a region.

The deletion of the node deletes the connector, and the deletion of the connector deletes the region.

## Groups of Objects

It is frequently necessary to perform the same operation on more than one object, and it would be inefficient to have to select and then operate on them one at a time. Therefore the editor allows you to designate a *group* of objects and operate on them simultaneously. There is then no one selected object: all the objects in the group are selected.

When a group has been selected, the editor indicates the group by drawing a gray border around each of its members. Handles do not appear around group members, since they intrinsically refer only to individual objects.

Whenever you select more than one object simultaneously, the editor enters *group mode*. This mode is not an alternative to graphics or text mode; it is orthogonal to them.

When the editor is in group mode, the mouse pointer becomes the *group tool*:



This tool is displayed whether the editor is also in graphics or text mode. Examine the status bar to tell which mode is in effect.

### Mixed Groups

The properties of nodes, connectors, and regions are so different that it is not possible to create a useful group that combines objects from more than one of these classes. The reasons are:

1. Whenever a node is changed, any connectors and regions associated with it automatically adjust themselves. Therefore there is never any reason to include connectors and regions in a group that contains nodes.
2. Allowing connectors and regions to be part of the same group could lead to unresolvable conflicts among the various operations that are performed on them automatically when their associated nodes are edited.

Therefore when more than one object is selected at a time, they must all be nodes, or all connectors, or all regions.

### Selecting a Group

There are many ways to select a group. None of them can be used to create a mixed group.

- The **Group** menu contains the commands **Select All Nodes**, **Select All Regions**, and **Select All Connectors**. These have the obvious effects. They may be combined with mouse and keyboard techniques to further specify the group.
- Pressing and holding **SHIFT** and then clicking on an object selects it (if it was not selected and is of appropriate type) or deselects it (if it was selected) without affecting the selection status of other objects.
- Depressing the mouse button when it is not inside an object, then moving it, creates a rectangular selection area. When the mouse button is released, all objects in the area (that are not selected and are of appropriate type) will become selected, and any other selected objects will cease to be selected.
- Pressing and holding **SHIFT**, then using the mouse to create a rectangular selection area, toggles the selection status of all

objects in the area (except that none becomes selected that is not of appropriate type) without affecting that of other objects.

Whenever a group rather than a single object is current, the status bar displays a description of the content of the group.

- Experiment with creating and manipulating groups in various ways, in both graphics and text mode. Note the status bar descriptions of the groups you create.

### Deselecting a Group

When the editor is in group mode, the command **Ungroup** is available in the **Group** menu. If you choose this command, the group will cease to be selected, and the object that was current before you entered group mode will become current again.

A group is also deselected if you select some other group in its place, or use the mouse to select a single object that is not a member of the group.

### Reconstructing a Group

Sometimes a particular group will be useful more than once during an editing session, but is not useful continuously. To save you the effort of reconstructing such a group each time you need it, the editor always records the composition of the group that is current when you leave group mode. It can use this information later to reconstruct the group.

When the editor is not in group mode, the command **Regroup** is available in the **Group** menu. If you choose this command, and you had previously defined a group at some time, the editor will re-enter group mode and reselect the group.

- Create a group, then deselect and reselect it by toggling group mode.
- Find out what happens if you move or delete group members while not in group mode, and then reenter the mode.

### Operating on Groups

You can do anything to a group of objects that:

1. Could be done to each of its members individually, and:

2. Is meaningful when applied to more than one object simultaneously.

For example, you can move a group, delete a group, make all the members of a group regions of the same parent, or turn a group of regions back into nodes even if they had different parents. But you cannot edit the text in a group of nodes, or reroute a group of connectors, since there is no way to deal with text fields or connector routes generically.

- Define various groups and operate on them until you are familiar with what you can and cannot do with them.

There are also operations that are meaningful for groups but not for the individuals within them. Most of them concern the physical layout of the group's members on the page, and are used to improve the physical appearance of a net.

## Intermission

- Close the diagram you have been working on. This tutorial doesn't make any further use of it, but you might want to save it to examine later as a reference.
- Quit Design/CPN.
- Take a break.

There's lots more to come.



# Chapter 5

## CP Net Components

As we noted in Chapter 3, a CP net is a graphical structure with associated computer language statements. The principal components of a CP net are:

- **Data:** CP nets make use of datatypes, data objects, and variables that hold data values. CP net data is defined using a computer language called *CPN ML*.
- **Places:** Locations for holding data.
- **Transitions:** Activities that transform data.
- **Arcs:** Connect places with transitions, to specify data flow paths.
- **Input Arc Inscriptions:** Specify data that must exist for an activity to occur.
- **Guards:** Define conditions that must be true for an activity to occur.
- **Output Arc Inscriptions:** Specify data that will be produced if an activity occurs.

This chapter defines all of these components, and shows how they interrelate syntactically to form a CP net. This chapter does not discuss how they work together dynamically to define a CP net's behavior when it is executed. That is covered in Chapter 7.

### The CPN ML Language

General descriptions of data flow suffice for a static paradigm such as IDEF0. A dynamic paradigm like CPN needs more: it must include the representation of actual data, with clearly defined types and values. The presence of data is the fundamental difference between dynamic and static modeling paradigms. All of the other differences exist to allow data to be manipulated in clearly defined ways, or result from the possibility of doing so.

## Design/CPN Tutorial

---

CP nets allow data to be of any type that can be defined on the computer. In order to define and manipulate such data, CP nets use computer language statements.

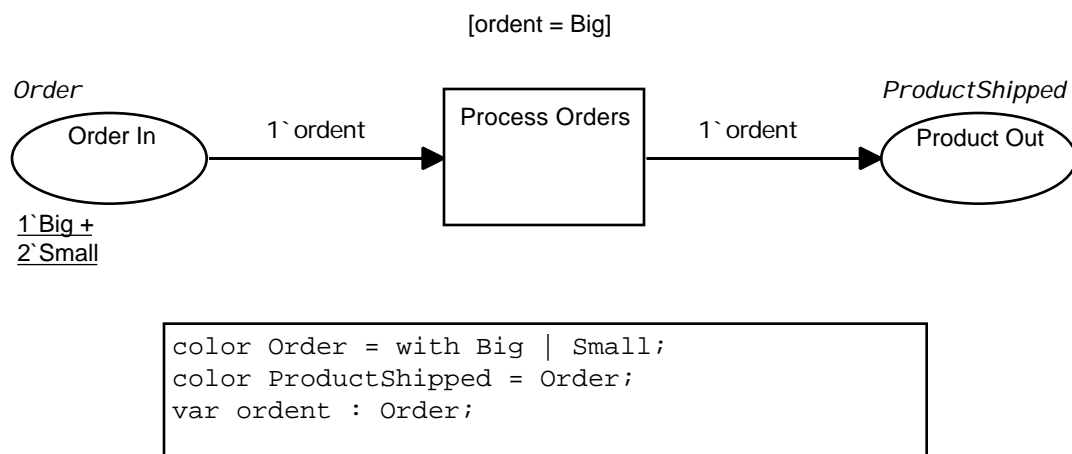
CP nets require an extremely rigorous approach to data. A weakly typed language like LISP, or even a strongly typed language that automatically coerces types as C does, would not provide the absolutely unambiguous approach to data that a CP net needs. The language ML (for Meta Language, no relation to Meta Software) is particularly well suited for use by CP nets. It is a strongly typed functional language that provides great economy of expression, is easily extended by the user, and can execute interpretively or be compiled.

To facilitate the use of ML with CP nets, Meta Software has added various extensions, resulting in the language CPN ML (for Colored Petri Net ML). Very little of ML is ever needed for working with CP nets, so in practice CPN ML is a very small language.

You do not need to become a CPN ML programmer in order to use CP nets. The CPN equivalent of programming is done by creating graphical net structure, not by writing language statements. You need only learn the particular syntactic conventions through which CPN ML does various low-level things that any computer language does: declaring datatypes and variables, comparing one data value with another, and so on. Information on these conventions will be presented as needed throughout this tutorial.

### A CP Net Example

The various CP net components will be discussed in the context of the small CP net that appeared in Chapter 3. This net is called FirstNet.





As each CP net component is discussed, the instance(s) of it in FirstNet will be shown in boldface (for text) or with thickened borders (for graphics).

### Nets and Models

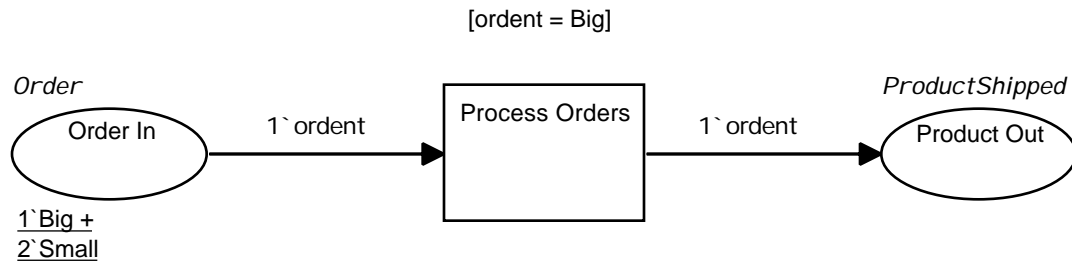
A CP net exists in and of itself as a syntactic structure. There is no requirement that a net be a model of anything in particular. A net that is entirely meaningless, if it is syntactically correct, is just as much a net as one that constitutes a useful model. The relevance of a CP net to a system, by virtue of which the net functions as a model, really exists only in our minds: the net itself is just a structure that executes according to certain rules.

FirstNet is not very interesting as a model, because it does not define any structure or behavior that is complex enough to justify a modeling effort, and what little structure and behavior it does define is unrealistic. Nevertheless, it contains most of the constructs that are used by CP nets of any size, and can be used to demonstrate the essential algorithm by which CP nets execute. Once you understand these matters in the minimal context of FirstNet, you should have little difficulty applying them to much more interesting nets

### CP Net Data

CP nets use datatypes, data objects, and variables. CPN datatypes are called *colorsets*. CPN data objects are called *tokens*. Tokens are somewhat like objects in an object-oriented programming system. If you are familiar with object-oriented programming, you are ahead of the game, but such familiarity is not necessary.

All CPN datatypes and variables must be declared. They are declared in a declaration box called a *global declaration node*. FirstNet's global declaration node contains three declarations:



```
color Order = with Big | Small;  
color ProductShipped = Order;  
var ordent : Order;
```

The meanings of these declarations are explained in this section.

### Colorsets

CP net tokens can be of all the datatypes generally available in a computer language: integers, reals, strings, booleans, lists, tuples, records, and so on. In the context of CP nets, these types are called colorsets.

“Colorset” is really just a synonym for “token datatype.” The term “colorset” is used, rather than a standard term such as “datatype,” to avoid confusion between token datatypes, which are extensions to standard ML, and ordinary ML datatypes. The origin of the term “color” was described in Chapter 1.

Just as every piece of data in an ordinary computer program is of some datatype, so every token in a CP net is of some colorset. All colorsets used in a CP net must be explicitly declared. The declaration syntax is:

```
color name = definition;
```

where name is the name of the colorset, and definition specifies what it consists of. The syntax for definition varies, depending on what sort of colorset is being declared.

### Enumerated Colorsets

One common colorset defines tokens that may have any of a set of predefined literal values. The possible values are all enumerated in the colorset declaration, so this colorset is called an *enumerated colorset*. The syntax of an enumerated colorset declaration is:

```
color name = with value { | value }...;
```

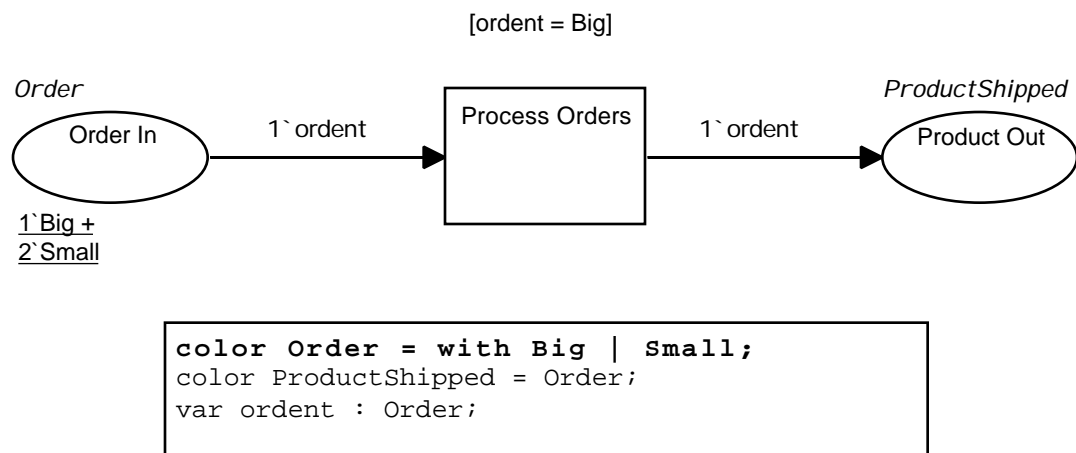
The construction “{ } . .” indicates that the material in braces may be repeated zero or more times. Underlining indicates a term that is to be replaced with an actual value.

Examples:

```
color oneval = with EXISTS;  
color booltok = with yes | no;  
color chord = with Major | Minor |  
Augmented | Diminished;
```

The first declaration defines an enumerated colorset named `oneval`. All tokens of this colorset have the same value: `EXISTS`. The second defines an enumerated colorset `booltok`. Every token of type `booltok` has either the value `yes` or the value `no`. The third declaration defines `chord`, whose tokens may have any of the values `Major`, `Minor`, `Augmented`, or `Diminished`. All of these names and values, and all CPN ML textual objects, are case-sensitive.

FirstNet uses an enumerated colorset, `Order`:



The declaration indicates that `Order` tokens may have either of the values `Big` or `Small`. Thus FirstNet provides the ability to model orders that are of two kinds: “big” orders and “small” orders. This is a fairly coarse granularity, but it will suffice for now.

Mnemonic names such as `Order`, `Big`, and `Small`, are very helpful in making a model easy to understand, but their meaningfulness has no syntactic significance. Syntactically, we could just as well declare:

```
color A = with B | C;
```

If we used `A`, `B`, and `C` just as we would use `Order`, `Big`, and `Small`, the model would be functionally the same, but of course would be harder to make sense of.

## String and Integer Colorsets

Strings and integers are commonly used in CP nets. The syntax for a string colorset declaration is:

For example:

```
color name = string;
```

The syntax for an integer colorset declaration is:

```
color name = int;
```

It is possible to restrict the length and character set of a string colorset, and the range of an integer colorset, by adding clauses to the declaration.

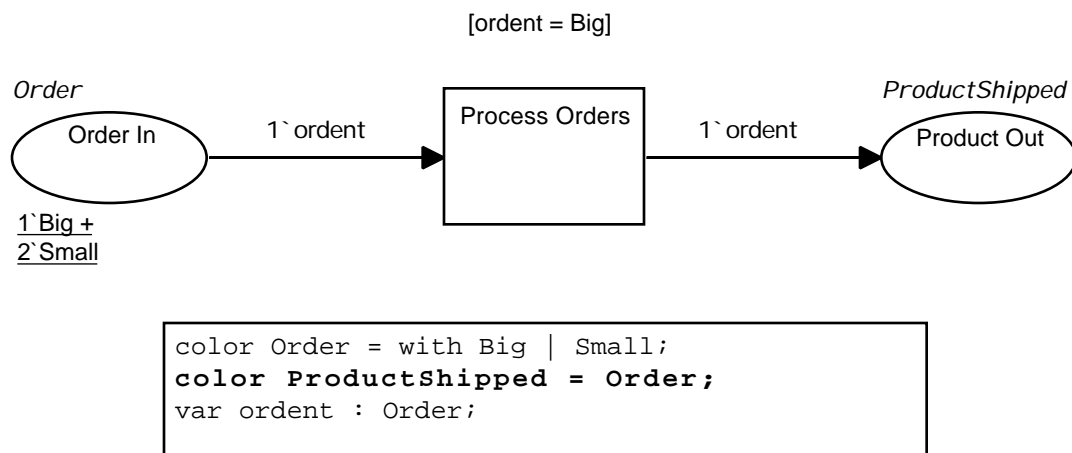
The syntax for other colorsets, including composite colorsets, will be given as needed in later chapters. Complete information on CPN ML colorset declaration appears in *The Design/CPN Reference Manual*.

## Duplicate Colorsets

It is often useful, both in programming and in modeling (which is really just a kind of programming), to have more than one datatype that consists of the same values. In CPN ML, such a datatype is called a *duplicate colorset*. The declaration syntax is:

```
color DuplicateName = ExistingName;
```

where DuplicateName is the name of the duplicate type being defined, and ExistingName is the name of some colorset that has already been declared. FirstNet declares a duplicate colorset called ProductShipped:



`ProductShipped` tokens may have the same values that `Order` tokens can: `Big` or `Small`.

Again, the granularity is fairly crude. The level of realism is also fairly low: big and small orders make some sense, but shipped products would probably be categorized differently. For example it might be more useful to have heavy and light shipped products, and use the distinction to determine how the products should be shipped. But our goal now is to study CP net components, not to do realistic modeling: greater realism at this early stage would just add complexity.

### Tokens

CPN data objects are called *tokens*. They are called tokens rather than objects to avoid confusion with graphical objects. Every token used in a CP net must be of one of the colorsets declared for the net.

A CP net token is represented by giving its value. For example:

```
Xenia  
"Clifton"  
45387
```

The first example specifies a token of an enumerated colorset. The token's value is `Xenia`. The second specifies token of a string colorset (a string token) with the value `"Clifton"`. The third specifies an integer token; its value is `45387`.

Note that when there is more than one colorset that includes some possible value, there is no way to tell by looking at a token which colorset the token belongs to. This doesn't create ambiguity, because the way tokens are used in CP nets always indicates the correct colorset contextually.

### Multisets of Tokens

In dealing with CP nets, it is often necessary to manipulate and refer to collections of tokens. A collection of zero or more tokens is called a *multiset*.

All tokens in a multiset must be of the same colorset. A multiset may contain multiple tokens that have the same value. Any subset or union of multisets is again a multiset.

A multiset is not a data structure that exists separately from the tokens it contains. It is just a convenient way to refer to a collection of tokens. Thus a single token and a multiset of one token are the same thing. A multiset of no tokens has no properties of its own, so there is only one such multiset, called the *empty multiset*. When the empty multiset must be designated explicitly, it is written as `empty`.

### Specifying Multisets

A multiset is specified by giving an expression that describes the multiset. The simplest such expression specifies a multiset of identical tokens. Such an expression is called a *multiset designator*.

A multiset designator consists of an integer denoting the number of tokens in the multiset, followed by a backquote (`), followed by the value of the tokens:

```
count`value
```

For example:

```
1`Xenia
3`"Clifton"
2`45387
```

The first example specifies a (multiset containing a) single token of an enumerated colorset. The token's value is `Xenia`. The second specifies a multiset of three string tokens, each with the value `"Clifton"`. The third specifies a multiset of two integer tokens, each with the value `45387`.

### Multiset Addition

Multisets of the same colorset may be added:

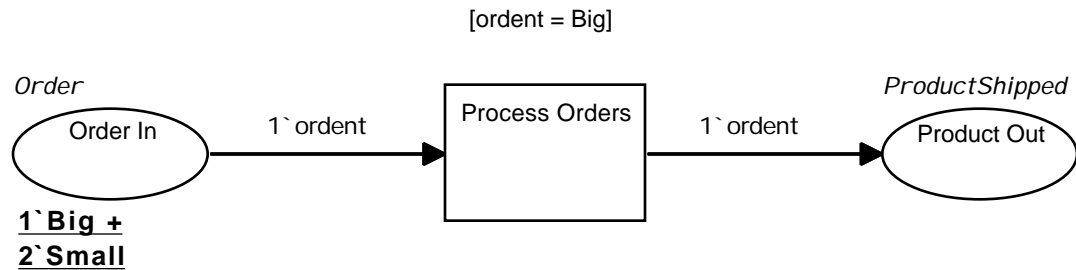
```
1`"Springfield" + 3`"Springfield" =>
4`"Springfield"
```

A multiset containing tokens with different values is specified by giving an expression that is a sum of multiset designators. Thus:

```
7`67 + 7`3 + 3`1
```

is a multiset expression. It specifies a multiset of 17 integer tokens: seven with the value 67, seven with the value 3, and three with the value 1. (The same multiset would result if the terms were added in any other order.)

FirstNet contains a multiset expression:



```

color Order = with Big | Small;
color ProductShipped = Order;
var ordent : Order;
  
```

The expression specifies one token with value *Big*, and two with value *Small*. Its significance will be explained as we proceed.

## Multiset Subtraction

Multisets of the same colorset may be subtracted:

```

5`"Clifton" - 3`"Clifton" => 2`"Clifton"
2`Gegner - 2`Gegner => empty
  
```

The empty multiset may be subtracted from any multiset:

```

1`513 - empty => 1`513
  
```

Multiset subtraction may not attempt to take away tokens that do not exist. Therefore:

```

2`10 - 3`5
1`"Helen" - 1`"Glen"
empty - 2`Birch
  
```

are not valid subtractions, and would result in an error.

## Multiset Subsets

When all the elements of one multiset also exist in a second multiset, so that the first could be subtracted from the second, the second is a *subset* of the first. (There is no need to call it a “submultiset”, because the context always prevents confusion with ordinary sets and subsets.) By definition, the empty multiset is a subset of every multiset.

Thus the subsets of the multiset:

```

1`Land + 2`Sea
  
```

## Design/CPN Tutorial

---

are:

```
1`Land + 2`Sea
1`Land + 1`Sea
1`Land
2`Sea
1`Sea
empty
```

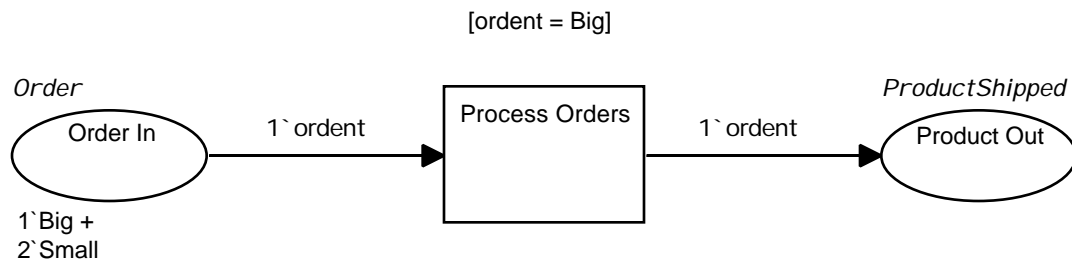
### CPN Variables

CP net execution often requires token values to be accessed and used in various ways. Such usage is no different than the manipulation of data values that occurs in ordinary computer programs.

It would not work to restrict a CP net to handling only token values that are known in advance. This would be equivalent to requiring a computer program to use only constants. Therefore CP nets contain variables that can take on token values as needed. These variables are called *CPN variables*.

Like colorsets, CPN variables must be explicitly declared. Every CPN variable is of a particular colorset, and can take on only values of that type. When a CPN variable has taken on the value of some token, it is said to be *bound* to that value. Often a CPN variable is not bound to any value, and is said to be *unbound*.

FirstNet uses one variable: `ordent`:



```
color Order = with Big | Small;
color ProductShipped = Order;
var ordent : Order;
```

Since `ordent` is of type `Order`, it can be bound to either the value `Big` or the value `Small`. Those are the only values it can ever have.



## Places

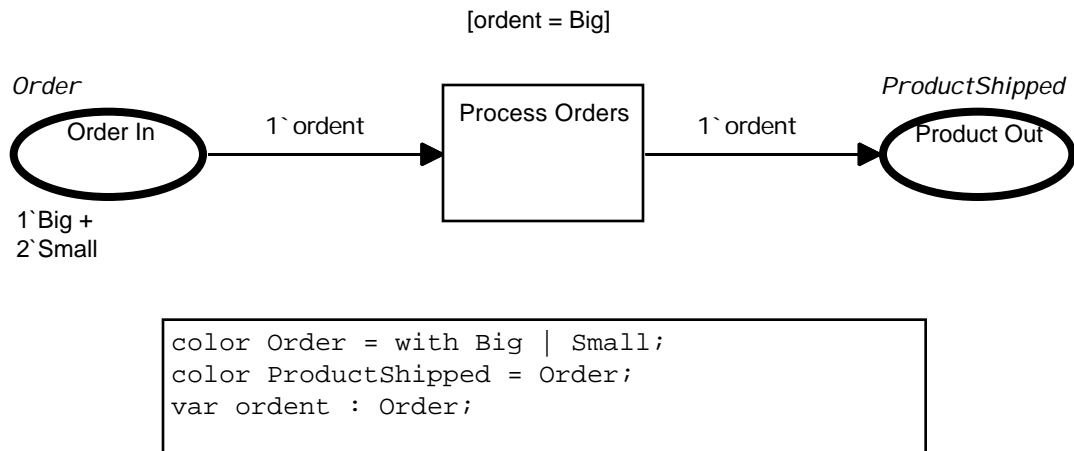
Tokens would be of little use if they could not be stored and accessed as needed. CP nets keep tokens in locations called places.

A *place* is a location that can contain zero or more tokens (a multiset) of some particular colorset. A place's colorset is one of the colorsets defined for the net. All tokens in a place must be of that colorset.

A place may optionally have a name. Such a name is useful for indicating what the place means as a part of a model, for identifying the place when humans confer about the net, and for labeling computer-generated information about activity in the place during simulation runs.

A place is graphically represented as an ellipse. Its name and/or colorset may optionally be displayed next to or inside the place. They are graphically represented as labels that are regions of the place. The regions are named for their contents: a place's name is kept in a *name region*, its colorset in a *colorset region*, etc.

FirstNet contains two places:



The place on the left is named `Order In`. Its colorset is `Order`. The place on the right is named `Product Out`; it is of type `ProductShipped`.

The locations, fonts and styles shown for the various place regions are fairly typical, but there is nothing special about them. Design/CPN allows you to position regions wherever you like, and to put them in any fonts and styles available on the computer. You can also make a region invisible; this is often useful to avoid visual clutter resulting from regions that must exist but are not currently of interest. The same is true of all CP net regions.

## Place Markings

The purpose of a place is to hold tokens. The tokens in a place, like any group of tokens, constitute a multiset. The multiset in a place is called the *marking* of the place. A place always has a marking: if it contains no tokens, its marking is the multiset `empty`.

When a place is empty, no marking is shown for it. (An explicit `empty` could be written, but that would serve no purpose.) The marking of a nonempty place is depicted as a circle with a number inside indicating the number of tokens in the multiset, followed by an expression that describes its contents:

① `1`"Xenia"`

④ `4`45387`

①⑦ `7`67 + 7`3 + 3`1`

## States and Markings

Taken together, the markings in all the places in a CP net constitute the *state* of the net. The first state of the net, before execution begins, is called the *initial state* of the net, and the place markings that constitute it are called the *initial markings* of the places.

As a net executes, tokens are put into and removed from places, so that the state of the net changes. Its state at a given time is called its *current state*, and the place markings that constitute that state are the *current markings* of the places. Thus a net's initial state is the first of a succession of current states, and a place's initial marking is the first of a succession of current markings.

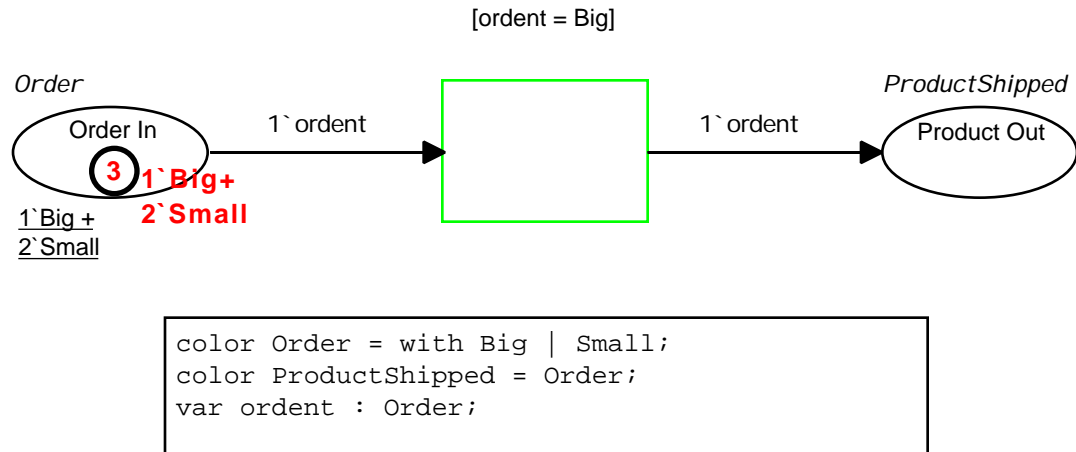
## Initial Marking Regions

In `FirstNet`, `Order In` has an associated multiset expression: `1`Big + 2`Small`. This is another region of the place, called the *initial marking region*. When `FirstNet` is executed, the tokens specified in this region will be put into the place `Order In` before execution starts, providing `Order In`'s initial marking.

`Product Out` has no initial marking region, so no tokens will be put in it before execution starts. Put another way, its initial marking is `empty`.

## Appearance of Markings

FirstNet with initial markings shown looks like this:



In this figure, the place `Order In` contains three tokens, one of value `Big` and two of value `Small`, as specified by the initial marking region. These tokens constitute the initial marking of the place, and are its current marking as well.

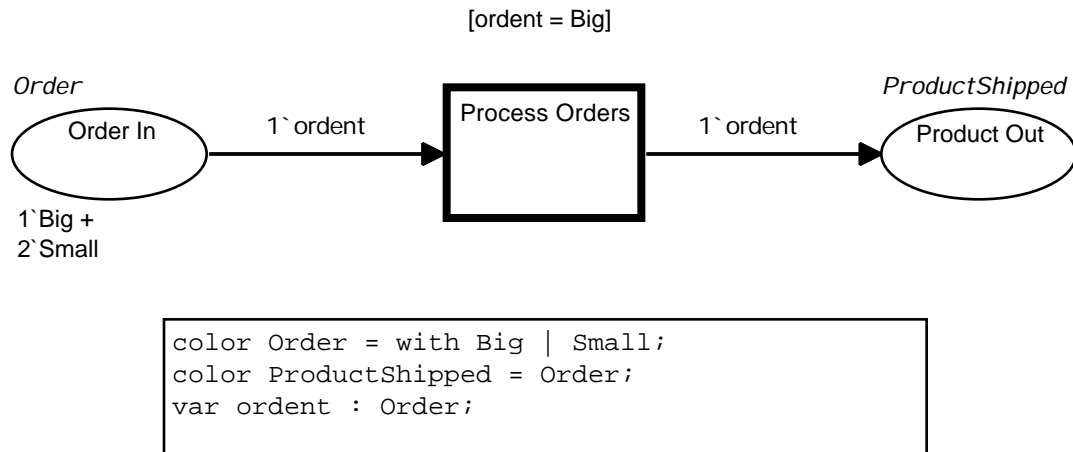
The existence of both an initial marking region and a current marking for a place is not a redundancy. The initial marking region is just a label with some text in it that specifies some tokens that will be put in the place before execution starts. The current marking represents the actual tokens.

A place's current marking may be depicted anywhere that is convenient. Placing it as shown above is just a convention, used because it gives a good appearance in this case.

## Transitions

A CP net *transition* is an activity whose occurrence can change the number and/or value of tokens in one or more places. A CP net transition may optionally have a name, kept in a name region, which serves the same purposes as a place name. A transition is graphically represented as a rectangle. If it has an associated name, it may optionally be displayed next to or inside the transition.

FirstNet has one transition: `Process Orders`:

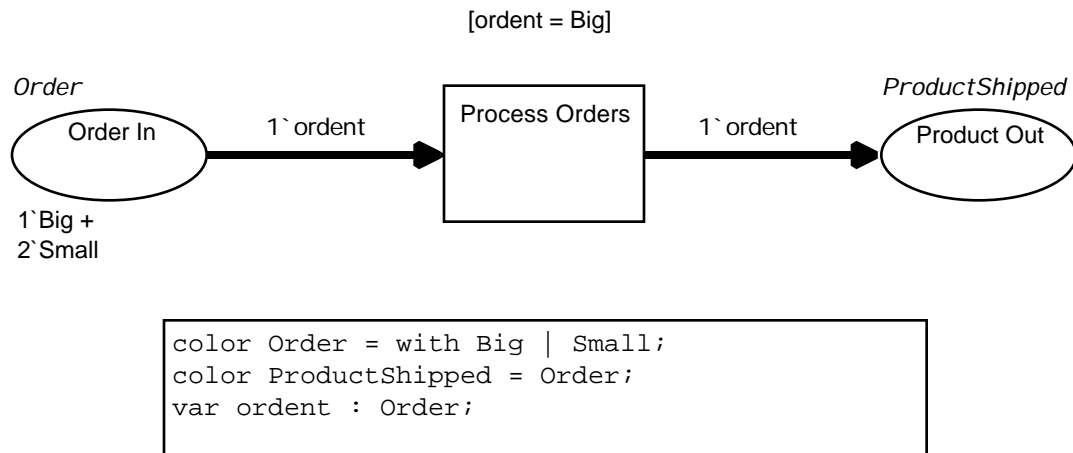


We'll look at exactly how transitions work in Chapter 7.

## Arcs

An *arc* is a connection between a place and a transition. Every arc connects one place with one transition. Every arc has a direction, either from a place to a transition, or from a transition to a place.

An arc is represented graphically as an arrow with a head that indicates its direction. FirstNet has two arcs:

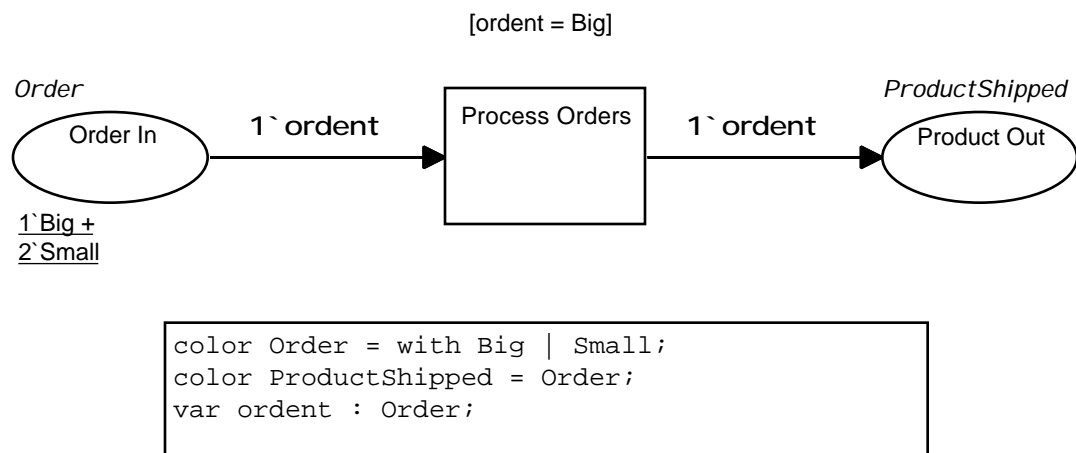


An arc that runs from a place to a transition is called an *input arc*, and the place it connects to is called an *input place*. An arc that runs from a transition to a place is called an *output arc*, and the place to which it connects is called an *output place*. Thus in FirstNet, `Order In` is an input place, and `Product Out` is an output place. It is possible for a place to be both an input place and an output place.

When a transition occurs, it may remove tokens from any or all of its input places, and put tokens into any or all of its output places. The number and values of tokens removed and added are determined by arc inscriptions and guards.

### Arc Inscriptions

An *arc inscription* is a multiset expression associated with an arc. It is kept in a region of the arc called an *arc inscription region*. FirstNet has two arc inscriptions:



An arc inscription on an input arc is called an *input arc inscription*; an arc inscription on an output arc is called an *output arc inscription*.

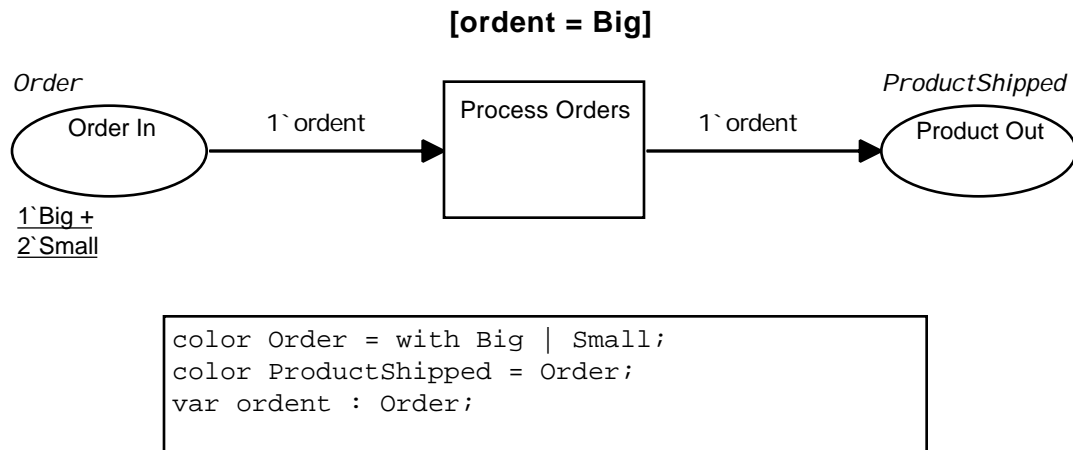
The tokens in the multiset specified by an input arc inscription must be present in the input place in order for the transition to occur (there may be other tokens there also). These tokens will be removed from the input place if the transition does occur. The multiset specified by an output arc inscription will be put into the output place if the transition occurs. Details on how arc inscriptions specify multisets are in Chapter 7.

An arc inscription need not be given explicitly. The default arc inscription is `empty`, the multiset of no tokens. When an input arc inscription is given as or defaults to `empty`, no tokens need to be in the input place in order for the transition to occur. When an output arc inscription is given as or defaults to `empty`, no tokens will be put into the output place if the transition does occur.

### Guards

A *guard* is a boolean expression associated with a transition. Guards are customarily written inside square brackets, to help distinguish them from other regions.

There is only one transition in FirstNet, and hence only one guard:



A transition's guard must evaluate to boolean **true** in order for the transition to occur. Details on guard syntax and usage appear in Chapter 7.

A guard need not be given explicitly. The default guard is boolean **true**. When a guard is given as or defaults to **true**, the guard never restricts the transition from occurring.

### CP Net Execution

We have now looked at the principal components of a CP net, and have seen how they relate to each other syntactically. But that is only part of the story. We wouldn't need all this machinery just to describe the structure of a system: a static modeling paradigm such as IDEF0 can do that much less effort.

A CP net is more than a description of system structure: CP nets are intended to be executed. Such execution can provide information about system dynamics that could never be derived by looking at a static model and considering its implications. There is just too much information involved in the functioning of a complex system for the unaided human mind to cope with.

This chapter has not addressed the question of how CP net components work together to define how a net will behave during execution. This question is answered in Chapter 7. But before we pro-

ceed to CP net dynamics, it will be useful to gain a little more familiarity with CP nets as static representations. The best way to do this is to build one.

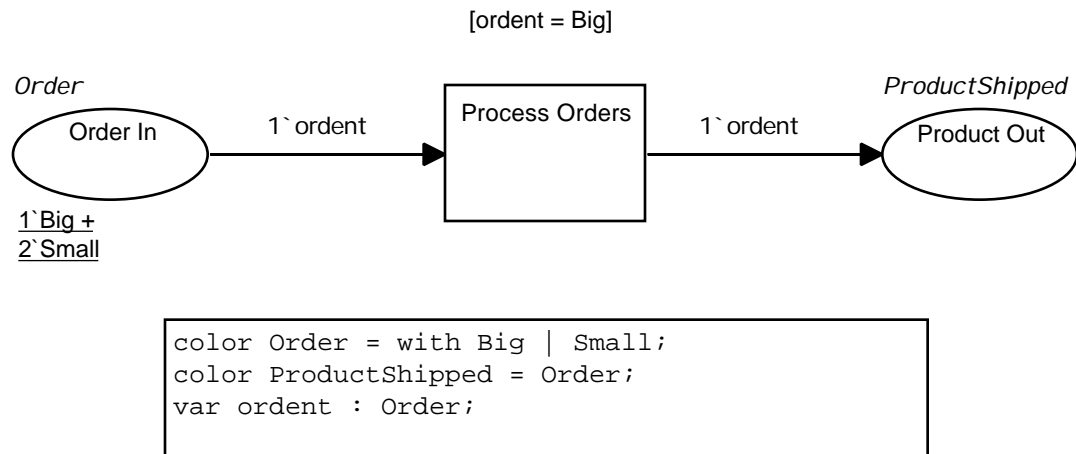




# Chapter 6

## Creating a Net With Design/CPN

This chapter shows you how to use the Design/CPN editor to create a small CP net. The net is FirstNet, the net that was used in Chapter 5 to illustrate the components of CP nets:



### Auxiliary Graphics and CPN Graphics

You could easily draw graphics that look exactly like the above net using the techniques covered in Chapter 4. But the result would not be a CP net: it would just be a collection of auxiliary graphical objects. A human who knows what is intended by the graphics can interpret them as a CP net, but a computer cannot, because the objects do not carry sufficient syntactic information.

In order for graphics to be meaningful to the computer as a CP net, the CPN meanings of the various objects must be explicitly indicated to the computer. There are two ways in which this can be done:

1. Draw auxiliary graphical objects first, and establish their CPN meanings afterwards. For example, one could draw an ellipse and a rectangle linked by a connector, then convert

the ellipse to a place, the rectangle to a transition, and the connector to an arc.

2. Construct the net out of objects that have appropriate CPN meanings from the start.

The problem with the first method is that converting large numbers of graphical objects to CPN objects would be very laborious and repetitive: the same stereotyped sequences of commands would have to be given over and over again. Therefore the second method is always used in practice.

The Design/CPN editor provides commands that let you draw a graphical object, make it a region if appropriate, specify its CPN meaning, and give it display attributes appropriate to that meaning, using only one command per object. These commands are available from the **CPN** menu. This chapter shows you how to use them.

A graphical object that is also a CP net component is called a *CPN graphical object*, which is usually shortened to *CPN object*. A node that is also a CP net component is called a *CPN Node*. A region that is also a CP net component is called a *CPN Region*. As these names suggest, the realms of auxiliary objects and of CPN objects are exactly parallel. The only difference between an auxiliary object and the corresponding CPN object is the presence or absence of a CPN meaning: graphically the objects are identical.

## Setting the Graphical Environment

A diagram contains many kinds of information, and all of them are represented in some way in the physical appearance of the diagram. To help make diagrams and net execution behavior visually self-explanatory, different types of information are customarily given different physical appearances, using a variety of graphical and textual conventions. These conventions are called *display attributes*. Their combined effect is to establish a particular *graphical environment*.

Display attributes exist at three levels: object attributes, diagram default attributes, and system default attributes.

### Object Attributes

Each graphical object has a set of display attributes that governs its individual appearance. These are the *object attributes*.

### Diagram Default Attributes

Each diagram has a set of display attributes that it applies by default to every newly created graphical object. These are the *diagram default attributes*. For brevity they are usually called the *diagram defaults*.

### System Default Attributes

Design/CPN keeps a master list of display attributes that it copies by default into every newly created diagram. These are the *system default attributes*. For brevity they are usually called the *system defaults*. (This “system” has nothing to do with the computer's operating system; it refers only to Design/CPN's internal record-keeping system.)

### Changing the Display Attributes

Display attributes can be changed whenever a net is not being executed. The change can be applied to a selected object or group of objects, to the diagram default attributes, to the system default attributes, or to any combination of these. Such changes are described in the next chapter.

It is also possible to replace all of a diagram's default attributes with the current system default attributes, or vice versa, as described later in this chapter.

## Establishing an Environment

In order for you to use this tutorial effectively, you must be working in the environment that the tutorial assumes. That environment is called the *tutorial environment*.

This is a matter of considerable importance. Design/CPN can provide very different environments for use in the very different purposes for which CP nets are created. Once you understand how to use Design/CPN, you will have no trouble establishing and working with any environment it can provide. But at this early stage, if you are not in the particular environment that this tutorial assumes, the results of carrying out the tutorial's instructions might bear little resemblance to the results the tutorial describes. This would be frustrating at best, if not totally confusing.

The tutorial environment is given by the diagram defaults of the diagram FirstNetDemo. To establish the tutorial environment, you must copy these defaults into Design/CPN's system defaults, as follows:

- Open the diagram FirstNetDemo in the TutorialDiagrams directory.

The diagram opens.

### Preserving the ML Configuration Options

When Design/CPN is installed, one of the steps is to record in the system defaults some information that Design/CPN needs in order to communicate with the ML process that was described at the beginning of this chapter. This information must be preserved when the tutorial environment is established. To preserve it:

- Choose **ML Configuration Options** from the **Set** menu.

A dialog appears. One of its options is **Load**.

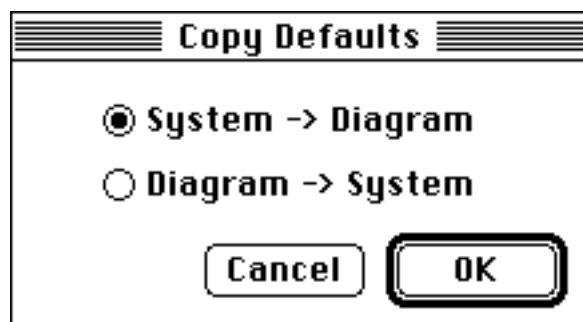
- Click **Load**.
- Click **OK**.

The necessary information about the ML process is copied from the system defaults to FirstNetDemo's diagram defaults. You can now make those diagram defaults the system defaults without erasing the information about the ML process.

### Copying the Diagram Defaults

- Choose **Copy Defaults** from the **Set** menu.

The **Copy Defaults** dialog appears:



- Click **Diagram -> System**.
- Click **OK**.

The dialog disappears. The system defaults are now the same as the FirstNetDemo diagram defaults. This insures that when you create new diagrams while working through this tutorial, they will have the correct defaults.

- Close the FirstNetDemo diagram.

In exercises later in this tutorial, you will be setting specific features of the environment to various values. An error might then leave you in an environment that does not match the tutorial's assumptions, so that its instructions do not have the effects described for them. If this occurs, and you cannot identify and correct the error, re-establish the tutorial environment, and start over from the beginning of the exercise where you encountered the difficulty.

## Creating the Net

The components of a CP net can be created in almost any order. The only requirements are:

1. A region's parent must be drawn before the region is drawn.
2. The nodes that an arc connects must be drawn before the arc is drawn.

Within these requirements, the choice of how to create a net is a matter of individual style. The order we will follow in this chapter has been chosen for simplicity rather than for efficiency. Later in this tutorial we will look at more efficient ways to create and modify nets.

- Start Design/CPN.
- Choose **N**ew from the **F**ile menu.

An new empty diagram appears. It has the diagram defaults that you just established as the system defaults by copying them from FirstNetDemo. The page displayed is named New#1.

## Creating the Transition

From the graphical standpoint, creating a transition is the same as creating a rectangle. The only difference is in the command you use to specify its creation.

- Choose **T**ransition from the **C**PN menu.

The editor is now in *transition creation mode*. The *transition tool* appears:



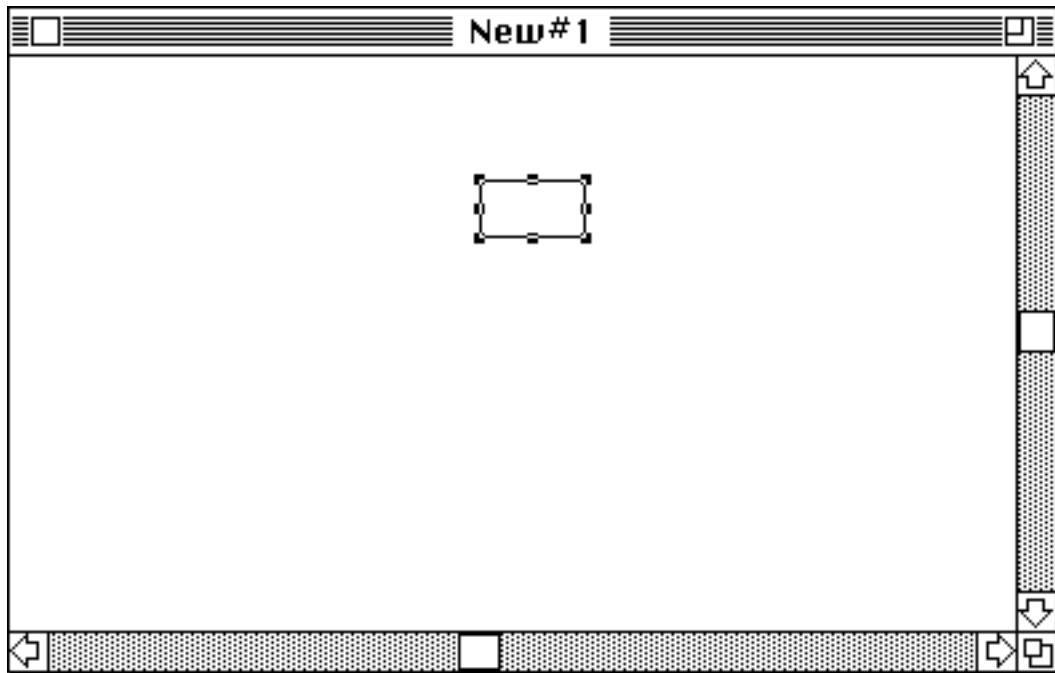
In transition creation mode, you can draw rectangles that are predefined as representing transitions.

- Draw a rectangle (transition) in the upper middle of the window.

A small editing box appears over the rectangle, just as with auxiliary rectangle creation. If you entered text, and did not specify a separate name region later, the text would by default become the name of the transition. If you did create a name region later, the text would thereafter have no functional significance. Such text is often used as a comments field. Don't type any text now.

- Leave transition creation mode via one of the techniques you used in Chapter 4 for exiting creation modes.

The window should now look something like this:



The rectangle you have just drawn looks like any other, and has all the usual graphical properties of a rectangle. But look at the status bar: it shows that you have drawn a transition, not just a rectangle. The editor has automatically performed the additional operations needed to designate the rectangle as a transition.

- Move and reshape the transition.

The transition acts exactly like an ordinary rectangle. Graphically it *is* a rectangle. Its CPN status as a transition has nothing to do with its graphical nature.

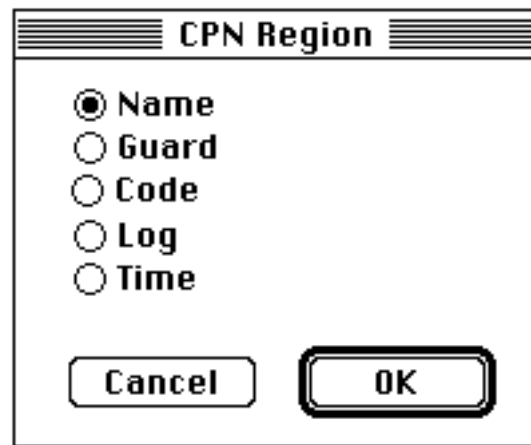
- Restore the transition to its original location and size.

### Naming the Transition

To name a transition, we attach a label region to it and type the name into the label. This is done with a single command. If there were more than one node on the page you would have to select which one you want to name, but there is only one, so it is selected already.

- Choose **CPN Region** from the **CPN** menu.

The **CPN Region** dialog for transitions appears:



The default is **Name**, which is what we want, so:

- Click **OK**.

The dialog disappears. The editor is now in *name region creation mode*. The mouse pointer becomes the *region tool*:



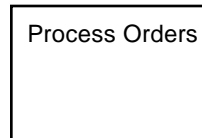
This tool is exactly like the label tool, except that the result of using it is a CPN region of whatever kind you have indicated, rather than a generic label.

- Move the tool to the inside top of the transition and click the mouse.
- Type "Process Orders".

- Leave name region creation mode.

The result looks like a label, and has all the graphical properties of a label, but the status bar shows that it is also a name region. The editor has automatically performed the additional operations needed to designate the label as a region that specifies a name.

- Reshape the transition and reposition the name region as needed so that the transition looks like this:



### Creating the Transition's Guard

Creating a guard is just like creating a name, except that you specify a guard region rather than a name region.

- Select the transition.
- Again choose **CPN Region** from the **CPN** menu.

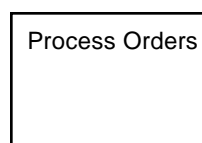
The **CPN Region** dialog for transitions reappears:

- Click **Guard**.
- Click **OK**.

The dialog disappears. The editor is now in *guard region creation mode*. The mouse pointer is again the region tool.

- Move the tool to the area above the transition and click the mouse.
- Type “[ordent = Big]”.
- Leave guard region creation mode.
- Reposition the guard region as needed so that the transition looks like this:

[ordent = Big]





You have now created a transition with a name and a guard.

### Creating the Input Place

Creating a place is graphically the same as creating an ellipse. The only difference is in the command you use:

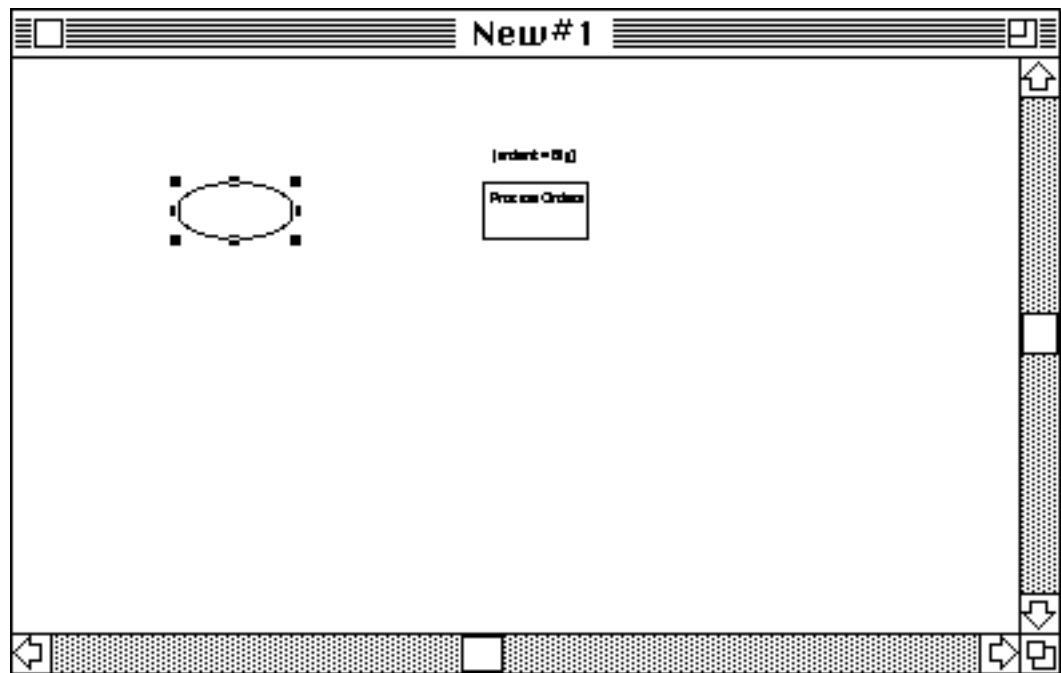
- Choose **Place** from the **CPN** menu.

The editor is now in *place creation mode*. The *place tool* appears.



- Draw an ellipse (place) halfway between the transition and the left side of the window. Don't enter any text into the place.

The window should now look something like this:



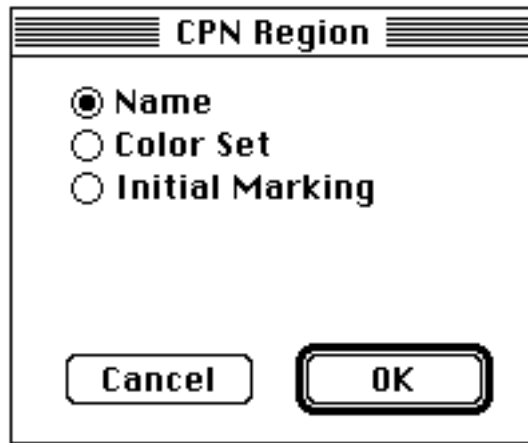
Look at the status bar: it shows that you have drawn a place, not just an ellipse.

### Naming the Place

Naming a place is essentially the same as naming a transition. Since you just created the place, it is already the current object.

- Choose **CPN Region** from the **CPN** menu.

The **CPN Region** dialog for places appears:



The default is **Name**, which is what we want, so:

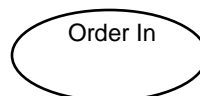
- Click **OK**.

The dialog disappears. The editor enters name region creation mode, and the mouse pointer becomes the region tool.

- Move the tool to the inside top of the place and click the mouse.
- Type "Order In".
- Leave name region creation mode.

The status bar shows that the region is a name region.

- Reshape the place and reposition the name region as needed so that the place looks like this:



### Specifying the Place's Colorset and Initial Marking

By now you probably see the general pattern for creating a CPN region:

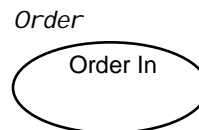
1. Select the object that is to have the region.
2. Choose **CPN Region** from the **CPN** menu.
3. Indicate the desired type of region.
4. Enter the text of the region.
5. Adjust appearance as needed.

This pattern works for all CPN regions that you create with the editor. Lets use it to establish a colorset and initial marking for the place.

- Select the place.
- Choose **CPN Region** from the **CPN** menu.
- Click **Color Set**.
- Click **OK**.

The editor enters colorset region creation mode; the mouse pointer becomes the region tool.

- Move the tool to the area above the ellipse and click the mouse.
- Type "Order".
- Leave colorset region creation mode.
- Position the colorset region so the place looks like this:

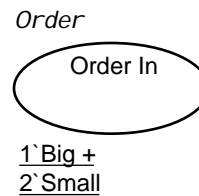


Now let's add the initial marking:

- Select the place.
- Choose **CPN Region** from the **CPN** menu.
- Click **Initial Marking**.
- Click **OK**.

The editor enters initial marking region creation mode; the mouse pointer becomes the region tool.

- Move the tool to the area below the ellipse and click the mouse.
- Type “1`Big + 2`Small”. Type a RETURN just before “2`Small”, to create a line break. Be careful to use backquote, not single quote.
- Leave initial marking region creation mode.
- Position the initial marking region so the place looks like this:



The reason for breaking the initial marking region onto two lines is to leave space for information that will be displayed during execution, as we will see in Chapter 8. Such linebreaks have only graphical significance: they count as whitespace, and do not affect meaning.

### Creating the Output Place

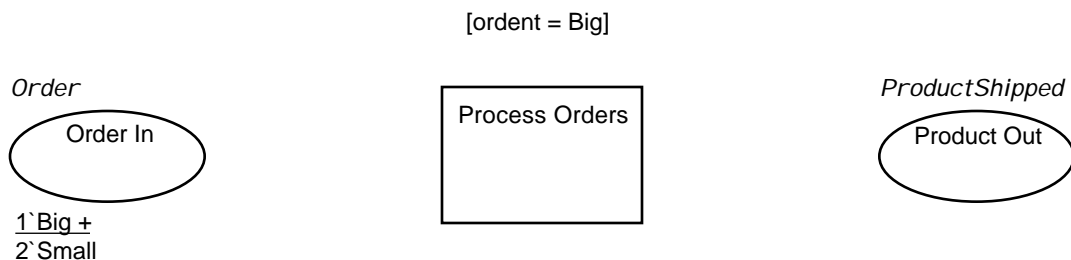
Creating the output place is exactly like creating the input place, except that there is no initial marking region.

- Create the output place, positioning it halfway between the transition and the right side of the window.

Don't worry if the output place isn't exactly the same size as the input place, or about any other minor variations in appearance that occur as you create the net. Techniques for polishing a net's appearance are covered in Chapter 11.

- Add the name and colorset regions to the output place.

When you are done, the growing net should look about as follows:



### Creating the Arcs and Arc Inscriptions

Creating an arc with an arc inscription is graphically the same as creating a connector with a text region.

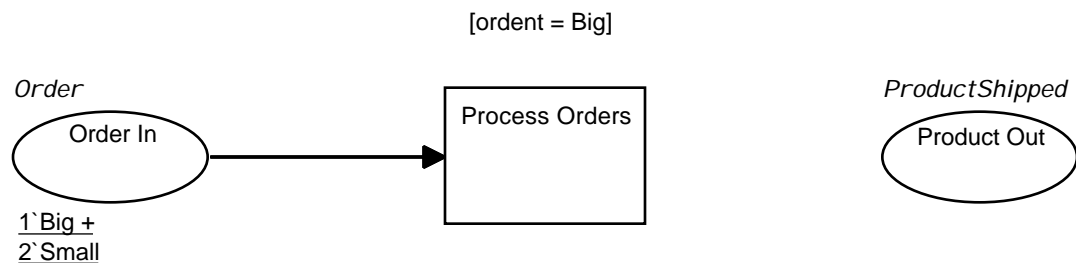
- Choose **Arc** from the **CPN** menu.

The editor is now in *arc creation mode*. The *arc creation tool* appears:



- Draw a connector (arc) from the place `Order In` to the transition.
- Leave arc creation mode.

The status bar shows that the connector is an arc. The net should look like this:

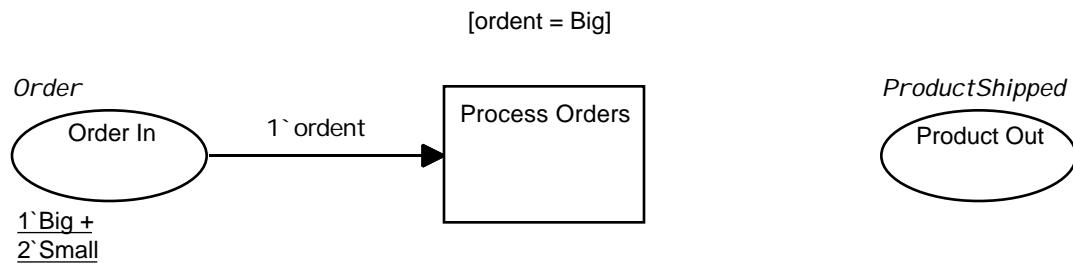


Now to add the inscription:

- Choose **CPN Region** from the **CPN** menu.

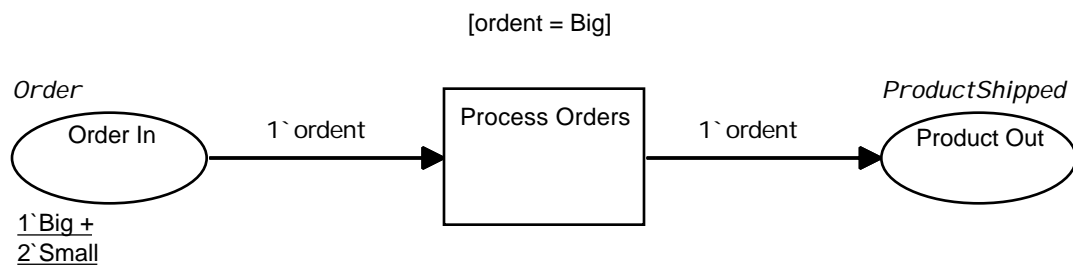
No **CPN Region** dialog appears. None is needed, because an arc can have only one type of region. The editor is now in *arc inscription creation mode*, and the mouse pointer is the label tool.

- Position the label tool above the input arc and click the mouse.
- Type “1`ordent”. (Backquote, not single quote.)
- Leave arc inscription creation mode.
- Reposition the inscription so the net looks like this:



- Use the same steps to create an arc from the transition to *Product Out*, with an inscription “1`ordent”.

The net should now look as follows:



You have now created two arcs with inscriptions. You have also finished creating the graphics of the net. All that remains is the creation of the global declaration node.

### Creating the Global Declaration Node

Graphically, creating a global declaration node is just a matter of drawing a rectangle and typing some text into it.

- Choose **Declaration Node** from the **CPN** menu.

The **Declaration Node** dialog appears:

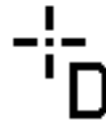


**Global** is the default, indicating a global declaration node.

- Click **OK**.

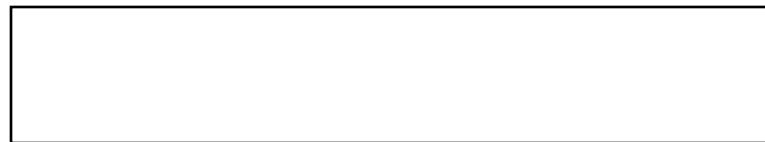
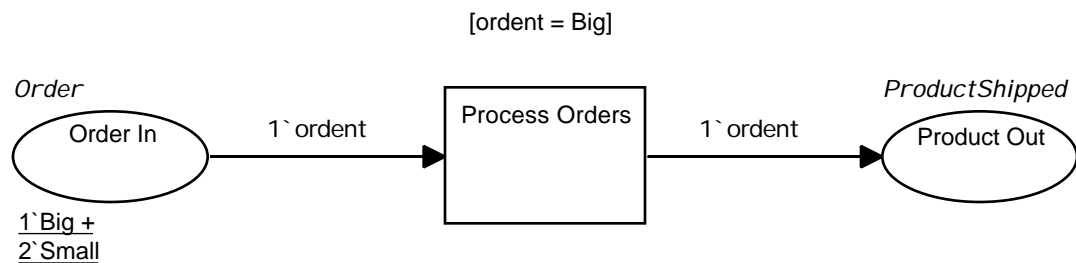
## Creating a Net

The editor is now in *global declaration node creation mode*, and the mouse pointer becomes the *declaration node tool*:



This tool draws a rectangle that is also a declaration node.

- Draw a rectangle. Position it so the net looks about as follows. (Recall that depressing SHIFT while creating a rectangle causes the rectangle to move with the mouse rather than reshaping.)



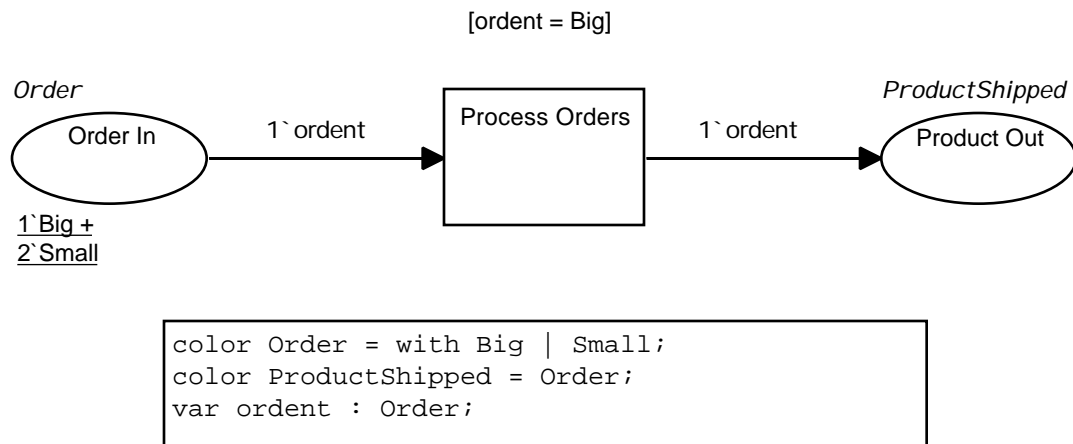
A small editing box appears over the rectangle. Text entered now or later into the declaration node's text field will be treated by Design/CPN as CPN ML code that specifies global declarations. You can stay in the creation mode and enter text immediately, but it is generally more convenient to enter it from text mode.

- Leave the creation mode.
- Enter text mode.
- Type three lines of text:

```
color Order = with Big | Small;  
color ProductShipped = Order;  
var ordent : Order;
```

- Leave text mode.

You have now created the global declaration node. The net should look like this:



That's it. The net is done.

## Saving the Diagram

The net you have just created will be the foundation of all the nets you build while going through this tutorial. You should save it now, so it will be available when the time comes to extend it.

- Choose **Save As** from the **File** menu.

The **Save As** dialog appears.

- Save the diagram in NewTTDiagrams (the directory you created in Chapter 2). Call the diagram NewFirstNet.

## More Efficient Editing Techniques

The editing techniques you used in this chapter are sufficient to create any CP net, but they would not be particularly convenient. For example, it is often useful to sketch out a net before filling in its details. You might then want to draw some or all of the graphics before creating any regions. Frequently a net has many similar or identical arc inscriptions; it would be useful to be able to create just one inscription and then copy it to every arc that needs it, perhaps modifying the various copies slightly after all have been created.

The Design/CPN editor provides both of these capabilities, and very many more. It can be used to create CP net structure with little if any wasted effort, and is compatible with essentially any editing style. Many of these techniques will be demonstrated later in this tutorial. They are all described in *The Design/CPN Reference Manual*.



# Chapter 7

## CP Net Dynamics

CP net components were covered in Chapter 5. This chapter shows how those components interrelate to specify the behavior of an executing CP net.

This chapter does not give complete details on any topic that it covers. Trying to cope with every detail of CP net dynamics from the start just obscures the essentials. It is more efficient to develop an overall understanding first, then systematically extend that understanding to provide a complete picture.

### Executing CP Nets

CP nets are actually programs, expressed in a hybrid language of graphics and text. CP net syntax is designed to insure that any legal CP net is completely and unambiguously defined.

CP nets do not execute themselves. Like any program, a CP net is executed. A CP net could be compiled into a stand-alone executable file, and executed directly by the computer. The problem with this method is that the file would have to contain a lot of code to manage the details of net execution, and to provide an interactive interface to it. This code would be the same in each file, which would be very wasteful. Therefore CP nets are executed by a run-time package called the *simulator*.

### The Design/CPN Simulator

Design/CPN contains a CP net simulator that is closely integrated with the editor, allowing a net to be iteratively constructed and executed with a minimum of overhead. The simulator both manages and displays CP net execution, and allows its course to be controlled in various ways to facilitate study and debugging.

The simulator is not just a passive intermediary between the computer and an executing net. It is an active agency that drives net execution forward by investigating the state of the net, determining how to change that state, and making the requisite changes.

When a CP net is given to the simulator for execution, the simulator first creates tokens as specified by any initial marking regions, and puts the tokens in the places. This establishes the initial state of the net. The simulator then executes the net by identifying transitions that can occur and effecting their occurrence.

### Understanding CP Net Execution

In order to build an understanding of CP net execution, and of what the simulator does to accomplish it, we must first look at two topics:

1. When can a transition occur?
2. What happens when a transition occurs?

Let's take each of these topics up in turn.

### When Can a Transition Occur?

A transition can occur whenever certain conditions are met. When those conditions are met, the transition is said to be *enabled*. The fact that a transition is enabled does not mean that it will actually occur: some other enabled transition might occur first, and change the state of the net so that the first transition is no longer enabled.

Three factors work together to determine whether a transition is enabled:

1. The multiset of tokens in each input place of the transition.
2. The input arc inscription on each input arc connected to the transition.
3. The transition's guard.

These three factors work together to determine whether a transition can occur. They work together so closely that none of them can be fully understood without understanding the other two. Therefore we must proceed iteratively in describing their contributions, until the nature of their shared activity becomes clear.

### Input Arc Inscriptions

An input arc inscription is an expression on an arc that connects a place to a transition. The inscription (possibly in conjunction with a guard) specifies a multiset, which may be empty. The default input arc inscription is `empty`.

## Guards

A *guard* is a boolean expression that is associated with a transition. This expression must evaluate to **true** in order for the transition to be enabled. The default guard is **true**.

## Criteria for Enablement

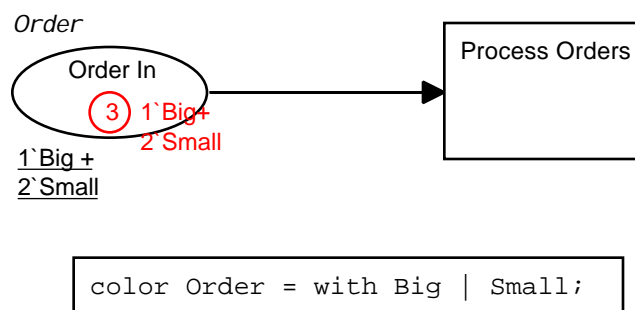
If each of a transition's input places contains the multiset specified by the place's input arc inscription (possibly in conjunction with the guard), and the guard evaluates to true, the transition is enabled, and can occur. Otherwise it is not enabled, and cannot occur. A transition's output places have no effect on its enablement.

A multiset whose existence allows a transition to be enabled is called an *enabling multiset*. When an enabling multiset exists, so that a transition is enabled, the transition is said to be *enabled with that multiset*.

Don't be surprised if you experience rather more heat than light at this point. The rest of this chapter should clarify matters considerably.

## Examples in This Chapter

Our standard example so far, FirstNet, has more in it than we need for examining transition enablement. Therefore we will start with an even smaller net:



Output places don't affect enablement, so there is no output place in this net.

Order In's marking is shown, since it is a factor in whether Process Orders is enabled. From this point on we will show place markings whenever they are relevant to some point being made. But keep in mind that markings are not part of the net, as places and transitions are, but rather indicators of its state.

`Process Orders` has no (explicit) guard. Such a guard defaults to **true**, and so can be ignored: its requirement is guaranteed to be satisfied irrespective of other factors.

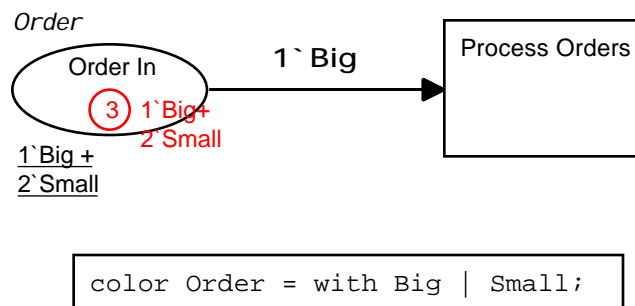
Enablement in this simple case is determined only by the tokens in the input place, and the input arc inscription. The above figure shows no inscription. Let's see how different inscriptions affect the enablement of `Process Orders`.

## Specifying Exact Token Values

Sometimes the exact token values that are to enable a transition are known in advance. In such a case, input arc inscriptions can specify the needed values literally, and no guard is needed.

### Specifying a Single Token

The simplest form of input arc inscription specifies a multiset of one token, and gives the tokens value as a constant:



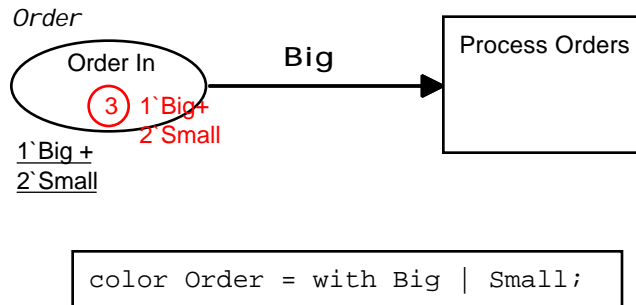
Here the input arc inscription specifies the multiset *1`Big*: that is, a multiset of one token whose value is *Big*. Does such a token exist in the input place *Order In*? Yes. Therefore `Process Orders` is enabled (can occur). The fact that there are also two *Small* tokens in *Order In* makes no difference one way or the other.

### The Simulator's Algorithm

When the simulator examines the above net to see whether `Process Orders` is enabled, it scans the tokens in *Order In* and tests each one to see whether its value is *Big*. If such a token is found (as it will be in this case), the simulator ends its search at that point, and puts `Process Orders` on a list called the *enabled list*. The purpose of this list will be described later.

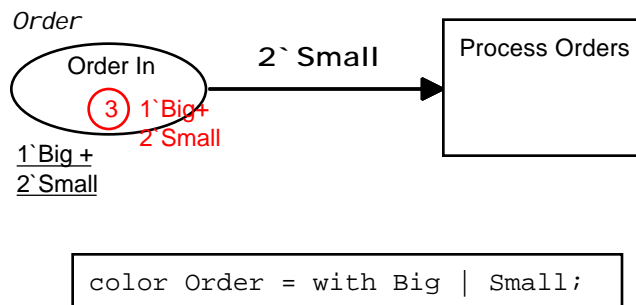
## Omitting a Count of One

As a convenience, the count in an arc inscription that specifies just one token can be omitted. Thus the following net is exactly equivalent to the one above:



## Specifying More Than One Token Instance

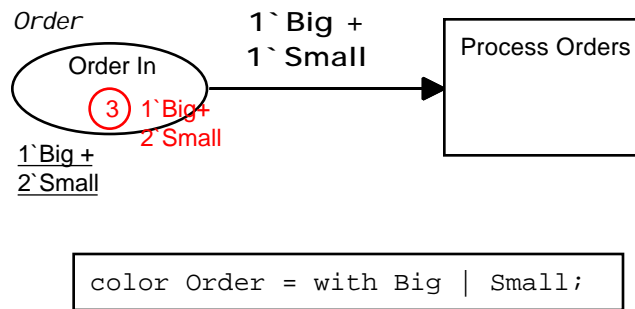
An input arc inscription can specify more than one tokens of a given value:



Here the input arc inscription requires two tokens of value `Small`. (Of course, this is not a reasonable requirement for an actual order processing system, but our purpose now is just to study CP net machinery.) The tokens exist, so the transition is enabled. The simulator will end its search as soon as it encounters the second value `Small` token, and put `Process Orders` on the enabled list.

## Specifying More Than One Token Value

An arc inscription can specify tokens of more than one value. For example:



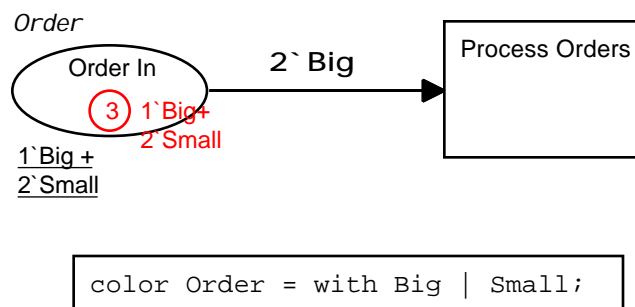
Process Orders is again enabled.

### The General Rule

In general, `Process Orders` will be enabled whenever the multiset specified by the input arc inscription is a subset of the multiset in `Order In`. Thus any of the following arc inscriptions (some of which we have seen already) will enable `Process Orders`:

1`Big + 2`Small  
1`Big + 1`Small  
1`Big  
2`Small  
1`Small  
empty

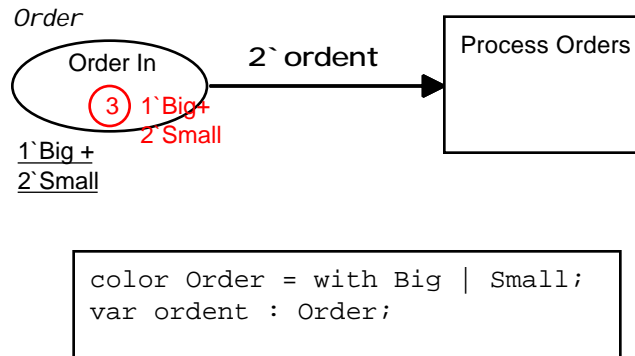
However:



There is only one `Big` token in `Order In`, but the arc inscription requires two, so `Process Orders` is not enabled. There is an infinite number of possible non-enabling arc inscriptions. All share the same property: they define a multiset that is not a subset of the multiset in `Order In`.

## Specifying Variable Token Values

Sometimes it does not matter exactly what values enabling tokens have: all that matters is that they are present in appropriate numbers. To allow for such a case, input arc inscriptions can use variables rather than constants:



Here `ordent` (which stands for “order entered”) is a CPN variable that can take on any value from the colorset `Order`.

## Binding an Arc Inscription Variable

Initially `ordent` has no particular value: it is said to be *unbound*. Since `ordent` is unbound, the arc inscription that uses it does not evaluate to a multiset: trying to evaluate it would just produce an error.

The simulator's task at this point is to determine whether `Process Orders` is enabled. To do that it needs to evaluate the arc inscription `2`ordent`, and to perform the evaluation it must have a value for `ordent`. Lacking any other source for a value, it chooses one from among the possible legal values and makes it the value of `ordent`. When this has occurred, `ordent` is said to be *bound* to that value.

Suppose the simulator binds `ordent` to `Big`. The inscription then evaluates to `2`Big`. But there is only one `Big` token in `Order In`, so `Process Orders` is not enabled when `ordent` is bound to `Big`. The simulator does not give up, however: it next tries binding `ordent` to `Small`. The arc inscription then evaluates to `2`Small`. There are two `Small` tokens in `Order In`, so `Process Orders` is enabled when `ordent` is bound to `Small`.

A binding that causes a transition to be enabled is called an *enabling binding*, and the transition is said to be *enabled with that binding*. In this case, there is one enabling binding, `ordent = Small`. The simulator will put `Process Orders` on the enabled list along with a record of the enabling binding.

### Constraining Token Values

We have now looked at a way to specify the values of enabling tokens exactly (with constants), and a way to leave their values unspecified (with variables). Obviously these methods would not be enough in a realistic model. In order to model real systems, we need a way to require token values to meet any criterion we can define. Such a requirement is called a *constraint*.

In order to constrain token values, CP nets use arc inscriptions in conjunction with guards. The method is to bind CPN variables in arc inscriptions, and perform boolean tests on those values in a guard.

### Guard Syntax

A guard is a boolean expression that operates on token values (or parts of composite values). Guards have the syntax typical of boolean expressions with infix notation. The comparison operators used are:

=	equal
>=	greater than or equal
>	greater than
<=	less than or equal
<	less than
<>	not equal

These may be linked together with (in order of precedence):

<b>not</b>	boolean NOT
<b>andalso</b>	boolean AND
<b>orelse</b>	boolean OR

Guards are customarily written enclosed in brackets, to help distinguish them visually from other net components.

### Use of Parentheses

When **not**, **andalso**, or **orelse** are used, parentheses are required around the comparisons that they link. Thus:

```
[A < B andalso C > D]
```

is illegal. Instead use:

```
[(A < B) andalso (C > D)]
```

Parentheses may also be used as needed for clarity or to override the precedence order.



## Shortcut for andalso

A comma may be used in place of **andalso**. Thus:

```
[(A < B) andalso (C > D)]
```

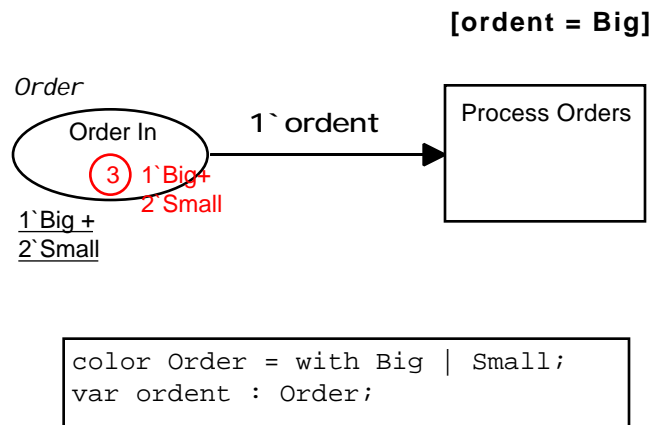
and:

```
[(A < B), (C > D)]
```

mean the same thing.

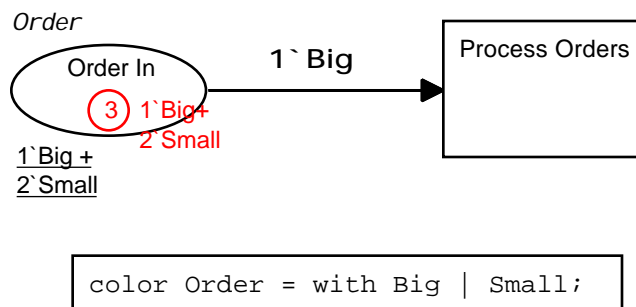
## Constraining a Single Token

The simplest constraint restricts the value of a single token:



In order for `Process Orders` to be enabled, there must be a binding for `ordent` such that `1`ordent` evaluates to a multiset that exists in `Order In`, and `[ordent = Big]` is true.

Obviously this is not much of a constraint: the transition is enabled when `ordent = Big`. We might as well have stuck with our very first example:

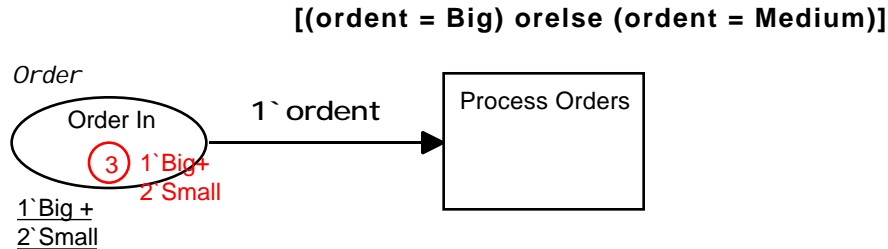


The equivalence between these two nets is a simple example of a very general truth: the realms of arc inscriptions and guards are not

disjoint. A given constraint can often be implemented in several ways, each of which uses the capabilities of arc inscriptions and guards in a different manner to produce the same effect.

### More Complex Constraints

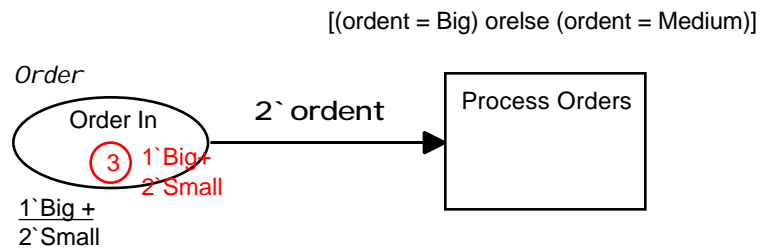
Let's look at a slightly more interesting example:



```
color Order = with Big | Medium | Small;  
var ordent : Order;
```

There are now three types of Order: Big, Medium, and Small; and hence three possible bindings for ordent. Binding ordent to Small doesn't work, because then the guard is not true. Binding it to Medium doesn't work, because there are no Medium tokens in Order In. Binding it to Big works. There is a Big token in Order In, which satisfies the arc inscription, and the binding causes the guard to evaluate to true. Such a binding is sometimes said to *satisfy* the guard.

On the other hand:



```
color Order = with Big | Medium | Small;  
var ordent : Order;
```

Here the transition is not enabled. If ordent = Big or ordent = Medium, there are not enough tokens to satisfy the requirements of the arc inscription, but if ordent = Small, the guard is false.

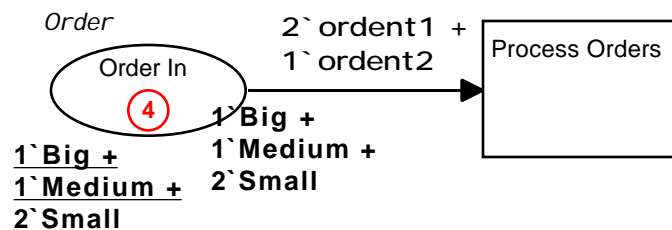
## Constraining More Than One Token

The techniques we have looked at are completely general: you can do anything with them that makes syntactic sense. For example:

```

[ ((ordent1 = Big) or else (ordent1 = Medium))
  andalso (ordent1 <> ordent2) ]

```



```

color Order = with Big | Medium | Small;
var ordent1, ordent2 : Order;

```

In this case there are two CPN variables to bind. The simulator will try various bindings for each of them, looking for a binding that satisfies the arc inscription and the guard. A successful binding will bind `ordent1` to `Big` or `Medium`, and `ordent2` to either of the values that `ordent1` is not bound to; and will cause the arc inscription to evaluate to a multiset such that `Order In` contains two of whatever `ordent1` is bound to, and one of whatever `ordent2` is bound to.

This transition is not enabled. Examine it carefully until you are sure you understand why.

## What Happens When a Transition Occurs

You now have a picture of what is necessary in order for a transition to be enabled: that is, in order for it to be able to occur. Now let's look at what happens if it actually does.

The “if” is significant. Not every enabled transition actually occurs: some other enabled transition might occur first, and change the net so that the first transition is no longer enabled. The nature of such a change will soon become obvious.

It is often convenient to imagine a transition as an active principle that changes the state of a net when it occurs, but that is not actually the case. The simulator performs all actions that constitute transition occurrence: a transition is not really an independent agency, but a representation of an activity that can change the state of a system.

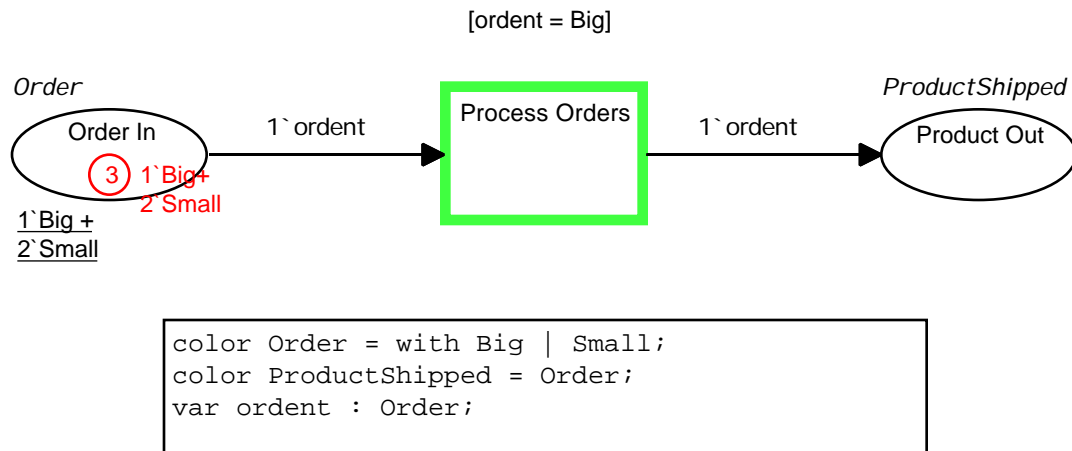
When the simulator effects the occurrence of a transition, it is said to *fire* the transition. For brevity, transitions themselves are often said to fire, but it is important to remember that the simulator actually does everything: the transition just defines what is to be done.

When the simulator fires a transition, it does the following:

1. Rebind any CPN variables as indicated by the enabling binding.
2. Evaluate each input arc inscription.
3. Remove the resulting multiset from the input place.
4. Evaluate each output arc inscription.
5. Put the resulting multiset into the output place.

### A Simple Example

To illustrate net execution we need an output place. Let's start off with our old friend, FirstNet, and see how it executes.



Note the highlighting around `Process Orders`. This is a convention that indicates that the transition is enabled.

#### Rebind any CPN Variables Per the Enabling Binding

A transition doesn't necessarily fire as soon as it is found to be enabled. The enabling binding has to be restored before firing can proceed.

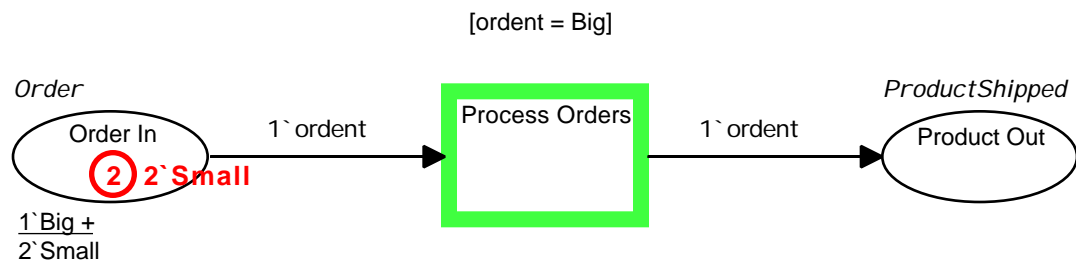
The enabling binding was `ordent = Big`, so `ordent` becomes bound to that value.

### Evaluate Each Input Arc Inscription

There is only one input place, so there is only one input arc inscription to evaluate. When `1`ordent` is evaluated with the binding `ordent = Big`, the result is the multiset `1`Big`. This is the enabling multiset: its existence is what caused `Process Orders` to be listed as enabled.

### Remove the Enabling Multiset from Each Input Place

This is just a matter of multiset subtraction. There is a multiset in the input place, and there is an enabling multiset. Subtracting the latter from the former removes the enabling multiset from the input place. When the subtraction has been accomplished, FirstNet looks like this:



```
color Order = with Big | Small;
color ProductShipped = Order;
var ordent : Order;
```

`Process Orders` is still highlighted, but the highlighting is now twice as thick. Double-thickness highlighting indicates that a transition is in the process of firing. The marking of `Order In` is now `2`Small`, reflecting the subtraction of the enabling multiset `1`Big` from the place's previous marking, `1`Big + 2`Small`.

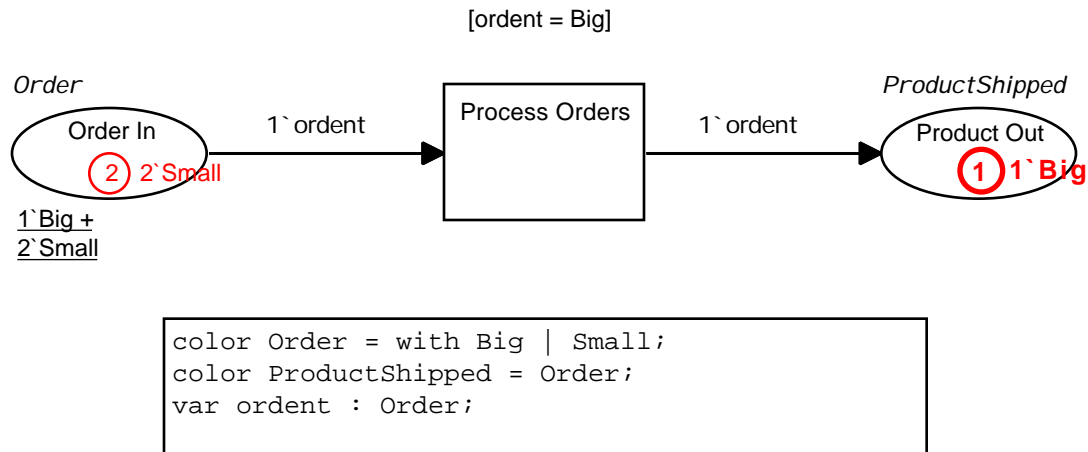
You may wonder where subtracted tokens go. They don't go anywhere: they just disappear. This is the principal difference between a token and an object in an object-oriented programming system: tokens are really indicators of the state of a net, not objects that are conserved. This has many advantages, as we shall see.

### Evaluate Each Output Arc Inscription

There is only one output place, so there is only one output arc inscription to evaluate. The inscription uses the CPN variable `ordent`, which is currently bound to `Big`. When `1`ordent` is evaluated with the binding `ordent = Big`, the result is the multiset `1`Big`. This is the *output multiset*.

### Put the Output Multiset into the Output Place

This is just a matter of multiset addition. There is a multiset in the output place (in the current case it is the multiset `empty`), and there is a newly created multiset. Adding the latter to the former puts the output multiset into the output place, where it joins any tokens that were already there. When the addition has been accomplished, FirstNet looks like this:



There is no highlighting now, because firing is complete and the transition is no longer enabled. The marking of `Product Out` is now `1`Big`, reflecting the addition of the output multiset `1`Big` to the output place's previous marking, `empty`.

The `1`Big` that has been added to `Product Out` has no connection whatever with `1`Big` that was subtracted from `Order In`. It is a new token: the fact that its value is the same as that of the subtracted token results from the fact that both got their values from the binding of `ordent`. If the output arc inscription had been `10`ordent`, there would now be ten `Big` tokens in `Product Out`.

## Executing a Net in the Simulator

We have now covered the essentials of CP net execution. We could go on to trace many other execution possibilities for FirstNet, paralleling the variations we looked at when looking at enablement, but doing so would not be efficient. FirstNet is appropriate for illustrating the rules that govern net execution, but it is too simple to do anything very interesting. We would just see the same things over and over again.

The theory of net execution is not the practice of it. Let's take FirstNet into the simulator and actually execute it. Once you can use both the editor and the simulator, we will begin to extend FirstNet

so that it becomes a realistic model, rather than just an exercise. Ultimately it will evolve into a very useful and informative model, called the Sales Order Model, that demonstrates the skills you need to make any CPN model.





# Chapter 8

## Executing a Net With Design/CPN

This chapter shows you how to use the Design/CPN simulator to execute a CP net. The net you will execute is FirstNet, the same net that was used in Chapter 5 to illustrate the components of CP nets, and that you created a version of in Chapter 6.

You could take NewFirstNet, the diagram you created in Chapter 6, into the simulator and execute it, but the results might not be ideal. As with any computer programming task, a variety of errors can occur during the creation of a net. If you attempted now to execute NewFirstNet, you might encounter errors that this tutorial does not anticipate, and has not yet equipped you to deal with.

There is also the problem of being able to observe what happens as a net executes. Effective observation sometimes requires rearranging a net somewhat, so that graphical information about the course of execution is not obscured by components of the net. We have not yet studied the techniques necessary for doing such rearranging.

Therefore we will begin working with the simulator by executing one of the diagrams supplied with this tutorial: FirstNetDemo. This diagram contains a version of FirstNet that is known to be error free, and has been arranged for optimum observability. Following this course will allow you to learn the techniques used to execute a net, and to study the standard information Design/CPN displays to describe the phases of net execution, without interference from unanticipated factors.

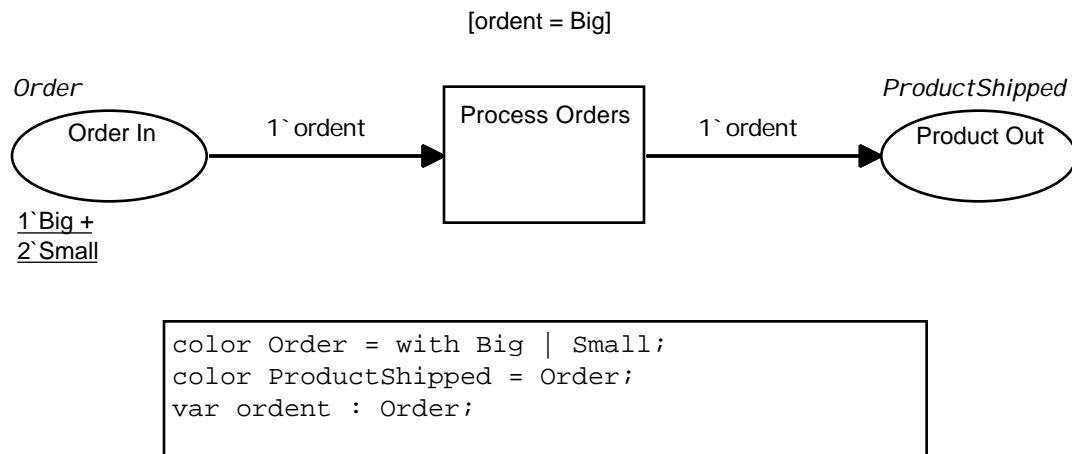
### Opening the Net

- Choose **Open** from the **File** menu.
- Navigate to the directory TutorialDiagrams.
- Open the diagram FirstNetDemo.

## Design/CPN Tutorial

---

The diagram opens, and a page named Fnet#1 appears. The page contains a standardized version of FirstNet:



## Loading ML Configuration Information

The diagram you have just opened was not created on your system, so it does not contain the information necessary to allow Design/CPN to communicate with the ML process. Therefore you must load that information into the diagram.

- Choose **ML Configuration Options** from the **Set** menu.

A dialog appears.

- Click **Load**.
- Click **OK**.

The necessary information about the ML process is copied from the system defaults to FirstNetDemo's diagram defaults. Design/CPN can now access the ML process, permitting it to syntax check and then execute the net.

## Performing a Syntax Check

There would be no use attempting to execute a net that had syntax errors. Therefore Design/CPN requires that a syntax check be performed before you take a net into the simulator.

- Choose **Syntax Check** from the **CPN** menu.

Design/CPN now starts a separate application called the ML Interpreter, which it uses to compile and execute computer code written in a language called CPN ML. If you see a dialog that mentions a problem of any kind, see Appendix C before you proceed.

The diagram on which you are performing the check had an ML file when it was under development at Meta Software, but does not have one now. Therefore, a dialog appears that states:

**Cannot find ML file.**

and offers you two options:

**Build from scratch**

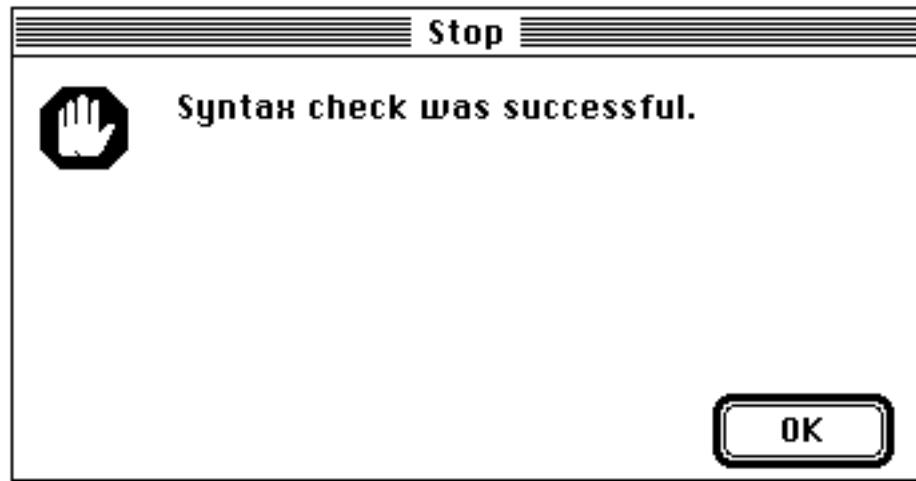
**Modify ML file name**

The first choice, **Build from scratch**, is the default: choosing it will cause the simulator to build a new ML file based on information in the diagram and DB files.

- Click **OK**.

Several things happen in sequence:

1. The status bar displays: Checking Syntax.
2. The ML process becomes active.
3. The status bar displays: Checking Global Declaration Node.
4. The status bar displays: 0 Pages, 0 Places, 0 Transitions Checked.
5. As the syntax check proceeds, the count of pages, places, and transitions changes to indicate the progress of the check.
6. The status bar displays: Finished Syntax Check.
7. The **Syntax Check Successful** dialog appears:



- Click **OK**.

Design/CPN has found no syntax errors. If there were errors, the course of events would have been different, as described in Chapter 9.

You may have noticed some other status bar messages that flashed by very rapidly while the syntax check was underway. These messages describe operations that are of interest only to those concerned with Design/CPN internals, so you can ignore them.

If you direct Design/CPN to enter the simulator, and have not first done a syntax check, Design/CPN will do one automatically. If the check is unsuccessful, Design/CPN will remain in the editor and display information about the error(s), as described in Chapter 9.

## Designating a Prime Page

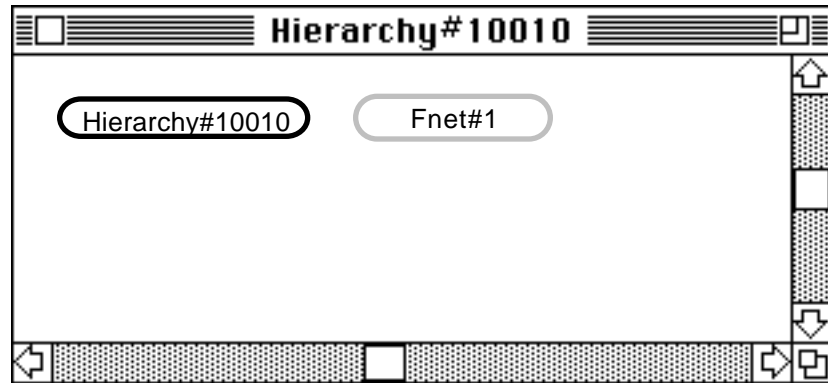
The process of creating and debugging a large net is a substantial undertaking. It would not be feasible to require that a net be completely executable before any of it could be executed. The same consideration arises in ordinary programming: a program must be usable in pieces, with the pieces in various stages of completion, in order for efficient development work to be done.

To facilitate incremental net development, Design/CPN does not automatically execute an entire diagram if it executes any of it. It executes only those parts of the diagram that either appear on a specially designated type of page, called a *prime page*, or are in some way referenced (directly or indirectly) by a prime page. A diagram may have any number of prime pages. In order to execute, a diagram must have at least one prime page, or the simulator will not find anything to execute.

The diagram has only one executable page, but that page must still be designated as a prime page. To so designate it:

- Choose **Open Page** from the **Page** menu.

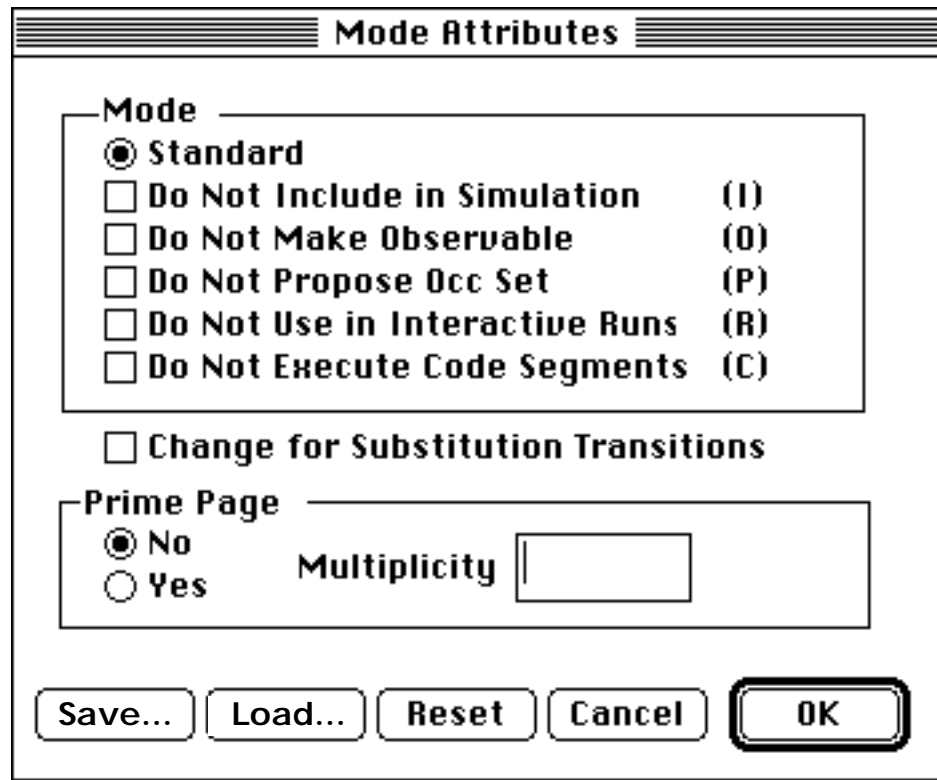
The hierarchy page appears:



(The window will be larger on your screen, but will contain the information shown here.)

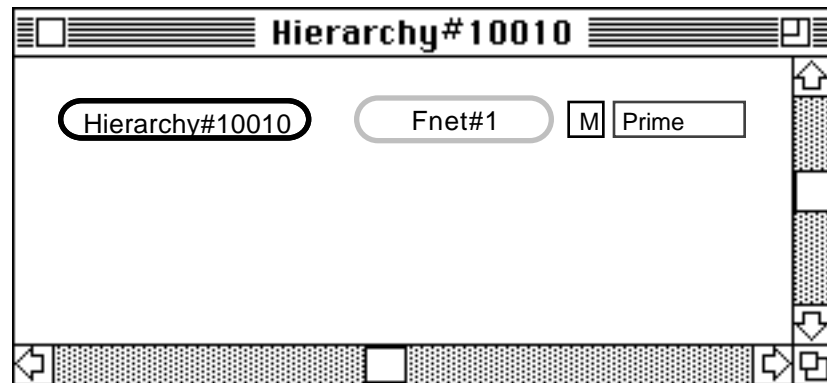
- Select the page node for Fnet#1 (it may already be selected).
- Choose **Mode Attributes** from the **Set** menu.

The **Mode Attributes** dialog appears:



- Under **Prime Page**, click **Yes**.
- Click **OK**.

The page Fnet#1 is now a prime page. Some new information appears on the hierarchy page to indicate this.



The new information consists of two regions. The **M** is called a *page mode key region*; the **Prime** is called a *page mode region*. These regions together indicate that Fnet#1 is a prime page.

Now let's go back to Fnet#1:

- Double-click on the page node for Fnet#1.

Fnet#1 becomes the current page.

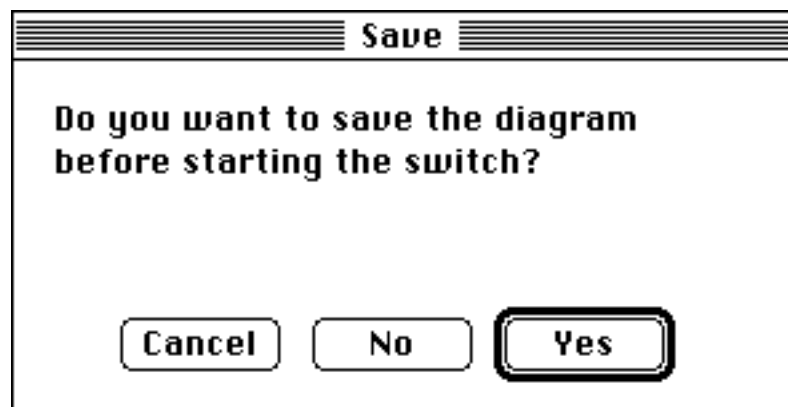
## Entering the Simulator

You have insured that the net has no syntax errors, and have designated a prime page. You can now enter the simulator and execute the net.

- Choose **Enter Simulator** from the **File** menu.

If you see a dialog that mentions a problem of any kind, see Appendix C before you proceed.

Design/CPN displays the following dialog:



This is a good time to save a copy of the diagram. If you must interrupt your work with the tutorial while you go through the rest of this chapter, resume by opening your copy, not the original. You will also need this copy in later chapters. Meta

- Click **Yes**.

The original diagram in TutorialDiagrams is locked, so the **Save As** dialog appears.

- Navigate to the NewTTDiagrams directory.
- Save a copy of FirstNetDemo under the name FirstNetCopy.

When you save the file, a dialog will appear offering you an opportunity to specify the name of the ML file. This dialog appears only when a new ML file has been created “from scratch.” The default name shown in the dialog is correct, so:

- Click **OK**.

Design/CPN now generates the code needed to execute the diagram. As before, status bar messages that appear only momentarily and are not of general interest are omitted.

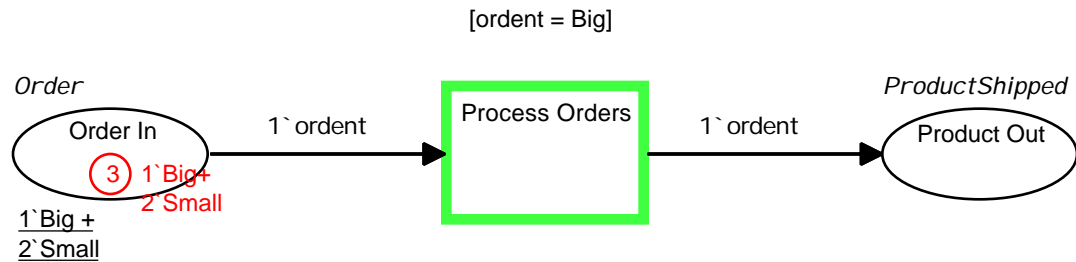
1. The status bar displays: 0 Pages and 0 Transitions Switched. To “switch” a page or transition is to generate the code needed to execute it.
2. As Design/CPN switches pages and transitions, it updates the page and transition counts to indicate its progress.
3. The status line displays: Creating Instances. Design/CPN is generating tokens as specified by the initial markings in the diagram.
4. The status line displays: Generating Automatic Code. Design/CPN is generating code to be used during automatic execution.
5. The status line displays: Updating Open Pages. Design/CPN is making the tokens and regions it has generated visible on any open pages, so that the appearance of the pages correctly depicts the current diagram state.
6. The status line displays: Ready to Simulate.
7. When the switch is finished, Design/CPN beeps.

You have successfully entered the simulator, and are ready to execute the net.

## Simulation Regions

Now that you are in the simulator, the net has a different appearance than it did in the editor. A multiset representation has appeared inside `Order In`, indicating its current marking, and `Process Orders` has been highlighted, indicating that it is enabled. All of these indications are provided automatically by the simulator to make the state of the net more obvious.





```
color Order = with Big | Small;
color ProductShipped = Order;
var ordent : Order;
```

The simulator changes a net's appearance to indicate its state by adding various regions to the net. Such regions are called *simulation regions*. Simulation regions are similar to the CPN regions you created in Chapter 4, in that they are subsidiary graphical objects. The difference is that they are created automatically by the simulator to indicate net state, rather than by a human to indicate net structure.

## Simulation Regions Indicating Place Markings

The multiset representation in *Order In* consists of two regions: the number in a circle is called the *current marking key region*, and the count-value pairs next to it constitute the *current marking region*. Together these regions indicate that there are currently three tokens in the place, one of value *Big* and two of value *Small*, as specified by the initial marking region 1`Big + 2`Small.

As Chapter 5 pointed out, the existence of both the initial marking region and the current marking regions is not a redundancy. The initial marking region is just a label region with some text in it that tells the simulator what tokens to create as the place's initial marking. The current marking regions represent actual tokens, created on entry to the simulator.

As the net executes, the current marking regions will change to indicate the changing marking of *Order In*, but the initial marking region will remain the same. Current marking regions are used during simulation to indicate all non-empty place markings.

## Simulation Region Indicating Enablement and Firing

The highlighting around *Order In* indicates that it is enabled, as you can verify by looking at its marking and the arc inscription. This highlighting is actually a region called a *transition feedback region*, or for brevity a *feedback region*.

Whenever a transition is enabled or is in the process of firing, it is highlighted with a feedback region. When the transition is enabled, this region looks as it does in the case of `Order In`. When a transition is firing, the thickness of the region doubles.

## The Sim Menu

Net appearance is not the only thing that changes when you enter the simulator. Look at the menu bar: the **CPN** menu item has been replaced by **Sim**. In the simulator, you cannot create new net structure (though you can modify existing structure to some extent) so there is no need for the **CPN** menu commands. In their place Design/CPN provides the **Sim** menu. The commands in the **Sim** menu start, stop, and control net execution, as described in the rest of this chapter.

## Executing the Net

We will execute this net several times, and look at every detail of what happens. First let's watch the execution sequence as a whole:

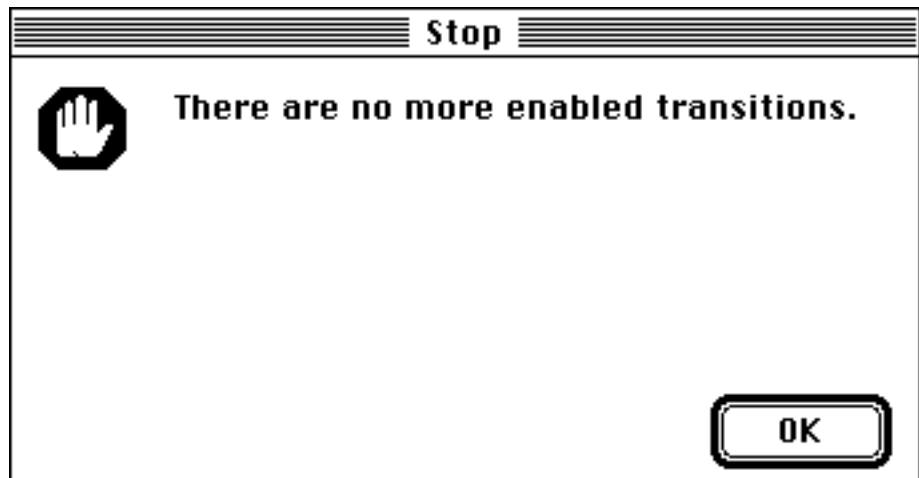
- Choose **Interactive Run** from the **Sim** menu.

A variety of actions follow. These constitute one step of the simulator's execution algorithm, which was described in detail in Chapter 7. When the step is finished, the **Step Finished** dialog appears:



- Click **Cont** (for Continue).

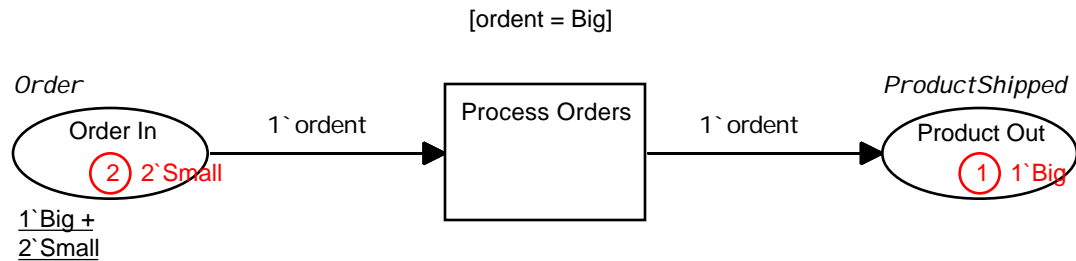
The simulator displays this dialog:



When the simulator attempted to continue execution, it found that there was nothing more for it to do, because there are no more enabled transitions. Execution of the net is complete.

- Click **OK**.

The net now has this appearance:



```

color Order = with Big | Small;
color ProductShipped = Order;
var ordent : Order;
    
```

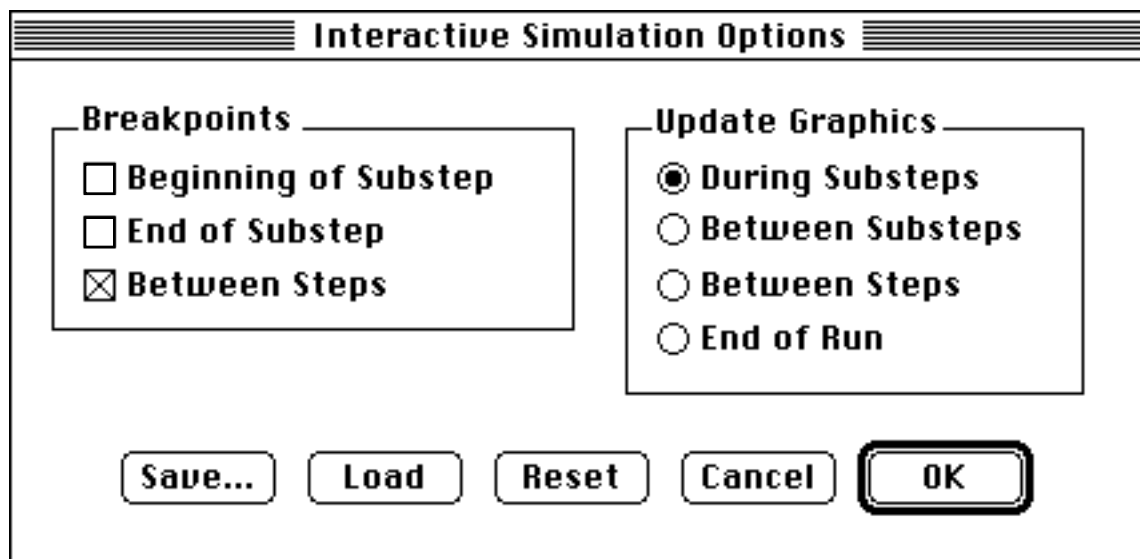
As specified by the arc inscriptions and guard, one token of value *Big* has been subtracted from *Order In*, and one token of value *Big* has been added to *Product Out*. Since *Product Out* is no longer empty, current marking regions have appeared in it to indicate its marking, 1`*Big*.

### Observing Net Execution

A lot of action occurred during execution of the net, and most of it went by too fast to be observable. In order to study what happens during execution, we need to slow things down. This is accomplished by setting breakpoints that cause the simulator to suspend execution at various points during its execution cycle.

- Choose **Interactive Simulation Options** from the **Set** menu.

The **Interactive Simulation Options** dialog appears:



Notice which choices are selected. You have already seen the effects of these choices:

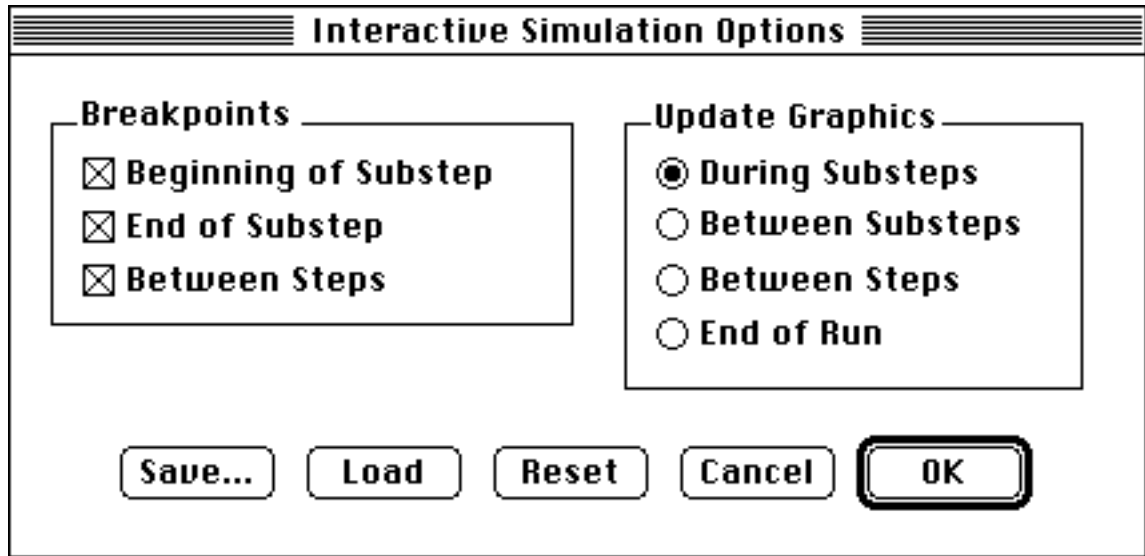
1. The **Step Finished** dialog appeared because **Between Steps** is selected under **Breakpoints**. Selecting this breakpoint causes the simulator to pause after each execution cycle (step) and display the **Step Finished** dialog, as shown above.
2. Various regions appeared briefly as the net executed, because **During Substeps** is selected under **Update Graphics**. Selecting this option causes the simulator to display regions that depict the details of transition firing, as described below.

To better display the details of net execution:

- Click on **Beginning of Substep**. Selecting this breakpoint causes the simulator to pause when it has determined what input tokens to subtract, and what output tokens to add.

- Click on **End of Substep**. Selecting this breakpoint causes the simulator to pause immediately after the firing transition has subtracted the input tokens and added the output tokens.

The dialog should now look like this:



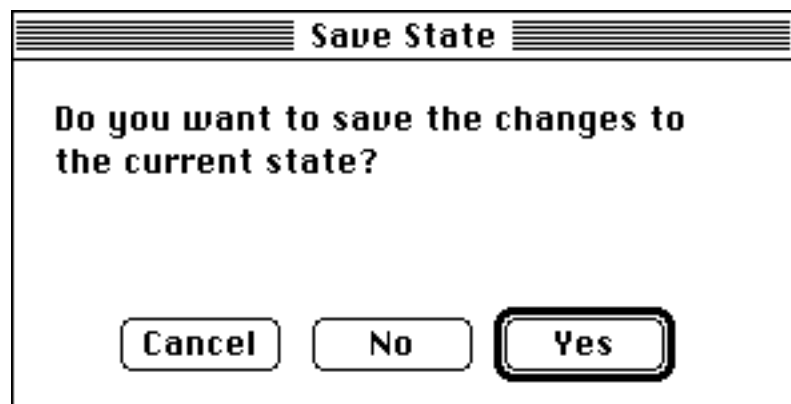
- Click **OK**.

### Re-Executing the Net

The net cannot be re-executed in its current state, for the same reason that execution could not continue: there are no enabled transitions. To execute the net again, we must first restore its original state.

- Choose **Initial State** from the **Sim** menu.

A dialog appears:



When a complex net has been executing for a long time, its current state represents a considerable investment in time, since it could be recreated only by redoing the entire execution process. This dialog helps to protect against accidental erasure of such a state. In this case:

- Click **No**.

The status bar describes the stages of initialization. When initialization is complete, the status bar displays Finished Initializing State. The net is now in exactly the same state that it was in before it executed.

### Starting Execution

It isn't possible to explain everything you will see the simulator do as you re-execute the net. Some of its actions relate to things we have not as yet covered. For now, just ignore anything you don't understand: all will be made clear in due time.

- Choose **Interactive Run** from the **Sim** menu.

Several things happen in sequence:

1. The highlighting around **Process Orders** disappears.
2. The status bar displays three messages

Calculating Occurrence Set

Updating Graphics

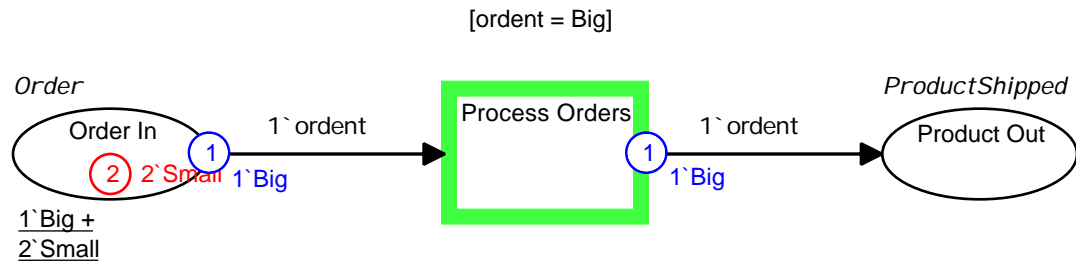
Breakpoint 1 Step: 1

3. Highlighting reappears around **Process Orders**.

The highlighting is twice as thick as before. The highlighting indicates that the transition is now in the process of firing.

### Breakpoint 1

Execution has stopped. You are at the breakpoint **Beginning of Substep**. (For brevity, the status bar describes this breakpoint as Breakpoint 1.) The net now looks as follows:



```
color Order = with Big | Small;
color ProductShipped = Order;
var ordent : Order;
```

At this point, the simulator has done the following (among other things):

1. Reserved the tokens that will be subtracted from `Order In` when the transition fires. Tokens that have been reserved in this way are called *input tokens* and are said to be *committed*. Their committed status is indicated by perching them on the boundary between `Order In` and the input arc.
2. Changed the current marking region of `Order In` so that it does not include the committed tokens. (This avoids confusion that could arise if the same token were represented in two multiset regions.)
3. Constructed the tokens that will be added to `Order Out` when the transition fires. Such tokens are called *output tokens*. They are shown perched on the join between `Process Orders` and the output arc.

The input tokens and output tokens are shown in regions similar to those used to indicate a current marking. Such regions are known generically as *multiset regions*. The multiset regions representing input tokens are called the *input token key region* and the *input token region*; the regions representing output tokens are called the *output token key region* and the *output token region*.

## Order of Net Execution Events

You may have noticed an inconsistency between what you are seeing now and the description of net execution in Chapter 7. The algorithm for firing a transition was there described as:

1. Rebind any CPN variables as indicated by the enabling binding.
2. Evaluate each input arc inscription.

3. Remove the resulting multiset from the input place.
4. Evaluate each output arc inscription.
5. Put the resulting multiset into the output place.

But the simulator has so far done the following:

1. Rebind any CPN variables as indicated by the enabling binding.
2. Evaluate each input arc inscription.
3. Evaluate each output arc inscription.

The explanation is this. It makes no functional difference whether the simulator completes processing of input places before turning to output places, or first evaluates all the arc inscriptions and then changes all the markings: the net ends up in exactly the same state either way. The first order is better for gaining an initial understanding of net execution, so it was used in Chapter 7. The second order produces a more useful net appearance at Breakpoint 1, so that is what the simulator uses. in practice.

### Continuing Execution

No tokens have been subtracted or added as yet, but all is ready:

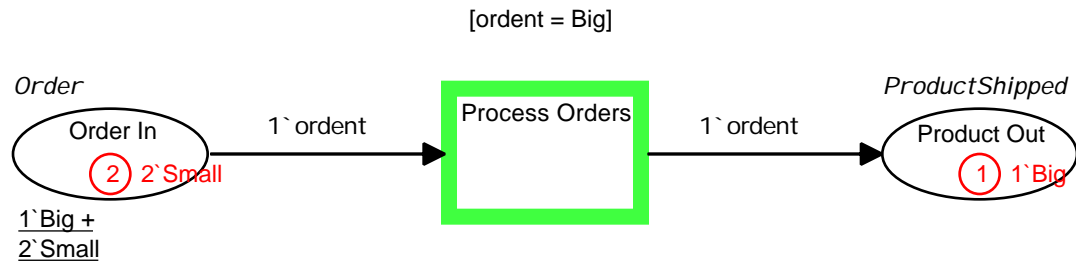
- Choose **Continue** from the **Sim** menu.

The transition fires, subtracting tokens from `Order In` and adding them to `Order Out`. The status bar displays Breakpoint 2 Step:  
1. Execution has stopped again.

### Breakpoint 2

You are at the breakpoint **End of Substep**. The net now has this appearance:





```

color Order = with Big | Small;
color ProductShipped = Order;
var ordent : Order;
  
```

The simulator has fired the transition, subtracted the input token, and added the output token. The transition is still double-highlighted, even though firing is complete. By leaving this highlighting visible at the breakpoint, the simulator makes it easier for you to keep track of which transition(s) have just fired. This can be very useful when you are studying the execution of a complex net.

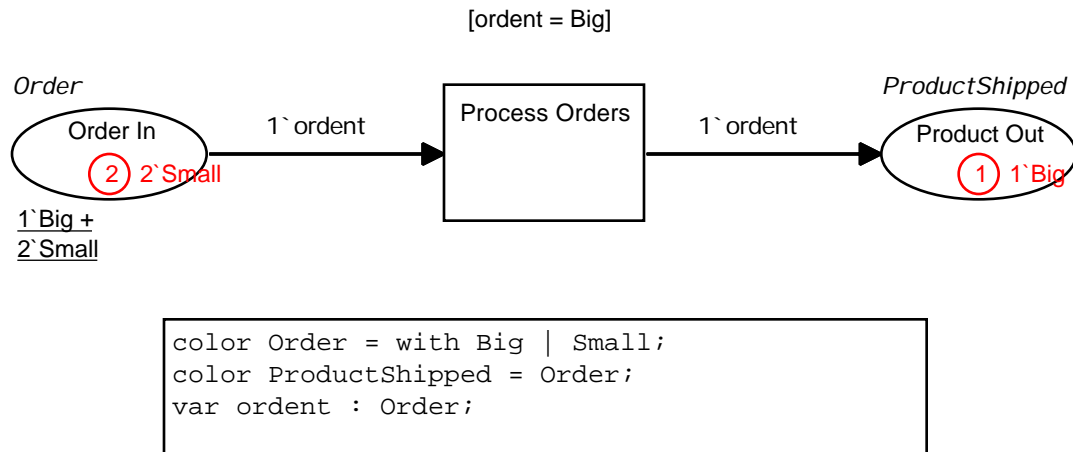
## Completing Execution

- Choose **Continue** from the **Sim** menu.

The **Step Finished** dialog appears. We already know that execution is complete (which the simulator does not yet know because it has not tried to start the next step) so there is no use continuing execution:

- Click **Stop**.

The dialog disappears, and the simulator updates the graphics to indicate the final state of the net. The net looks just as it did after it finished executing last time:



## Canceling Net Execution

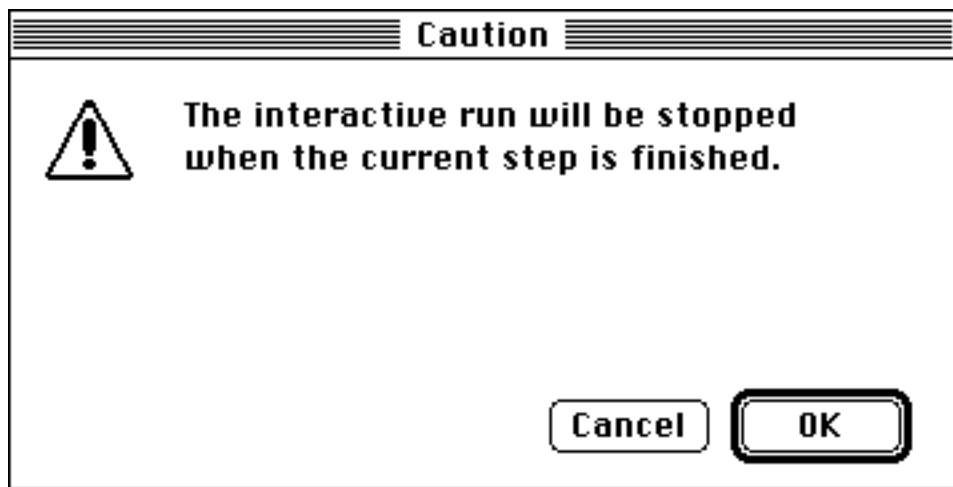
The net you have been working with is through executing after one step, so there is little advantage to canceling execution before it is complete. Realistic nets execute in many steps, and commonly execute indefinitely. Therefore a method for canceling execution is needed. Let's look at one way to do this:

- Choose **Initial State** from the **Sim** menu.
- Choose **Interactive Run** from the **Sim** menu.

Execution proceeds to Breakpoint 1.

- Choose **Stop** from the **Sim** menu.

A dialog appears:



The simulator cannot cancel execution partway through a step, because the net would be left in an intermediate state that could not correctly represent whatever the net models. Instead, it records the fact that execution is to be canceled after the current step.

- Click **OK**.
- Choose **Continue** from the **Sim** menu.

You are now at Breakpoint 2.

- Choose **Continue** from the **Sim** menu.

Execution stops at the end of the step. The **Step Finished** dialog does not appear: there is no need for it, since you have already indicated that you want to **Stop**.

You could resume execution by again choosing **Interactive Run** from the **Sim** menu. The run would proceed from where the previous run ended. However, doing so would serve no purpose at this point, since the simulator would only discover that there are no enabled transitions and hence nothing left to do.

## Leaving the Simulator

Leaving the simulator is similar to entering it. You can leave the simulator whenever execution is stopped between steps.

- Choose **Enter Editor** from the **File** menu.

The **Save State** dialog appears. Leaving and reentering the simulator restores the initial state of a net, so you will lose the current state if you do not save it before you leave the simulator. There is no need to save the state now, since it has no particular value, so:

- Click **No**.

You are now back in the editor. The net looks the same, but can no longer be executed. The **CPN** menu has replaced the **Sim** menu.

## Leaving During Execution

If you attempt to leave the simulator but find that **Enter Editor** is grayed out, the reason is that execution is not stopped between steps. Proceed to such a stopping point, and you will be able to leave the simulator.

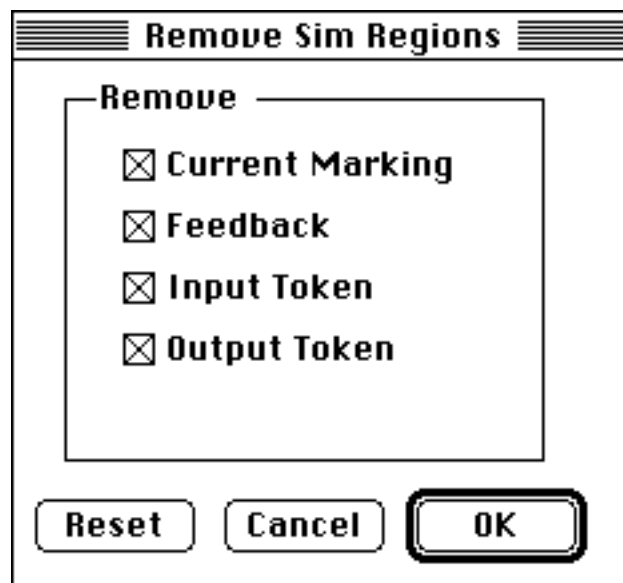
### Removing Simulation Regions

Design/CPN doesn't automatically remove simulation regions when you transfer back to the editor. You might want to keep the information in them around for some reason, perhaps to help with decisions about net modification. In this case, `Order Out` still has multiset regions showing its current marking.

If you don't want to keep leftover simulation regions after you are back in the editor, you can remove them by selecting them and deleting them with `DELETE`. However this method would be inconvenient if there were a large number of unwanted simulation regions, so Design/CPN allows you to eliminate them all in one operation.

- Choose **Remove Sim Regions** from the **CPN** menu.

The **Remove Sim Regions** dialog appears:



The four options describe various types of simulation region; the details don't matter at this point. To remove all simulation regions on the current page:

- Click **OK**.

Any simulation regions left over from net execution are gone. The results of any manual adjustments of simulation regions are gone also, whether or not the regions were visible in the editor when you removed all simulation regions. If you now re-entered the simulator and re-executed the net, the various simulation regions would appear in their default positions: with respect to simulation regions, it is as if you had never entered the simulator at all.

You can remove simulation regions from pages other than the current page. To do this, open the hierarchy page, select a group containing the page nodes for all the pages whose simulation regions you want removed, and execute **Remove Sim Regions** as described above.



# Chapter 9

## Handling CP Net Syntax Errors

You need only one more skill before you can build and execute your own CP nets with no assistance from prefabricated examples: the ability to cope with CP net syntax errors. The best way to acquire this skill is to intentionally create errors in a net that initially has none, and see what happens. The net we will use is FirstNet, the net you executed in Chapter 8.

CPN syntax errors never involve illegal net structure, because the Design/CPN editor does not allow you to create anything structurally illegal. Most CPN syntax errors involve one of the following:

- Missing net components.
- Typographical errors.
- CPN ML syntax errors.

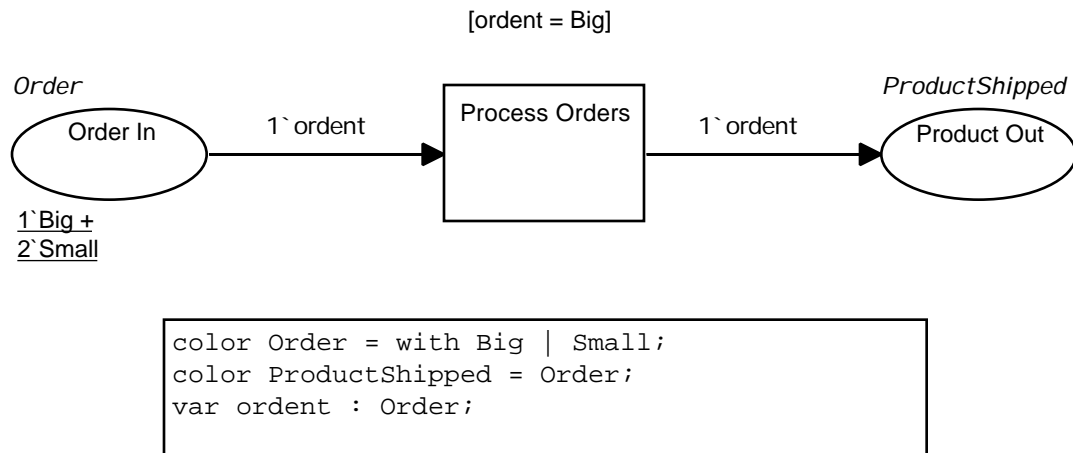
The errors you create in this chapter do not cover the spectrum of possible errors, nor do they need to. The goal here is to show you how Design/CPN responds when an error occurs. Once you have the general picture, you can deal with any particular error by just reading the resulting error message(s) and responding as appropriate.

### Opening the Net

- Open FirstNetCopy, the diagram you saved in the NewTTDiagrams directory while going through Chapter 8 (unless it is already open).

Using the copy you saved in Chapter 8 allows you to avoid having to reload the ML information and redeclare a prime page.

FirstNet in its error-free state looks like this:



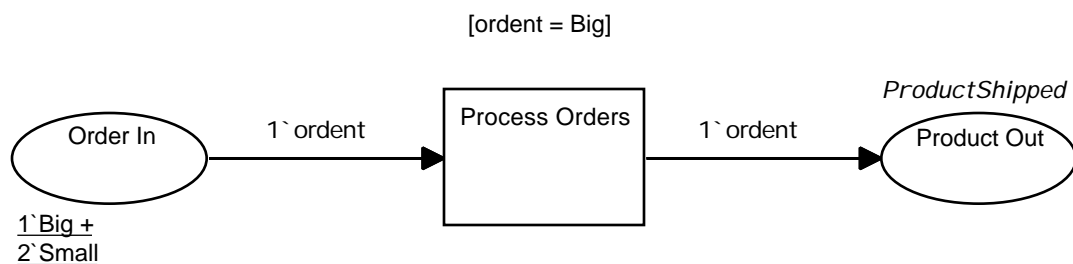
We know this net is error-free because we just syntax-checked and executed it. Let's change that.

## Missing Colorset Specification

Failure to specify a place's colorset is one of the most common syntax errors .

- Select the colorset region (*Order*) associated with *Order In*.
- Delete the region by pressing DELETE:

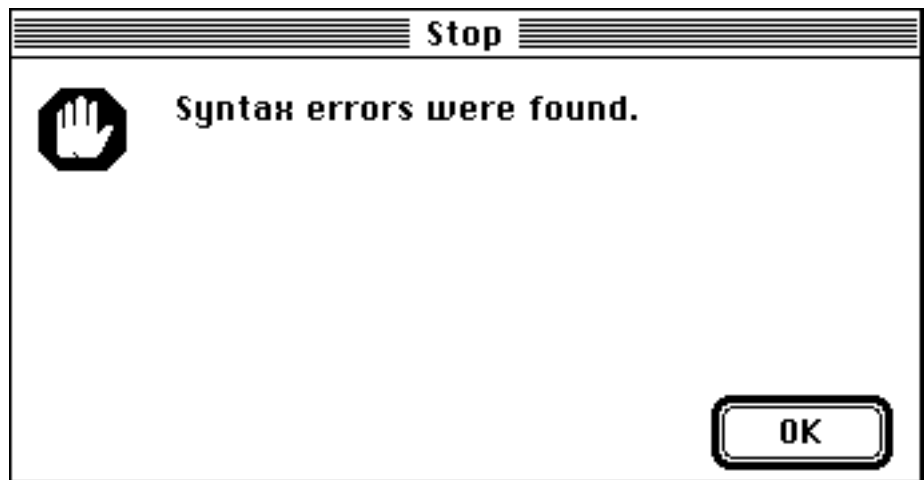
The graphical part of the net should now look like this:



- Perform a syntax check.

A dialog appears:





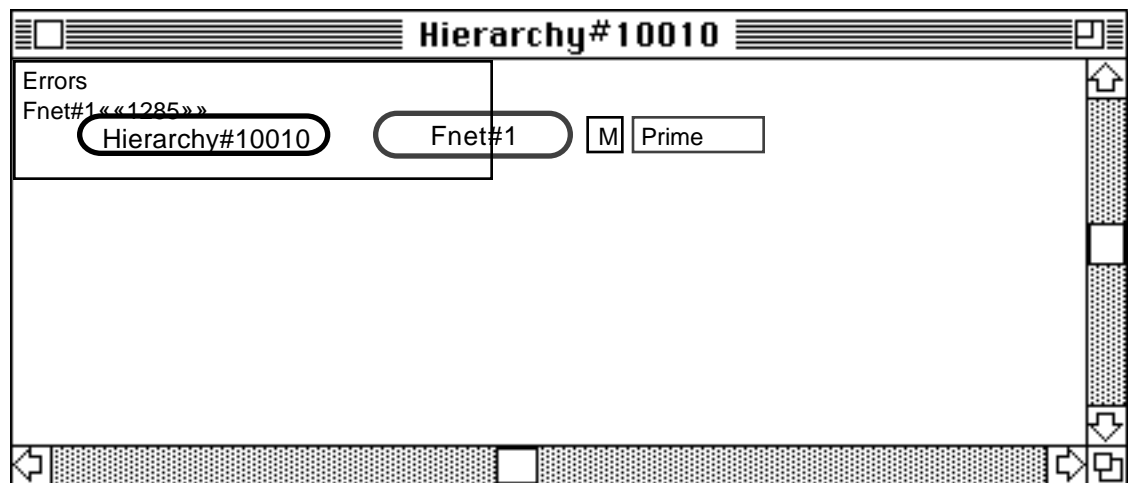
- Click **OK**.

This dialog appears whenever you request a syntax check on a diagram that is correct enough to be checkable, but that contains one or more syntax errors.

In this case, you know where the error is and what it consists of, but this would not ordinarily be the case. Unintentional errors must first be located in the diagram, then analyzed, then repaired.

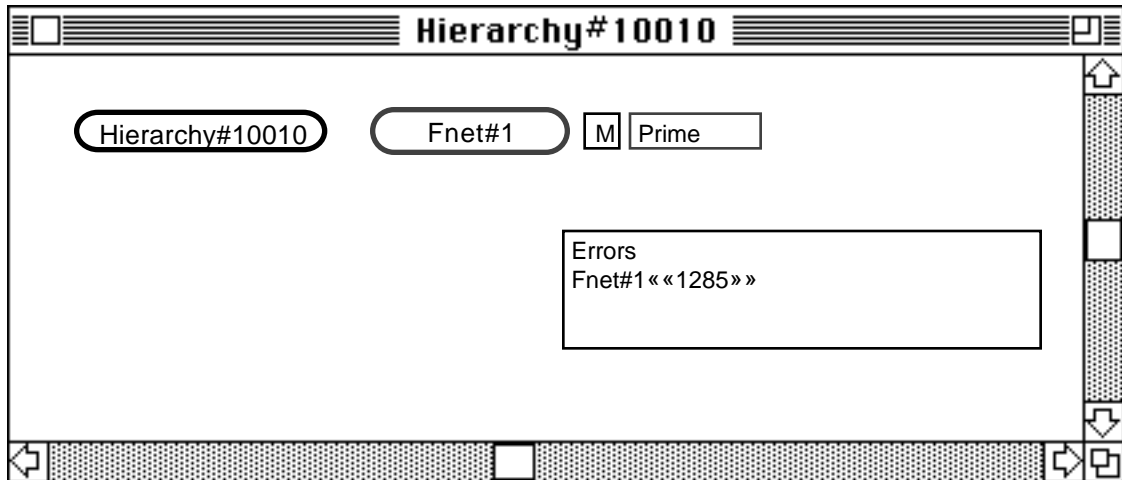
## Locating the Error

When syntax errors are found, Design/CPN brings the hierarchy page to the front, and adds to that page an auxiliary rectangle called an *error box*. This box is initially positioned in the upper left corner of the page border. Its text field identifies each page that has one or more errors.



The error box is really just an auxiliary rectangle: it can be moved and reshaped just as any rectangle can be.

- Move the hierarchy page error box away from the page nodes, and enlarge it if necessary until you can read its contents:



The text in the box indicates that there is an error on the page Fnet#1. If there were other pages with one or more errors, the name of each page would be listed below the Fnet#1 line.

We now know there is an error on the page Fnet#1. The next step is to go to that page and determine what the error is. You could just double-click on the page node for Fnet#1: the node is easy to find, since there is only one candidate. But realistic diagrams often have many pages; it would be inconvenient to have to scan the fine print in many page nodes looking for the right one every time you had to track down an error. Design/CPN provides an easier way.

### Text Pointers

Every Design/CPN error message that relates to a particular component of the diagram contains a pointer to that component. Such a pointer is called a *text pointer*. A text pointer consists of a number surrounded by angle braces. In the error message depicted above, the text pointer is ««1285»».

The number in a text pointer is just an arbitrary number that Design/CPN generates to distinguish one pointer from another. The number you see on your screen may be different from the one shown above.

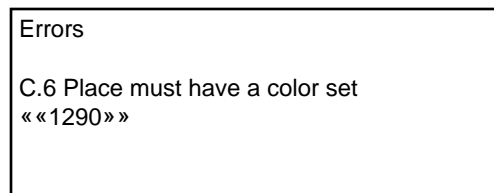
You can use a text pointer to jump directly to whatever it points to. For example, the pointer in the hierarchy page error box can take you to the page Fnet#1.

- Enter text mode.
- Place the text insertion cursor anywhere inside the text pointer.
- Press ALT-DOWN-ARROW.

Design/CPN opens the page Fnet#1.

Like the hierarchy page, Fnet#1 now has an error box, positioned in the upper left corner of the page border. That box is now the current object on Fnet#1. (The text pointer you just followed was actually a pointer to this error box, and thus to the page that contains it.)

- Leave text mode.
- Move the error box away from the diagram, and enlarge it if necessary until you can read its contents:



Now all you need to do is find a place on Fnet#1 that lacks a colorset region, and you have located the error. That is easy enough in this case — you already know, and even if you didn't, there are only two places to check. But a similar error might have happened on a page with dozens or hundreds of places. It would be inconvenient to have to examine all the places, and there is no need to: the text pointer in the error message can take you directly to the place in question.

- Follow the text pointer in the error message.

Design/CPN selects the place `Order In`. Since you are still in text mode, you could now edit the place's text field if that were appropriate.

### Fixing the Error

In this case editing `Order In`'s text field isn't the answer, so:

- Leave text mode.

All you need to do now is supply a colorset region:

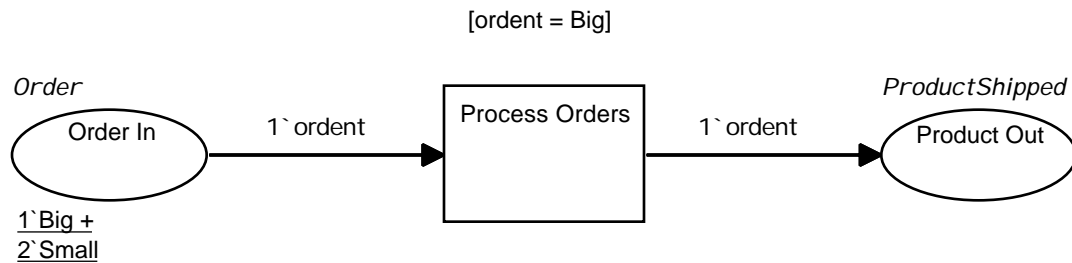
- Choose **CPN Region** from the **CPN** menu.

## Design/CPN Tutorial

---

- Create a colorset region for `Order In`, and type “Order” as the region's content.
- Leave the creation mode.

The graphical part of the net should now look as it did before you deleted the colorset region:



You have now fixed the syntax error - probably. To make sure:

- Repeat the syntax check.

You should see the **Syntax Check Successful** dialog. If you don't, something went wrong in the process of creating the colorset region. Perhaps you specified a region of the wrong type, or misspelled the colorset. In the former case, delete the region and start over. In the latter, select the region, enter text mode, and edit the name until it is correct.

## Undeclared Variables

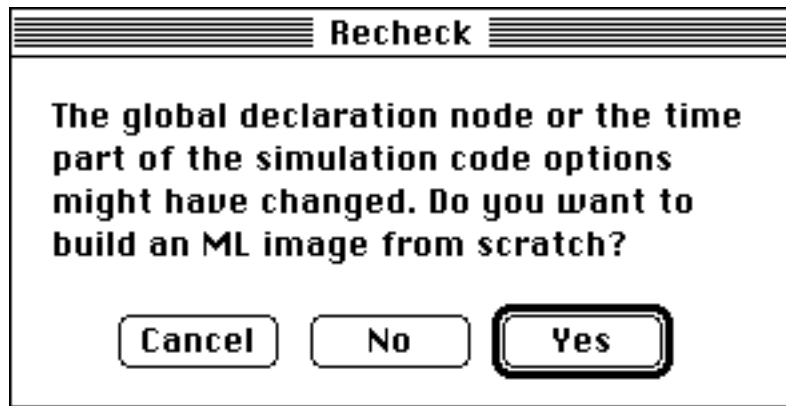
Another common error is failure to declare a variable.

- Select the global declaration node, enter text mode, and change `ordent` to `XXX`:

```
color Order = with Big | Small;  
color ProductShipped = Order;  
var XXX : Order;
```

- Do a syntax check.

A dialog appears:



The purpose of this dialog is to facilitate incremental syntax checking. Syntax-checking a large global declaration node can be time-consuming, and you might have made changes in such a node that have no bearing on your purpose in doing a syntax check. If you clicked **No**, Design/CPN would assume that the global declaration node has not changed since the last time you syntax-checked it, and the global declarations that existed then would remain in effect.

In this case rechecking the global declaration node is needed, so:

- Click **Yes**.

After a minute or so of syntax checking, the **Syntax Errors Found** dialog appears.

- Click **OK**.

### Locating the Error

As before, the hierarchy page is now on top. Since the page already had an error box, Design/CPN has not created another; it has left the existing box in the position you gave it, and has updated its text field. You may need to readjust the box to see all of the new information.

The hierarchy page error box looks the same as it did before.

- Follow the text pointer to Fnet#1.

The page Fnet#1 comes to the front.

As on the hierarchy page, Design/CPN has reused the existing error box. The box now reads:

### Errors

C.9 Guard region must be legal

Type checking error in:ordent

Unbound value identifier: ordent

[Closing <string>]

« «1303» »

If the error box is too small to show all its contents, leave text mode, reshape it, then re-enter text mode.

Use the text pointer in the error message to navigate to the diagram component it designates. That component turns out to be the transition `Process Orders`, even though the actual error, a failure to provide a variable declaration, occurred in the global declaration node.

Design/CPN does not attempt to analyze the ultimate origin of the errors it encounters: doing so would be unacceptably time-consuming, and could not produce reliable results in any case. Instead it indicates the location at which it encountered a problem, leaving it to you to determine whether the problem originates there or is a consequence of a problem somewhere else.

In this case, Design/CPN happened to check the transition before checking either arc inscription, and found an unbound variable in the guard. The error message therefore points to the transition. It is up to you to determine whether the wrong variable is used in the guard, or the right variable is used but has not been declared: this is a semantic rather than a syntactic question, so Design/CPN cannot determine the answer.

## Fixing the Error

In this case, editing the transition's text field is not the answer, but editing the global declaration node is. You are already in text mode, so:

- Select the global declaration node.
- Edit it so that it again reads:

```
color Order = with Big | Small;  
color ProductShipped = Order;  
var ordent : Order;
```

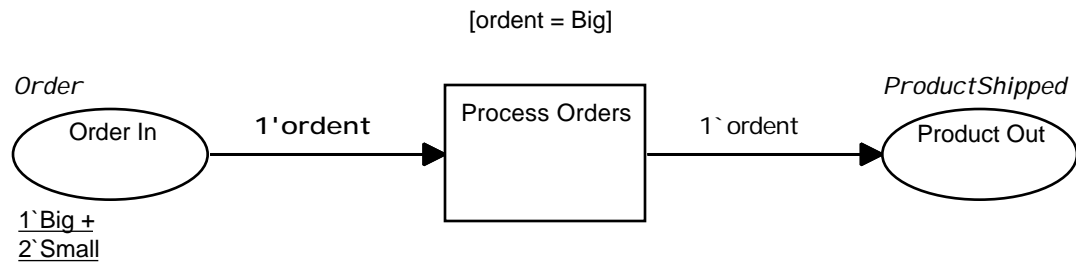
- Do a syntax check.

You should see the **Syntax Check Successful** dialog. If you don't, continue to work on the error until you do.

## Illegal CPN ML Constructs

The CPN ML parts of a CP net can experience the same kinds of errors that can occur with programming languages generally. Sometimes an almost invisible error can occur: Design/CPN's way of handling errors can lead you directly to it.

- Change the backquote ( ` ) in the input arc inscription between *Order In* and *Process Orders* to a single quote ( ' ):



- Do a syntax check.

The Fnet#1 error box reads:

```
Errors

C.11 Arc expression must be legal

Parse error:
  Was expecting ",'"

In: ... fun CPN'AF6 ( CPN'bb : CPN'BT4 ) : Order
ms = 1 <?> 'ordent

[Closing <string>]
[Closing <string>]
```

The description in the error box is not that informative. It describes the error as it appeared to the CPN ML syntax checker, not as it appears to a human observer. But:

- Follow the text pointer in the error box.

The input arc becomes the selected object.

- Select the input arc inscription region (You can select it without leaving text mode - just click on it.)

Even if you didn't already know what was wrong, a little examination would soon reveal it.

- Restore the single quote to a backquote.
- Run another syntax check.

This time there should be no errors.

## Conclusion

The three errors you have worked with in this chapter illustrate the three basic types of errors that happen with CP nets: missing net components, typographical errors, and CPN ML syntax errors. The techniques you used to investigate the errors are applicable to all CPN errors.

Of course, an introduction such as this chapter provides cannot create instant proficiency in error handling. Real proficiency comes only with time and experience: a tutorial cannot supply it.

As you proceed through this tutorial, you will build a series of increasingly complex nets. The potential for errors will increase correspondingly. Careful attention to detail will minimize the likelihood of errors. When they do occur, the techniques shown in this chapter can help you to track them down.



# **PART 2**

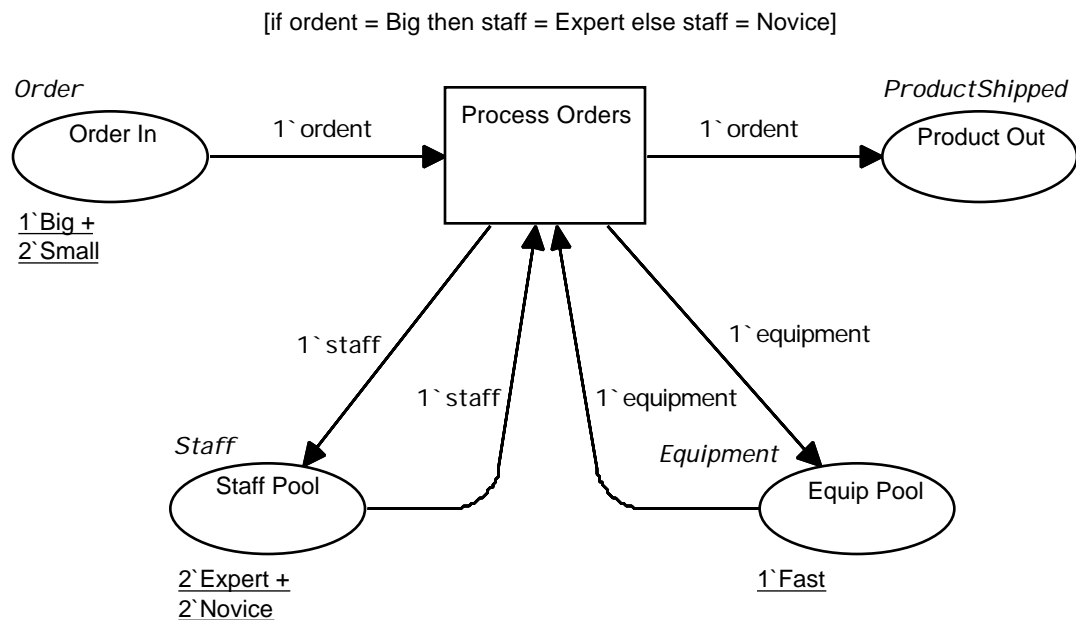
## **Design/CPN Techniques**

# Chapter 10

## Extending a CP Net

Everything we have done so far has been based on the same net: FirstNet. This net is useful for demonstrating essential points about CP nets and Design/CPN, but not much else. In particular, it sheds no useful light on how a CP net can be used to model a real system.

Let's begin to remedy that now, by extending FirstNet until it becomes SalesNet, the net that appeared, but was not explained, in Chapter 1:



```

color Order = with Big | Small;
color ProductShipped = Order;
color Staff = with Expert | Novice;
color Equipment = with Fast | Slow;

var ordent : Order;
var staff : Staff;
var equipment : Equipment;
  
```

### Building SalesNet

You have already practiced most of the skills you need to create and execute SalesNet. Instructions on how to use these skills will not be repeated. If you are not sure how to do any of the things that this chapter assumes you can do, you should review earlier chapters as needed to gain the needed information.

- Open NewFirstNet the diagram you created in Chapter 6 and saved in the NewTTDiagrams directory.

### Modifying the Global Declaration Node

To make room for the places to be added to the graphical part of the net:

- Move the global declaration node down about two inches.

To make room for the additional declarations:

- Reshape the global declaration node to be about two inches tall.

To add the additional declarations:

- Enter text mode and edit the global declaration node so that it looks like this:

```
color Order = with Big | Small;  
color ProductShipped = Order;  
color Staff = with Expert | Novice;  
color Equipment = with Fast | Slow;  
  
var ordent : Order;  
var staff : Staff;  
var equipment : Equipment;
```

### Modifying the Guard

You can change the contents of any region whenever you are in text mode in the editor. You should still be in text mode after editing the global declaration node.

- Click the mouse on the guard.

The guard is now the current object, and you can edit it.

- Edit the guard so that it looks like this:

[if ordent = Big then staff = Expert else staff = Novice]

- Leave text mode.

### Extending the Graphics

There is nothing new in the graphics: it is just more places, arcs, and regions, no different from those you created when you build NewFirstNet. Don't worry about perfecting minor details of appearance. Techniques for polishing a net's appearance are covered in Chapter 11.

When you are done, the net should look as shown at the beginning of this chapter.

### Performing a Syntax Check

To verify that the extended net is correct:

- Choose **Syntax Check** from the **CPN** menu.

If any syntax errors are found, correct them before you go on. If you cannot get a successful syntax check, open SalesNetDemo in the TutorialDiagrams directory and compare that version of SalesNet to your own. When your net passes the syntax check:

- Save your net in NewTTDiagrams under the name NewSalesNet.

## Discussion of the Model

SalesNet is a high-level model of a system we will call the Order Processing System. The model is “high level” in that it represents the whole system as a single activity, Process Orders, and makes no attempt to show the particular operations through which the orders are processed. Let's take a look at the system first, and then see how SalesNet models it as a CP net.

### Description of the System

The Order Processing System inputs orders, processes them in some way that uses staff and equipment, and ships product for each order. There are two types of order, big and small; two types of staff, expert and novice; and two types of equipment, fast and slow. The origin of the orders, and the nature of the product shipped, are unknown.

Each order is handled by one staff member, uses one piece of equipment, and results in one shipped product. Big orders are processed differently from small orders: a big order must be handled by an expert staff member, while a small order must be handled by a novice staff member. There is no requirement that big or small orders use any particular type of equipment: any equipment will do.

Obviously this is far from a complete description of a viable system for processing orders. That is not significant at this point: it is description enough for our current purposes. We will specify considerably more detail later in this chapter, and extend the model accordingly.

### How SalesNet Represents the System

You can probably tell a lot about how SalesNet models the Sales Order System just by looking at the net, but let's go over it completely to be sure that everything is clear.

#### Entities and Colorsets

There is a colorset in the model for each entity in the system, and each colorset provides the values necessary to represent the distinctions we want to make:

```
color Order = with Big | Small;
color ProductShipped = Order;
color Staff = with Expert | Novice;
color Equipment = with Fast | Slow;

var ordent : Order;
var staff : Staff;
var equipment : Equipment;
```

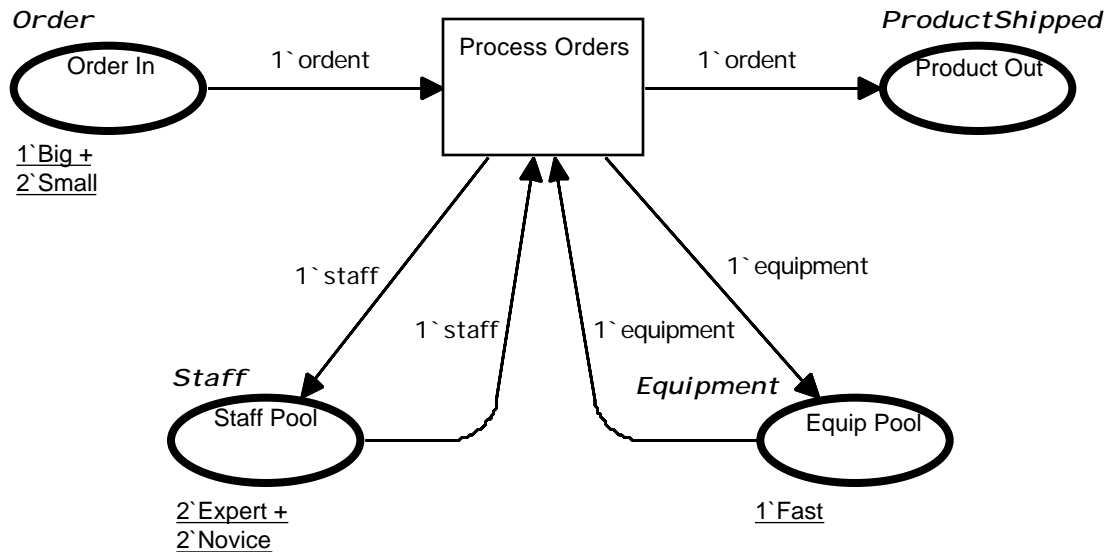
We could make finer distinctions by adding more values to the enumerations, or shifting over to integer colorsets; or more detailed descriptions, by using record colorsets with fields for various things we want to specify; or we could have even coarser colorsets, that offered only one type of order, one type of staff, and one type of equipment.

However, the colorsets as specified represent the system insofar as we know of it, neither blurring distinctions we want kept nor offering more distinctions than we need, so they are appropriate.

## Locations for Storing Data

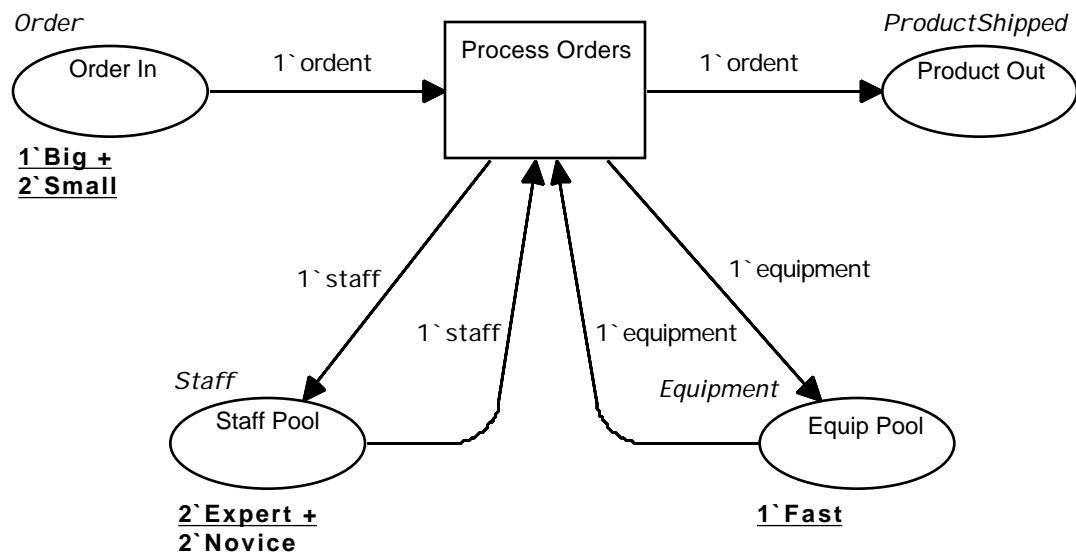
In this high level view, we don't represent any details about what happens as an order is processed. Therefore we don't need to represent any data about what entities do. All we need to know is where they are. Therefore there is a place for entities of each colorset:

[if ordent = Big then staff = Expert else staff = Novice]



The description of the system didn't specify anything about how many there are of any type of entity, so arbitrarily chosen values appear, represented as initial marking regions:

[if ordent = Big then staff = Expert else staff = Novice]

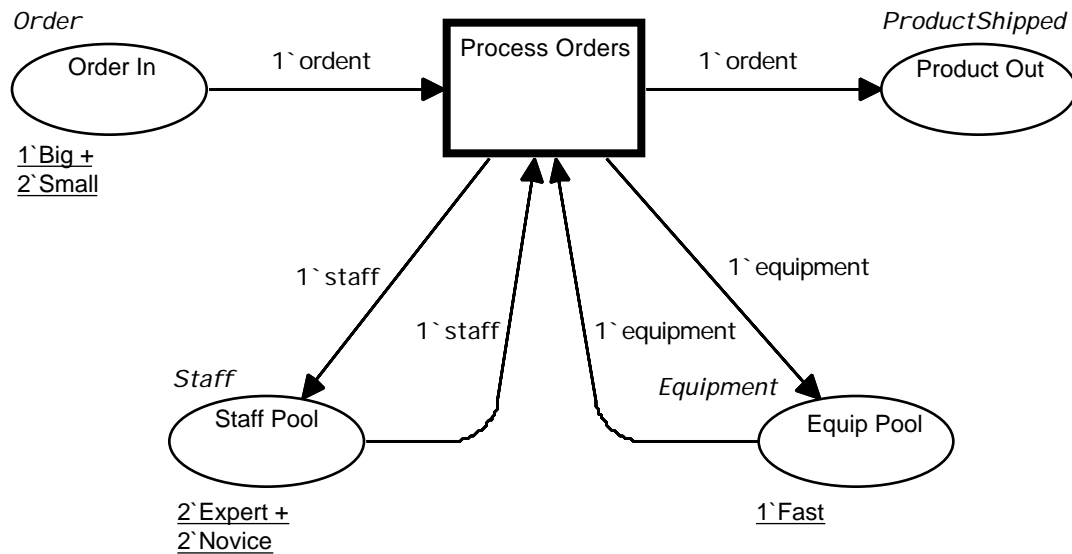


These initial values are one of the things we might vary while experimenting with the model, to try to determine their effect on system throughput.

### Activities for Transforming Data

There is only one activity, representing the entire system, so there is only one transition: `Process Orders`.

[if `ordent` = Big then `staff` = Expert else `staff` = Novice]

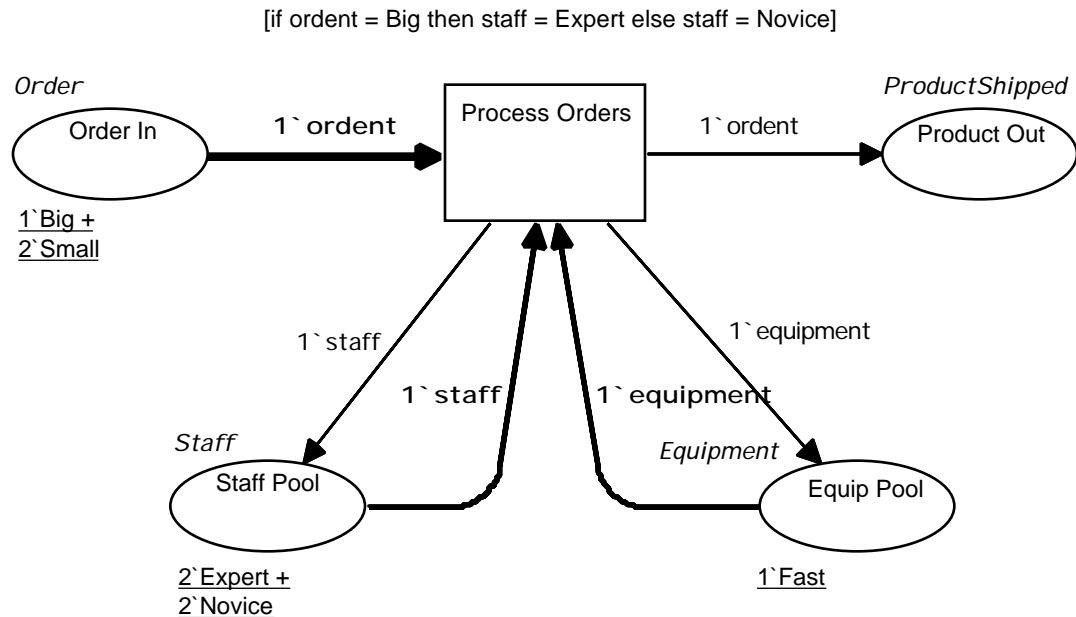


### Data and Conditions Needed for Activities to Occur

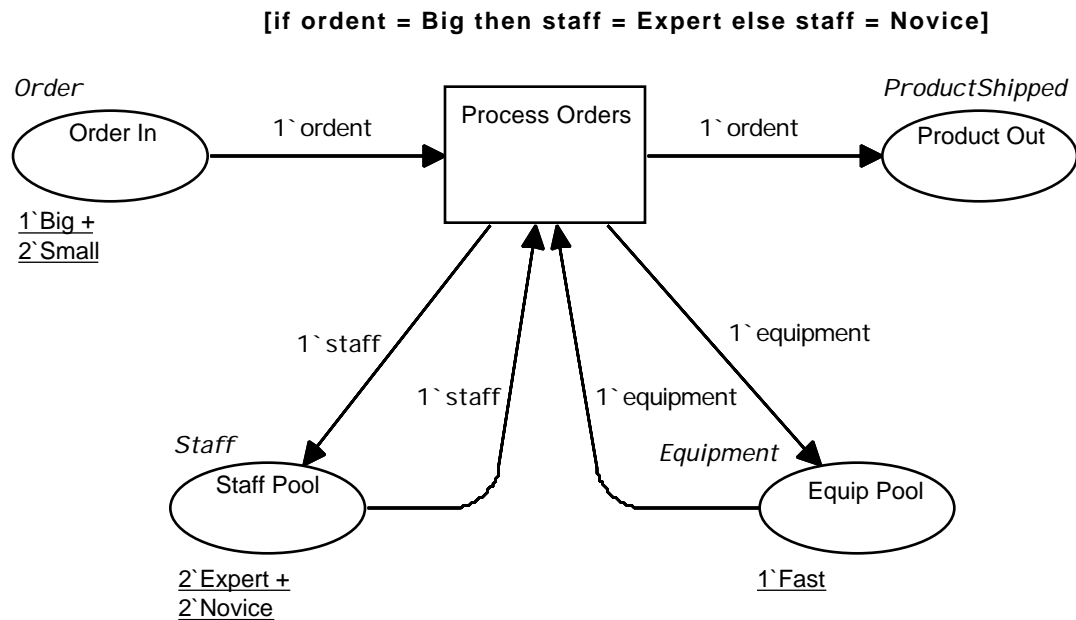
In order for `Process Orders` to fire, there must be:

1. At least one `Order` in `Order In`.
2. At least one `Staff` member of appropriate type in `Staff Pool`.
3. At least one piece of `Equipment` in `Equip Pool`.

The numbers of each entity needed are specified by the input arcs and arc inscriptions:



The matching of Expert staff with Big orders, and Novice staff with Small orders, is performed by the guard:



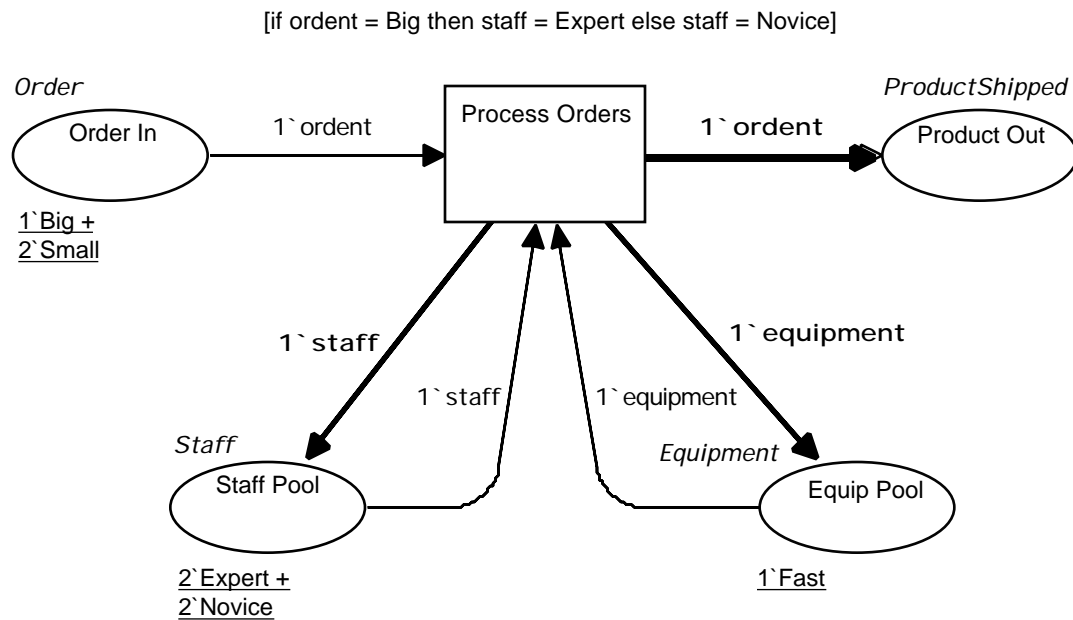
This guard will evaluate to **true** only when `ordent` is bound to Big and `staff` is bound to Expert, or `ordent` is bound to Small and `staff` to Novice. Any other combination of bindings will cause the guard to evaluate to **false**, so the transition will not be enabled with that binding. When the simulator checks `Process Orders` for enablement, it will try various bindings for `ordent` and `staff`, until it finds one that satisfies it (makes it true) or runs out of bindings to try.



The ability of a guard to constrain bindings of variables on more than one input arc was not demonstrated in Chapter 7, because there is only one input arc in *FirstNet*, but the practice does not introduce anything new. We saw in that chapter how a guard can be used to constrain two variables bound on the same arc. Using it on variables bound on different arcs is no different.

### Data That Will Be Produced if an Activity Occurs.

This is specified by the output arcs and arc inscriptions:

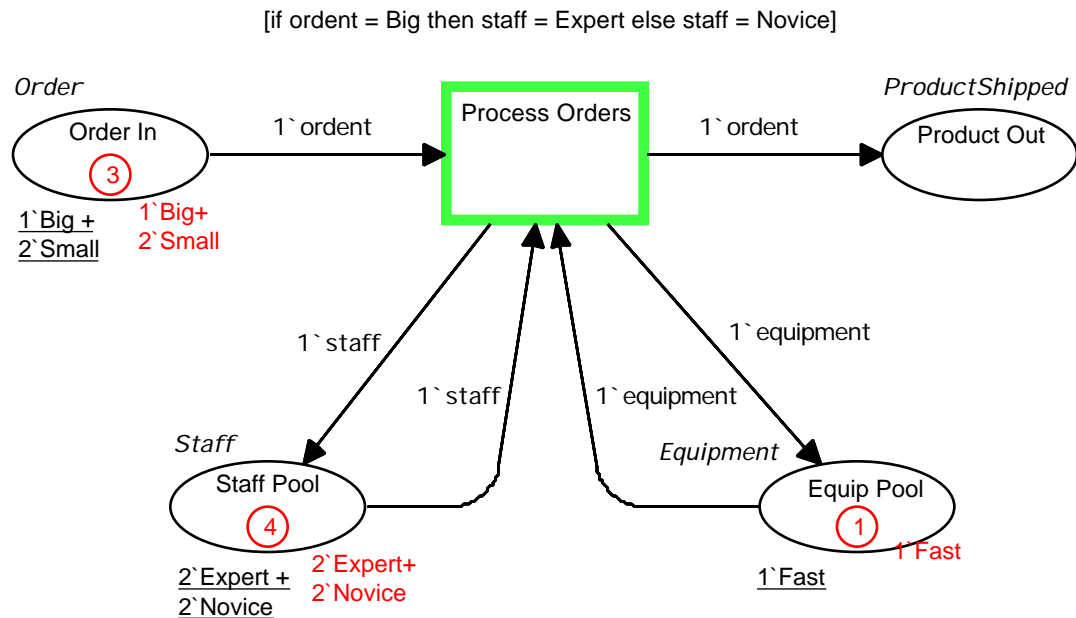


These arcs and inscriptions specify that if *Process Orders* fires, the firing will put a *Staff* token into *Staff Pool*, an *Equipment* token into *Equipment Pool*, and a *ProductShipped* token into *Product Out*.

The *Staff* tokens will have the value to which *staff* was bound as part of the enabling binding, and similarly for the *Equipment* token. The *ProductShipped* token will have the value to which *ordent* was bound in the enabling binding. Thus a *Big* order is big in that it results in a *Big* product, whatever that might be.

## What Happens When SalesNet Executes

After initial tokens are generated, but before any transitions fire, the SalesNet looks like this:



As SalesNet executes, Process Orders will fire once for each order in Order In. Each time it fires, the simulator will do the following:

### Rebind Any CPN Variables Per the Enabling Binding

Remember that a transition doesn't necessarily fire as soon as it is found to be enabled. The enabling binding has to be restored before firing can proceed.

Let's suppose that the enabling binding was:

```
ordent = Small
staff = Novice
equipment = Fast
```

The three CPN variables become bound to those values.

### Evaluate Each Input Arc Inscription

Given the bindings above, the input arc inscriptions evaluate to:

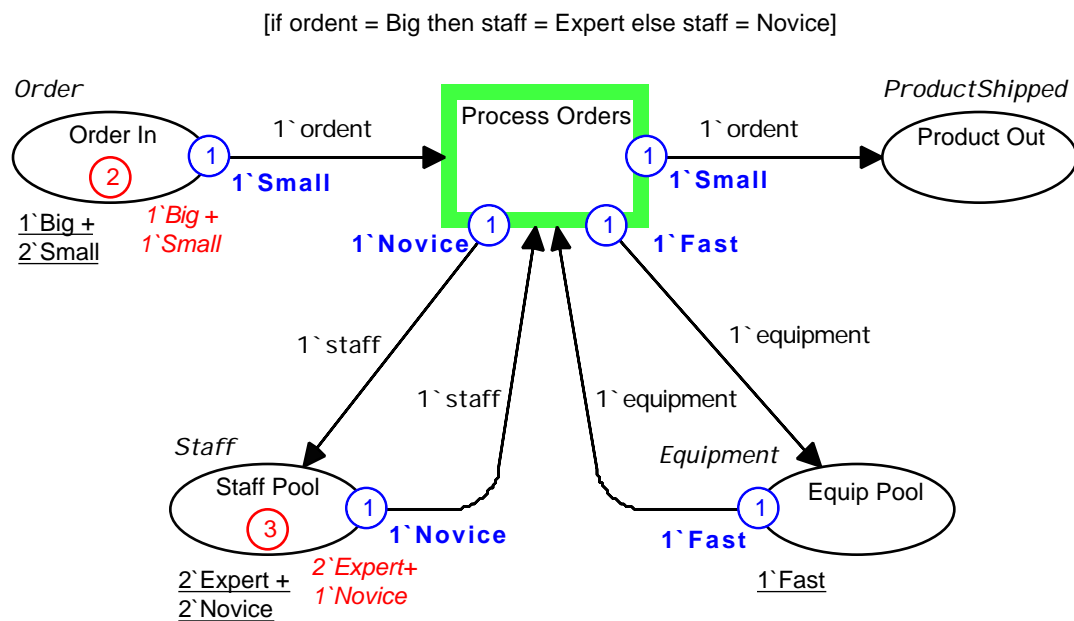
```
1`Small
1`Novice
1`Fast
```

### Evaluate Each Output Arc Inscription

Given the bindings above, the output arc inscriptions also evaluate to:

1`Small  
1`Novice  
1`Fast

Note that we're using the simulator's ordering here. If we were executing SalesNet in the simulator (which we soon will be), and stopped at Breakpoint 1 (Before Any Tokens Are Moved), the net would look like this:

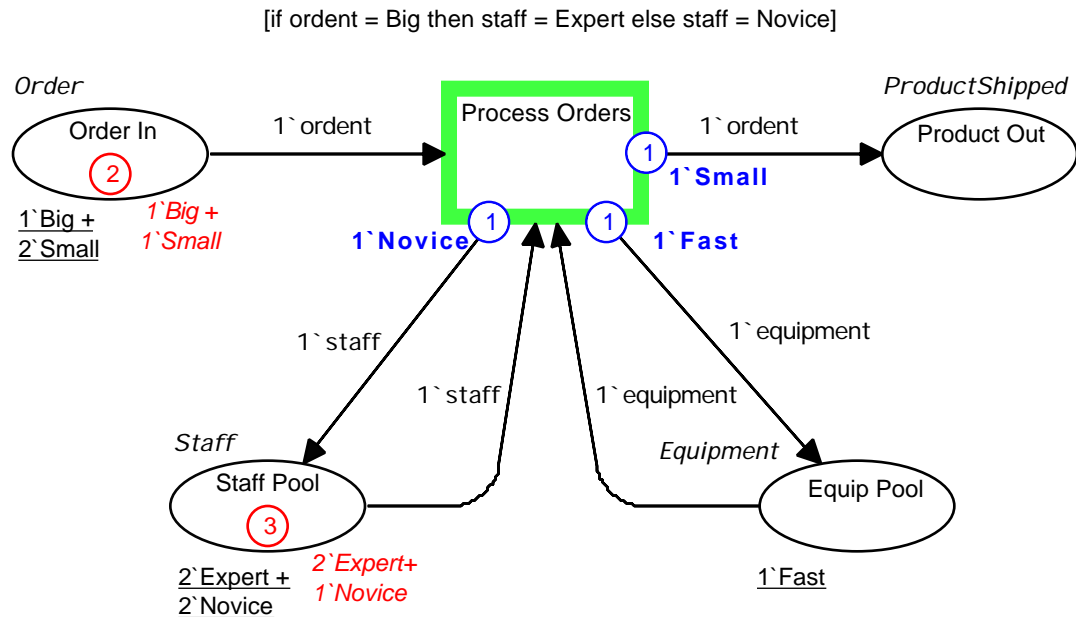


There are a lot of details here. To help you disentangle them, the regions that show tokens that are about to be removed from or added to places have been **boldfaced**, and the regions that show current markings have been *italicized*. This figure follows the simulator's convention that tokens about to be removed from an input place are not shown in the place's current marking, even though they have not actually been removed yet.

Please study this figure carefully. If you understand everything about it, you understand all the material that this tutorial has presented so far. If not, you should review as needed until you know what everything is in this figure, and why it is that way.

### Remove the Enabling Multiset from Each Input Place

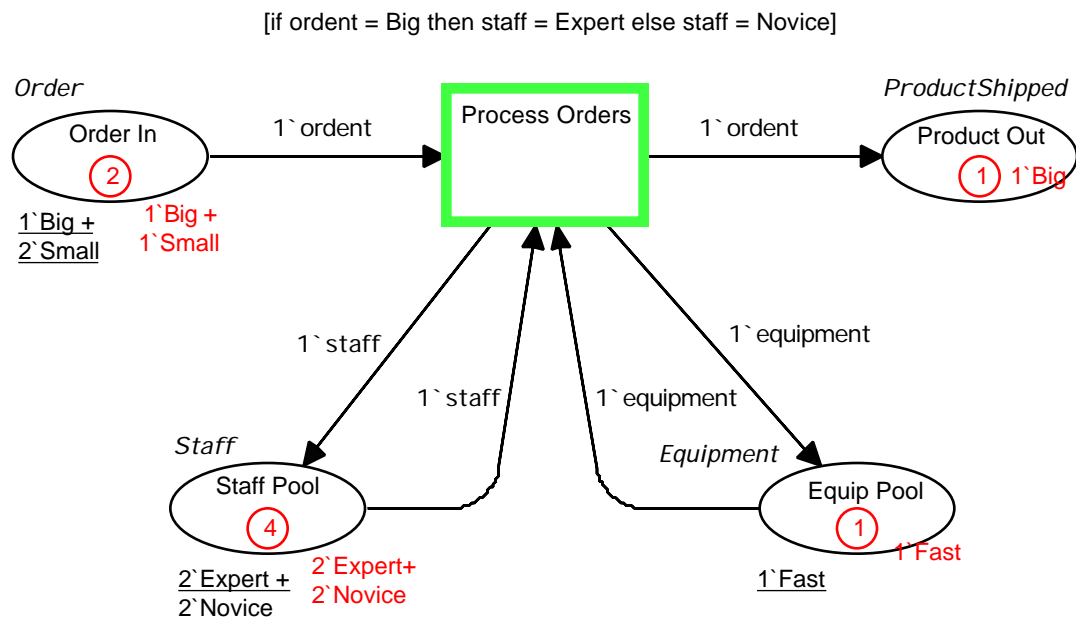
After the input tokens have been removed, but before any output tokens have been added, the net looks like this:



There is currently no token in Order In for the Small order, in Staff Pool for the Novice staff member, and in Equip Pool for the Fast equipment piece. The tokens have been subtracted and thrown away.

## Put the Output Multiset into Each Output Place.

After the output tokens have been added, the net looks like this:



## Design/CPN Tutorial

The `Novice` staff and the `Fast` equipment have been restored to their pools, and are available for use with subsequent orders. The `Big` order that was just processed is now represented in `Product Out`.

### Continue Execution

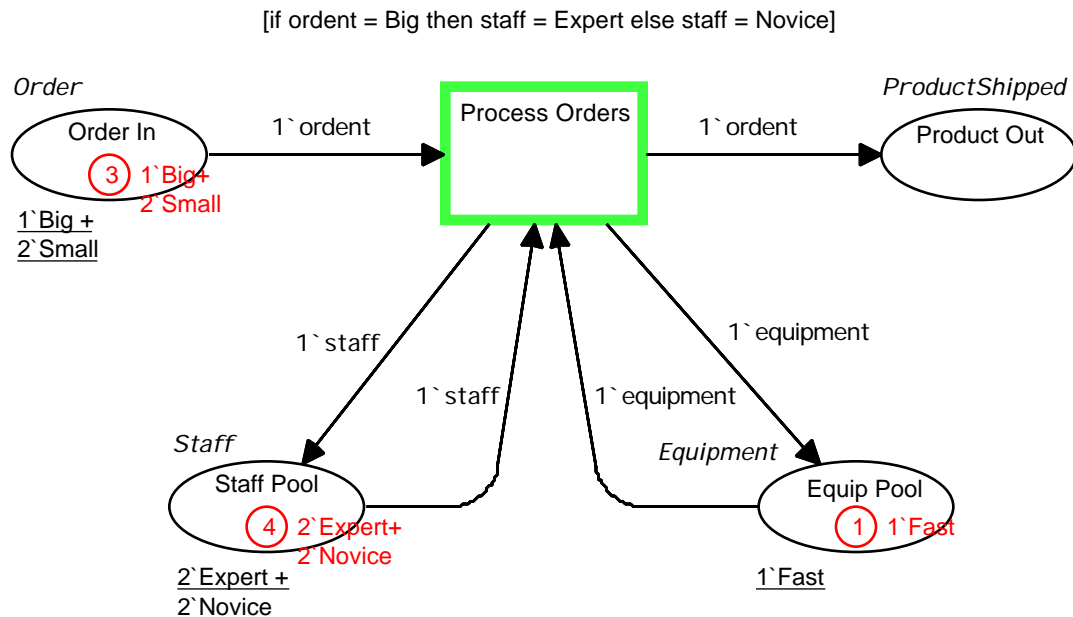
The firing of `Process Orders` is now complete. But it is still enabled, so it will fire again. After it has fired twice more, once for each job remaining in `Order In`, it will cease to be enabled, because there will be no `Order` tokens left to satisfy the input arc inscription `1`ordent`. Execution will then be complete, because there will be no more enabled transitions.

## Executing SalesNet

Let's take SalesNet into the simulator and see what it does. You should have a pretty good idea already, but there are always some surprises.

- Enter the simulator, as you did in Chapter 8. (Review that chapter if necessary.)

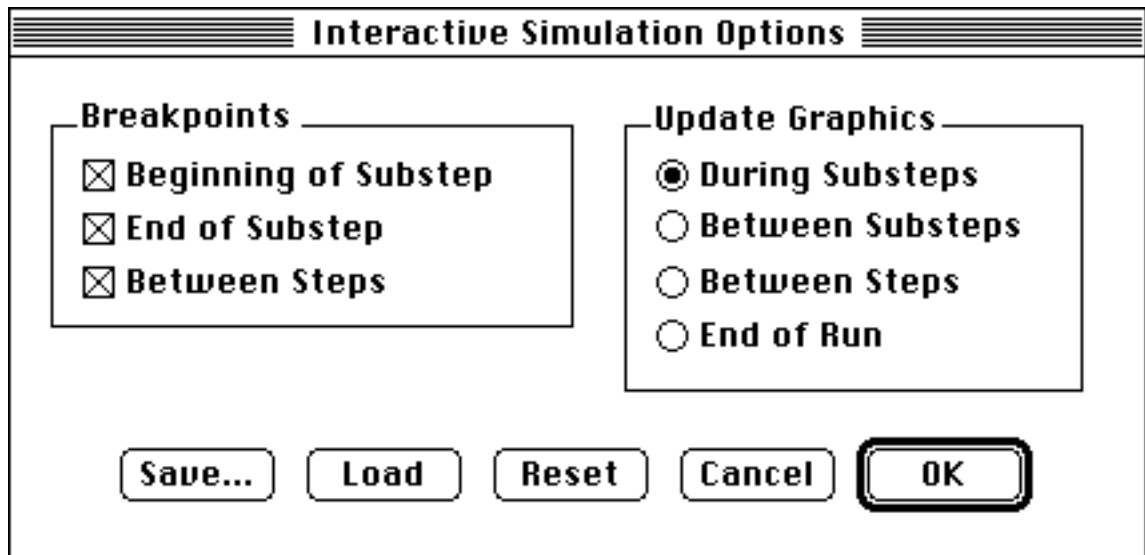
In the simulator, SalesNet looks like this:



### Setting Substep Options

For starters, let's set the same substep options that we used with FirstNet in Chapter 8.

- Choose **Interactive Simulation Options** from the **Set** menu.
- Click options as needed to set all three breakpoints:



- Click **OK**.

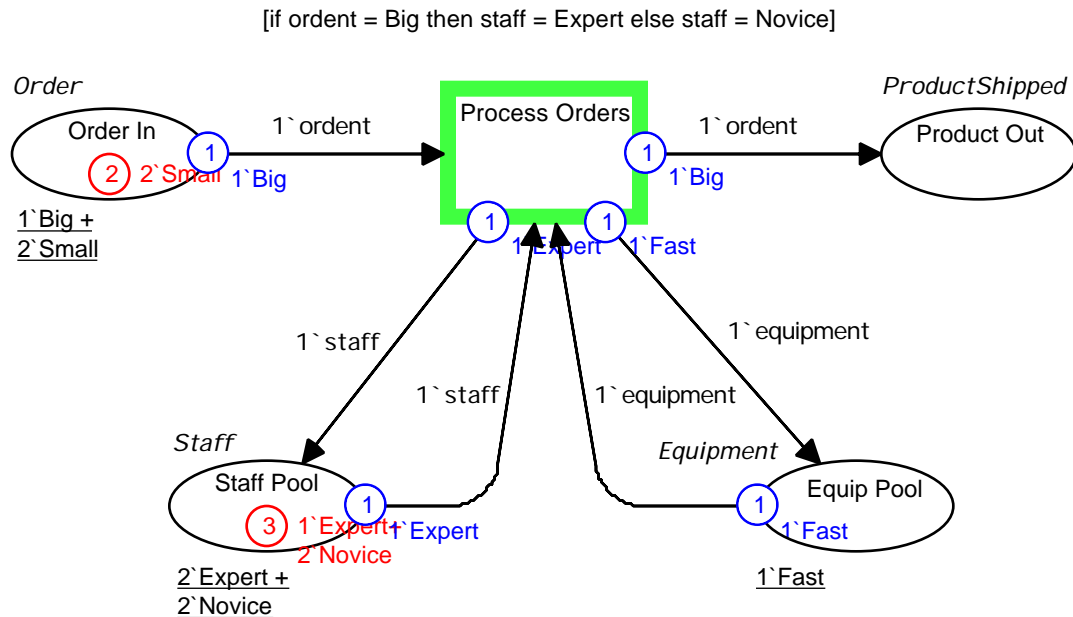
### Adjusting Simulation Regions

We'll need to do a little work to make the executing SalesNet look as good as it did in the figures earlier in this chapter.

- Choose **Interactive Run** from the **Sim** menu.

When execution stops at Breakpoint 1, the net should look like this:

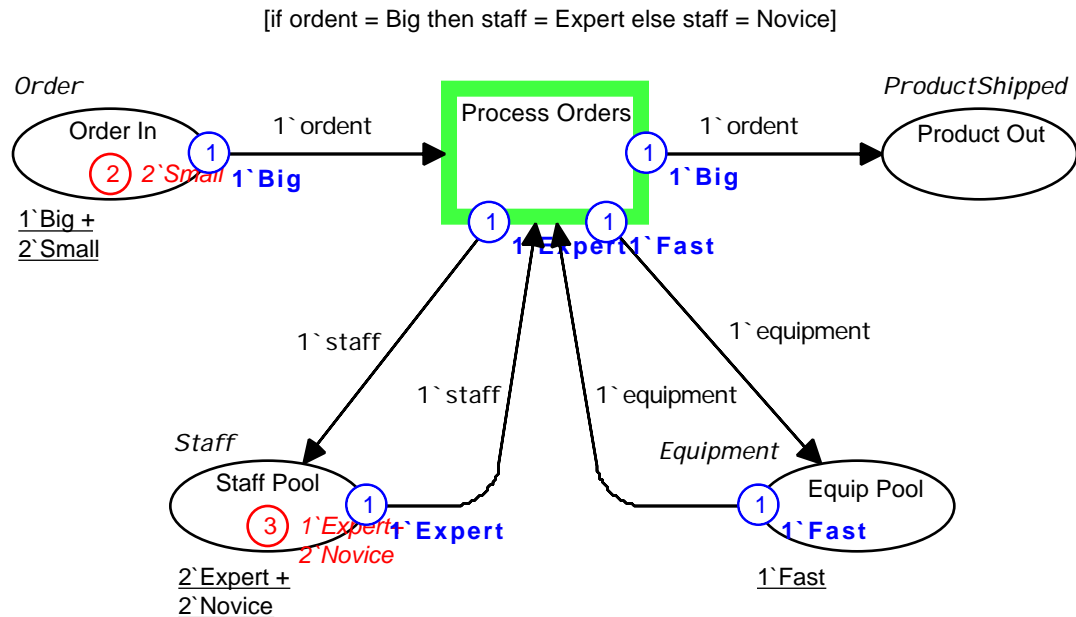
## Design/CPN Tutorial



What you see may be different from the above, because the simulator may have happened to start with a *Small* job, and/or with *Fast* equipment. It uses a random number generator to make arbitrary choices of this kind.

This is obviously not a very clear presentation of current markings, input tokens, and output tokens. The problem is that there is no algorithmic way for Design/CPN to decide where to put simulation regions. Therefore, it just follows a formula, and leaves it to the user to make adjustments as needed.

To help you see what is what, here is the net again, with input and output token regions again in **boldface**, and current marking regions in *italics*:



## Key and Popup Regions

You may have noticed a regular pattern in the names given to the current marking, input token, and output token regions in Chapter 8. There is always a *key region*, a number in a circle that tells the number of tokens in the multiset, and an associated region that shows the exact composition of the multiset. These associated regions are called *popup regions*.

Popup regions are so called because they can be made to appear (“pop up”) and disappear at any time by double-clicking on the associated key region. This trick works in both the editor and the simulator.

- Double-click on one of the key regions in SalesNet.

The associated popup region disappears.

- Double-click on the key region again.

The popup region reappears.

Popup regions are often a great convenience: the detailed information in the popup region can be displayed when it is useful, and hidden when it is not. When it is hidden, the key region remains to provide a summary and to permit quick access to the complete information in the popup region.

Popup regions are used for more than just representing multisets. For instance, the Page Mode Region that appeared on the hierarchy page when you made Fnet#1 a prime page is actually a popup re-



gion, and would disappear and reappear if you double-clicked the associated Page Mode Key Region. Whenever a region is described as a key region, there is an associated popup region, and vice versa.

The observability of a complex net can often be greatly enhanced by hiding all popup regions except those that are of immediate interest. But right now, all the regions are of interest, so there would be little advantage in hiding the popups. Instead, let's look at how they can be repositioned to provide a better appearance when they are displayed.

### Repositioning Simulation Regions

Key and popup regions are just ordinary graphical objects. You can move them anywhere you like by dragging them with the mouse. A popup region is a region of its key region: you can drag the key region and the popup will follow:

- Drag one of the key regions to some other location.

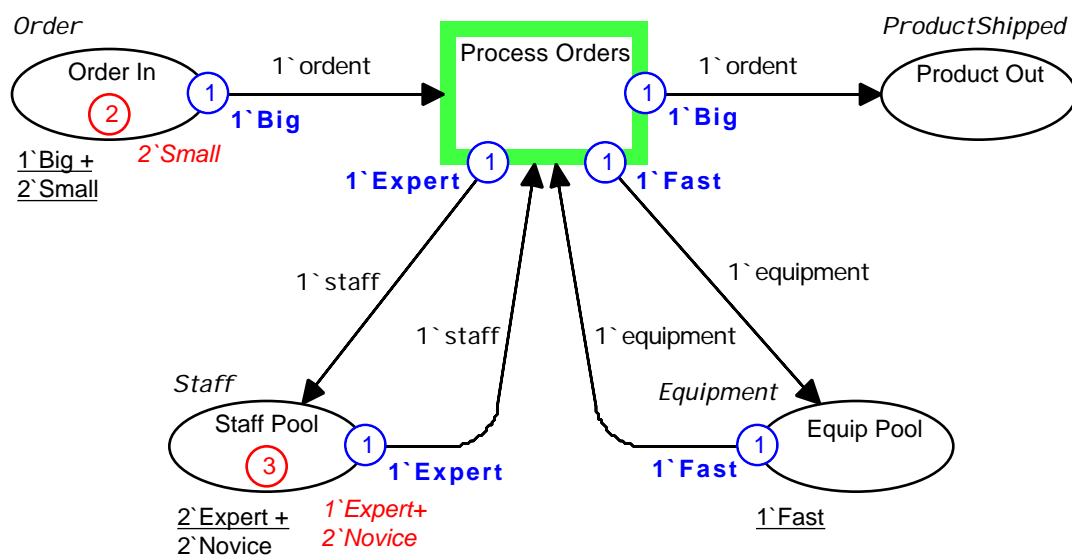
The popup tracks its parent.

- Drag the key region back to its original position.

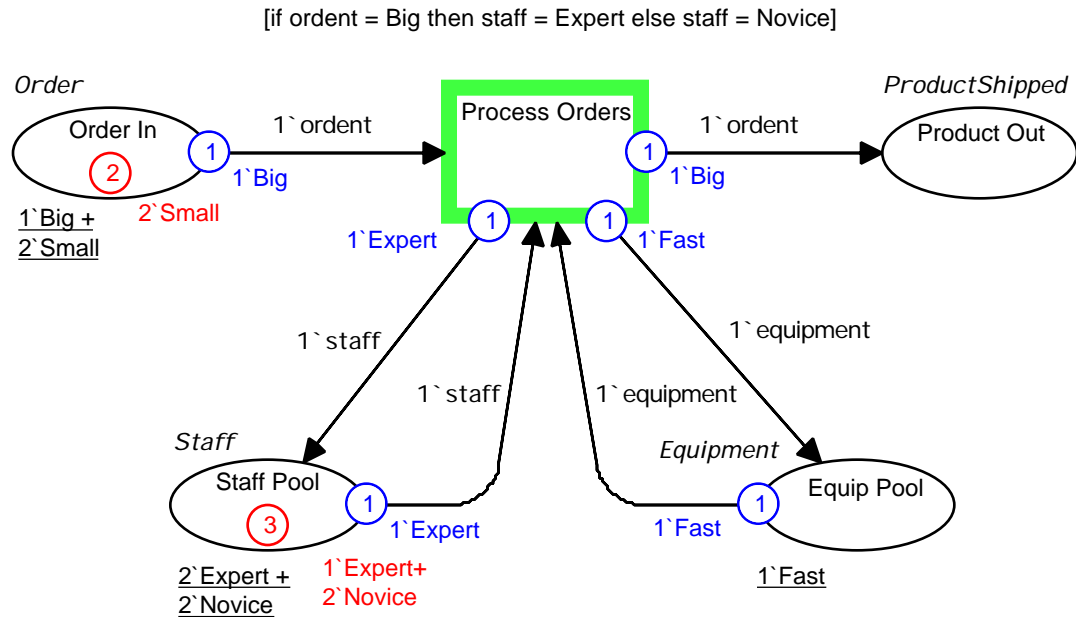
As it happens, the various key regions in SalesNet are in good positions. Their popups however are not. Let's change that.

- Drag the various popup regions to the positions indicated in this figure:

[if ordent = Big then staff = Expert else staff = Novice]



If you accidentally grab the wrong object with the mouse, just put it back and try again. When you have moved all the regions, the net should look like this:



### Continuing Execution

There isn't any need for detailed instructions at this point. Continue execution until it is complete, return to the initial state, and execute the net again, until you are entirely familiar with what is going on. As you gain experience with CP nets, you will find it increasingly easy to tell which component is which, and to decide where to put components for maximum clarity.

## Creating a Page for Global Declarations

SalesNet, and its predecessor FirstNet, are each contained on a single page. This page contains both the data declarations for the net (in the global declaration node) and all graphical structure that makes use of those declarations.

Putting the global declaration node on the same page as a net's graphics can be helpful when you are first learning about CP nets, but otherwise it is not the best practice. As a net grows, the global declaration node ends up competing with it for page space, and has to be move aside where it can only be seen by scrolling over to it. Furthermore it is of little interest once you know what is in it. Therefore it is customarily kept on a page of its own, where it can be quickly accessed when needed and ignored otherwise.

### Creating a New Page

Putting the global declaration node on a page of its own is just a matter of creating the page, and then creating or moving the declaration node there. Let's move SalesNet's global declaration node to its own page.

- Leave the simulator.
- Choose **New Page** from the **Page** menu.

A new, blank page appears. Its name is New#2.

### Naming the Page

That name does not tell much about the page's intended purpose, so let's rename it to something meaningful.

- Choose **Page Attributes** from the **Set** menu.

The **Page Attributes** dialog appears:

**Page Attributes**

Name & No  #

☒ Change Current Page  
☐ Save as Defaults

**Page Kind**  
☒ Standard  
☐ Palette

**Page Border**  
☒ Visible  
☐ Invisible

Page Width   
Page Height

The page name field contains the current page name, New.

- Edit the page name field to specify the name “Declare”.
- Click **OK**.

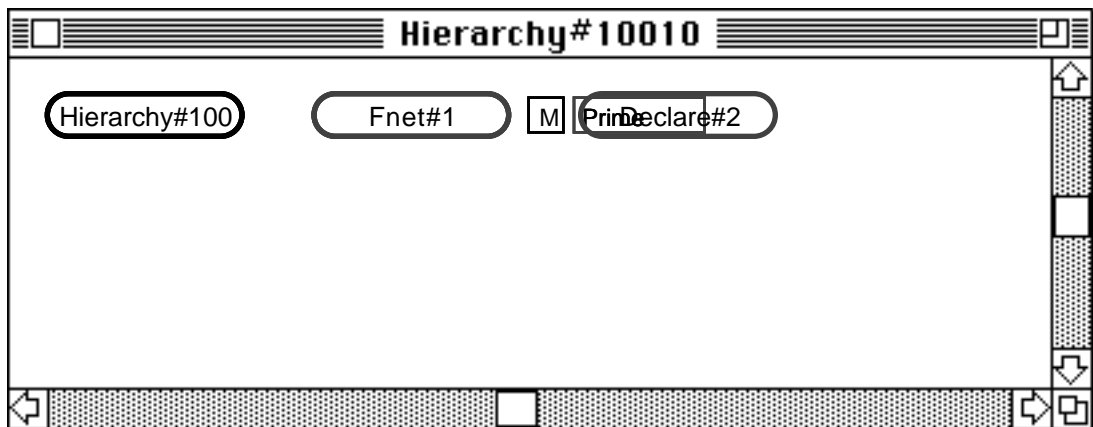
The name of the page is now Declare#2.

### Improving the Hierarchy Page

After you create a new page, it is a good idea to check the hierarchy page to be sure that it still has a good appearance.

- Choose **Open Page** from the **Page** menu.

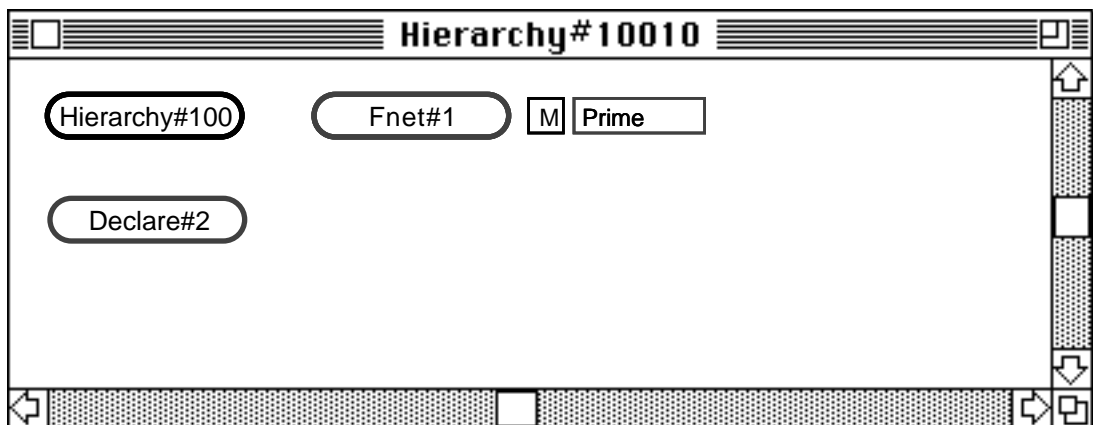
The hierarchy page appears.



The page node for Global#2 has been put in a default position, which unfortunately overlays the page mode region for Fnet#1.

- Use the mouse to reposition Global#2's page node to be underneath the hierarchy page node.

The page should look like this:

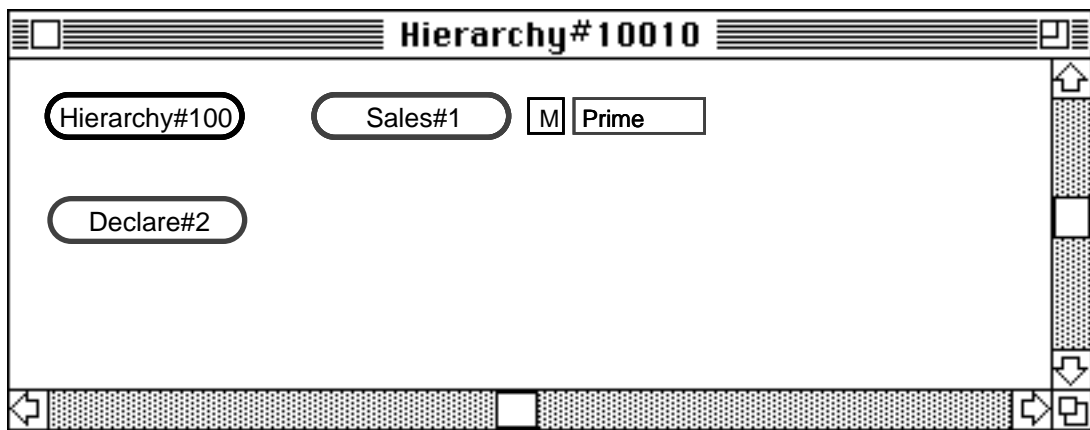


### Renaming a Page From the Hierarchy Page

The page name Fnet#1 is obsolete now, because you have extended FirstNet to become SalesNet. The page should be named Sales#1.

You don't have to make a page the current page in order to rename it. You can rename it, and make many other changes to its status, directly from the hierarchy page.

- Select the page node for Fnet#1.
- Choose **Page Attributes** from the **Set** menu.
- Rename the page Sales#1:



### Moving the Global Declaration Node

Now let's move the global declaration node to the new page Declare#2.

- Double-click on the page node for Sales#1.

Sales#1 becomes the current page.

- Select the global declaration node.
- Execute **Cut** (via the **File** menu or a keystroke shortcut).
- Choose **Open Page** from the **Page** menu.

The hierarchy page appears.

- Double-click on the page node for Declare#2.

Declare#2 becomes the current page.

- Execute **Paste** (via the **File** menu or a keystroke shortcut).

The global declaration node is pasted onto Declare#2.

- Position the node in the upper right corner of the window.

Note that neither putting the global declaration node on its own page nor naming that page Declare has any functional significance. The global declaration node may appear on any page, and that page may have any name.

## Saving the Net

NewSalesNet in its current form will be the basis of all the nets we build in the following chapters.

- Save the net. It is OK to overwrite the existing version.

We won't be using NewSalesNet in the next chapter, so:

- Close NewSalesNet.



# Chapter 11

## Concurrency and Choice

All of the nets we have looked at so far executed sequentially: only one thing happened at a time. Such execution does not require any particular sophistication or complexity on the part of the CPN simulator. It does one thing, and then the next, and then the next, as long as anything remains to be done.

Sequential operation is not typical of real systems. Systems that perform many operations and/or deal with many entities usually do more than one thing at a time. Activities that happen at the same time are called *concurrent activities*. A system that contains such activities is called a *concurrent system*. The phenomenon of concurrent activities is called *concurrency*.

### Concurrency Problems

Concurrency can create problems that do not arise in sequential systems. These problems typically involve competition for resources. In a system in which there is only one active agency, there can be no such competition, because there is nothing for the agency to compete with. But where there are concurrent activities that need the same resources, and there are not enough resources to insure that every activity will always have all that it needs, the activities are forced to compete for what resources there are.

There are two possible ways to deal with competition for resources. One is to implement a determinate algorithm for resource allocation. This solution is simple, but it is not always possible or desirable. Where it is not, the only way to resolve the competition is to randomly allocate resources to some contenders, and require others to wait. The subsequent course of events in the net often depends on which activities prevailed and which had to wait.

Activities that vie for resources in the absence of a determinate allocation mechanism are said to be *in conflict*. The act of adjudicating such a conflict is called *choice*. Since the choice is made at random, and may affect the subsequent course of execution in unpredictable ways, the result of choice is *indeterminacy*.



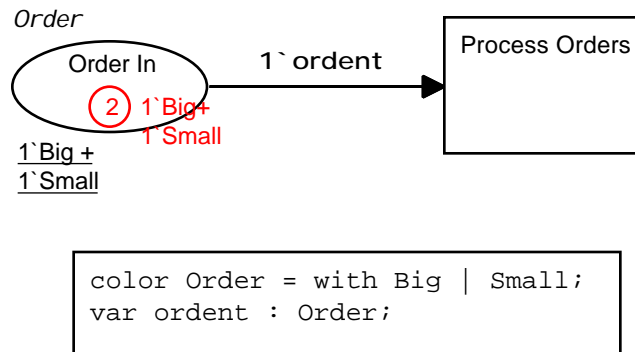
Due to the possibility of conflict and the resulting need for choice, it is impossible for the simulator to get by using only the simple methods we have ascribed to it so far. It must be able to handle conflicts and make choices that can be quite complex, and it must be able to do so efficiently, or the execution of a net that requires many choices will be too slow. This chapter describes the capabilities that the simulator uses to adjudicate conflicts by making choices.

## Representing Concurrency

Before we see how the simulator handles conflict and choice, we need to see how concurrency and conflict are actually represented in a net, and how they affect its execution. First let's look at the simplest form of concurrency.

### Multiple Enabling Bindings

In Chapter 7, all of the examples based on `FirstNet` shared a characteristic that was not pointed out: in each example, there was at most one binding for which the transition `Process Orders` was enabled. But this need not be the case. For example:



Here there are two orders waiting to be processed, a `Big` order and a `Small` order. Consequently there are two bindings of `ordent` for which `Process Orders` is enabled: `ordent = Big` and `ordent = Small`.

How should the simulator handle this situation? The two possibilities are:

1. Process one of the orders, and then process the other.
2. Process both orders at the same time.

The first is straightforward enough. It would entail `Process Orders` firing with one binding, and later firing with the other.

The second option may seem a bit strange: how can the same activity occur in two different ways at the same time? How can one thing do several things simultaneously?

### Concurrent Transition Firing

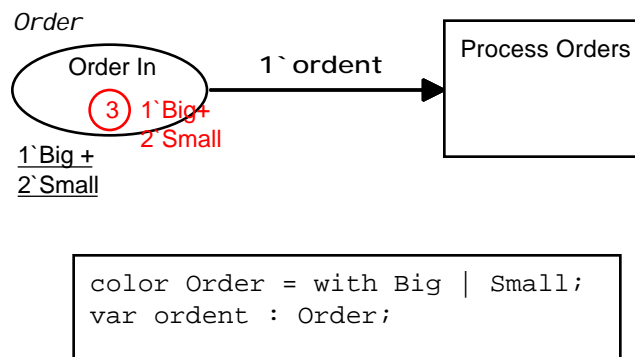
Nothing in the net specifies that `Process Orders` can do more than one thing at a time. But neither does anything specify that it cannot! CP nets make no assumption of sequential behavior: they allow as many things as possible to happen at the same time. Since concurrency abounds in real systems, this property is extremely convenient in a modeling paradigm.

In this case there are two jobs ready to process, represented by the two tokens that produce the two enabling bindings, so `Process Orders` will process them concurrently. If we don't want them processed concurrently, we must explicitly specify something that prevents it, as described later in this chapter.

It is tempting to see a transition like `Process Orders` as an active principle that maps inputs to outputs in a sequential way, but this image is wrong. A transition is just a representation of a way in which the state of a net can change: it has no life of its own. The mapping of inputs to outputs is done by the simulator, not by the transition. Nothing prevents the simulator from carrying out more than one such mapping at a time, and thereby implementing concurrency.

### Identical Enabling Bindings

Nothing requires that enabling bindings be unique. For example:

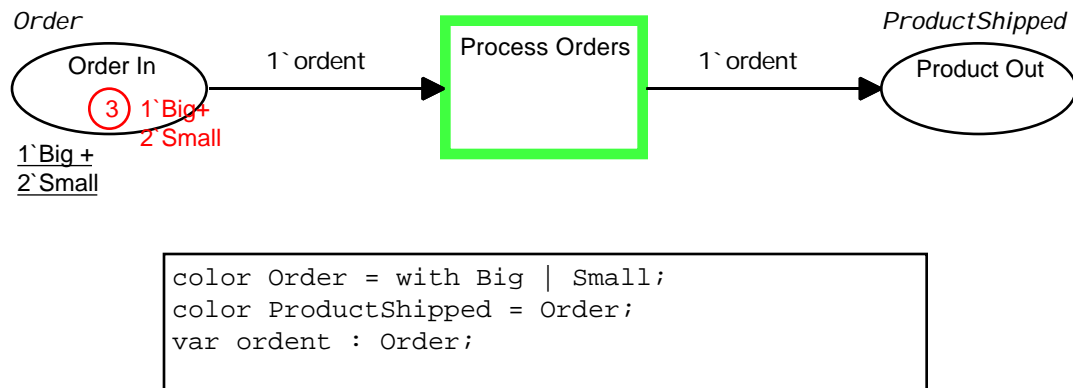


Here there are three enabling bindings, not two. The fact that two of them are identical does not mean they are not distinct entities. If we allowed only one binding at a time with a given value, this would be equivalent to ruling out the possibility of concurrently processing identical entities in the same way. This would obviously be unacceptable as a general restriction in a modeling paradigm.

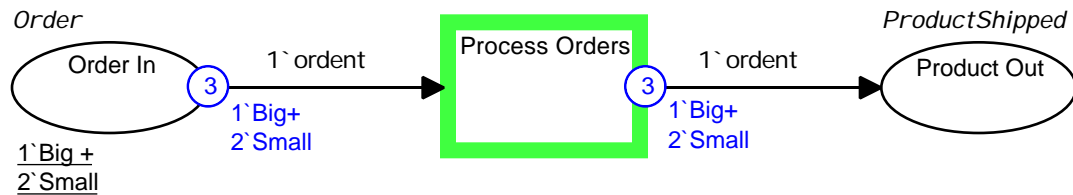
## Concurrent CP Net Execution

Let's take a look at a concurrent execution of FirstNet. One possible sequence is:

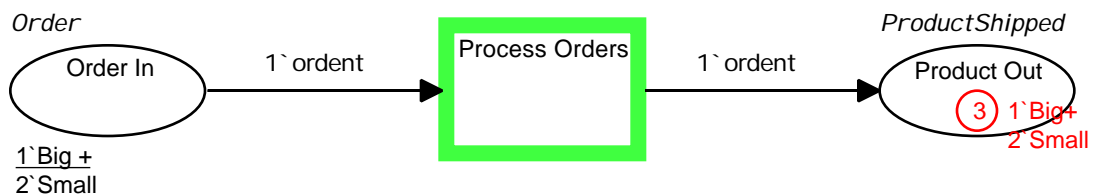
### Initial State of the Net



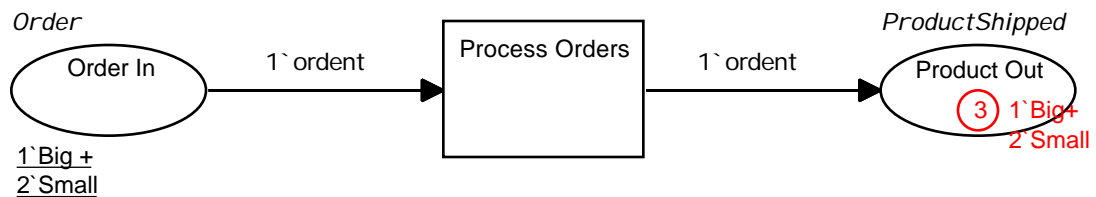
### Breakpoint 1: Beginning of Substep



### Breakpoint 2: End of Substep



### Execution Is Complete



### Analysis of the Execution

The only difference between this execution sequence and any previous execution of FirstNet is that three `Order` tokens were processed in the same step, while previously only one `Order` token was processed, after which there was nothing left to do. This merely reflects the fact that this time there are three enabling bindings, one for each of three `Order` tokens, while previously there was only one enabling binding.

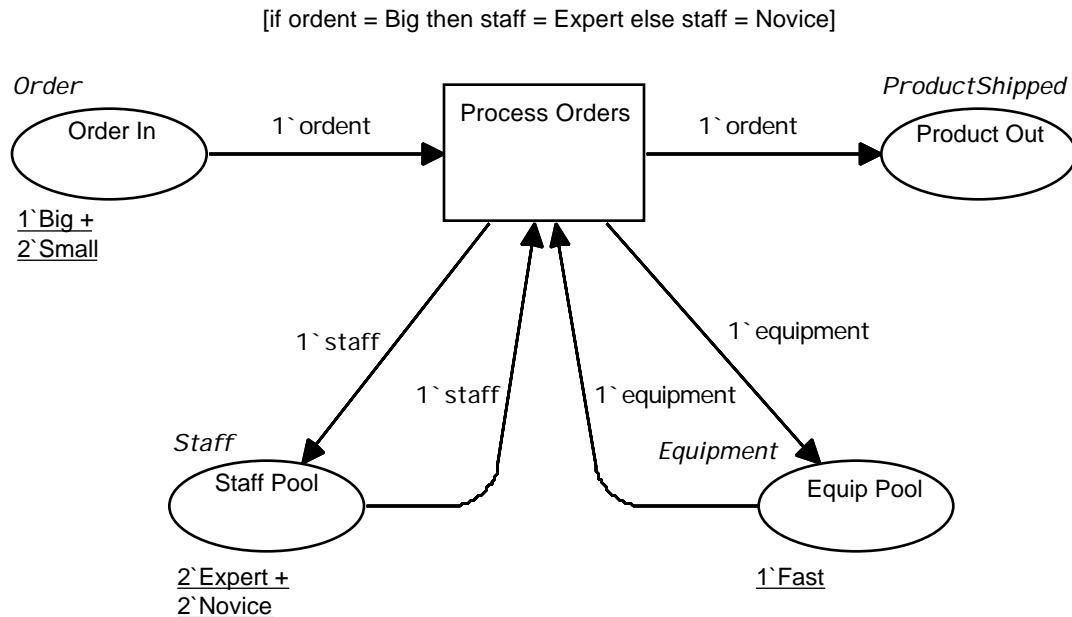
Note that nothing special was done to `Process Orders` to make it process one order or three: it just processes the orders that give rise to enabling bindings. If the result is concurrent processing, then it is. In a CP net, no special effort is required to specify either concurrent or sequential behavior: one just creates a net that has the desired structure, and its behavior, whether concurrent or sequential, follows as a matter of course.

### Representing Conflict

FirstNet places no limitation on the number of orders that can be processed concurrently. There could be a thousand or a million orders, and they would all be processed at once. Obviously this is unrealistic: in a real system, there is always some limit to how much can be done at one time.

Nothing requires that a CP net be realistic, or protects automatically against the consequences of unrealistic modeling. The problem here is that FirstNet is too simple. It contains no representation of anything that would limit concurrent processing of orders, so there is no limit. If we want to limit concurrency in a model, we must explicitly specify some limiting factor. A CP net is exactly what we make it be, and nothing more.

SalesNet shows how concurrency can be limited:



SalesNet specifies that a *Big* order can be processed only when there is an *Expert* staff member and one piece of equipment available, while a *Small* order can be processed only when there is a *Novice* staff member and one piece of equipment available.

There is no problem with staff availability: there are enough staff members to process all three orders concurrently. But there is only one piece of equipment. Since each order must have a piece of equipment in order to be processed, and there is only one piece, the three orders cannot be processed concurrently. One must use it and then replace it, after which another can use it, and so on.

The fact that the three orders cannot use the piece of equipment simultaneously means that they must compete to obtain it. That is, they are in conflict for the piece of equipment, and this conflict prevents them from being processed concurrently. *It is conflict that limits concurrency.*

If there were two pieces of equipment, concurrency would be partially limited. Two orders could be processed at a time, but there are three; the third would have to wait until one of the two has returned its equipment to *Equip Pool*. If there were three or more pieces of equipment, there would be no conflict in the net as shown, but increasing the number of orders to be processed would cause renewed conflict for equipment and possibly for staff as well.

### Conflicts and Bindings

It may seem strange to describe “orders” as being in conflict. This is really just a shorthand for saying that the activities of processing the orders are in conflict.

In SalesNet the three activities of processing the three orders are represented by the transition `Process Orders` in conjunction with three enabling bindings. The activities are in conflict because the transition can fire with only one of these bindings at a time.

The reason it can fire with only one binding at a time is that any one firing will remove the equipment token from `Equip Pool`, so that it is not there to be removed by some other firing. Consequently there can be no other firing until the equipment has been returned.

In other words, though there are three enabling bindings, only one of them at a time can actually be used. As soon as the transition fires with any one of them, there is no equipment token for use by any other. The other enabling bindings have in fact disappeared, since any enabling binding for `Process Orders` must include a piece of equipment and there is no longer any such piece.

When the transition has finished firing, restoring the equipment, two enabling bindings will remain, only one of which can actually fire; and so on, until no enabling bindings remain.

Don't be surprised if this is not all perfectly clear. You will soon be able to use the simulator to explore concurrency and conflict, and the experiments you can perform will reveal that there is actually considerably less here than meets the eye.

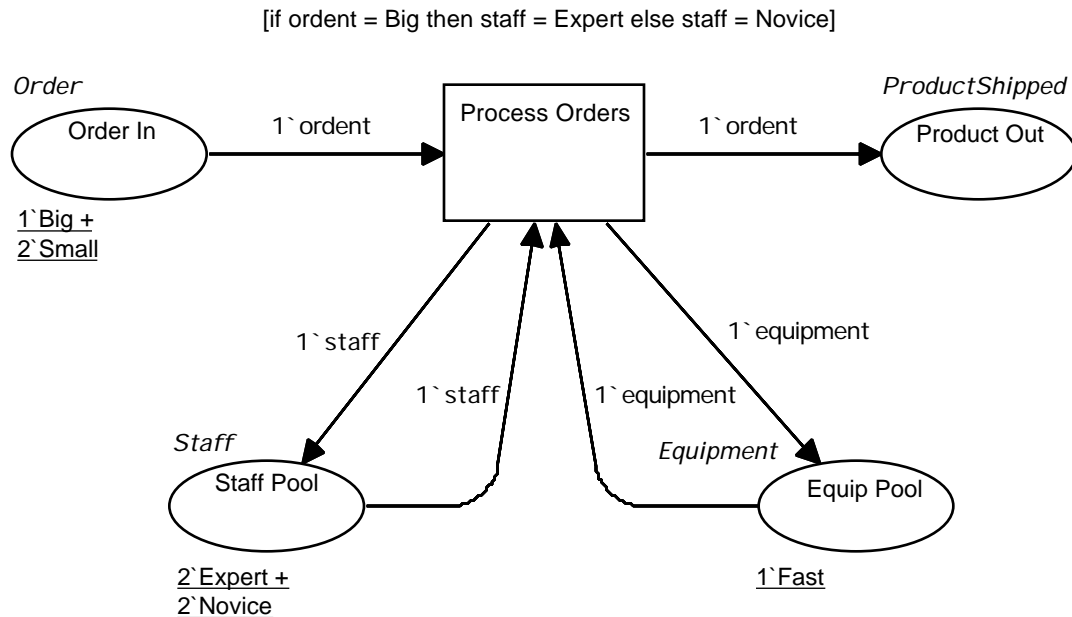
## Concurrent Execution of SalesNet

Let's go into the simulator and watch SalesNet execute concurrently. We'll look first at the case where there are enough resources to prevent conflict, then cut the number down so that conflict arises.

We could start with NewSalesNet, the net you constructed in the previous chapter, but that would not be ideal. The reason is that in order to approach concurrent execution systematically, we need some parameters that control simulator behavior to be set in a particular way. These parameters are described in Chapter 15; it would not work for you to try to cope with them yet. So we will begin with a net in which they are appropriately set already.

- Open the diagram SalesNetDemo in the NewTTDiagrams directory.

The diagram opens. Except for minor variations, it is the same as NewSalesNet:



### Loading ML Configuration Information

The diagram you have just opened was not created on your system, so it does not contain the information necessary to allow Design/CPN to communicate with the ML process. Therefore you must load that information into the diagram.

- Choose **ML Configuration Options** from the **Set** menu.

A dialog appears.

- Click **Load**.
- Click **OK**.

The necessary information about the ML process is copied from the system defaults to SalesNetDemo's diagram defaults. Design/CPN can now access the ML process, permitting it to syntax check and then execute the net.

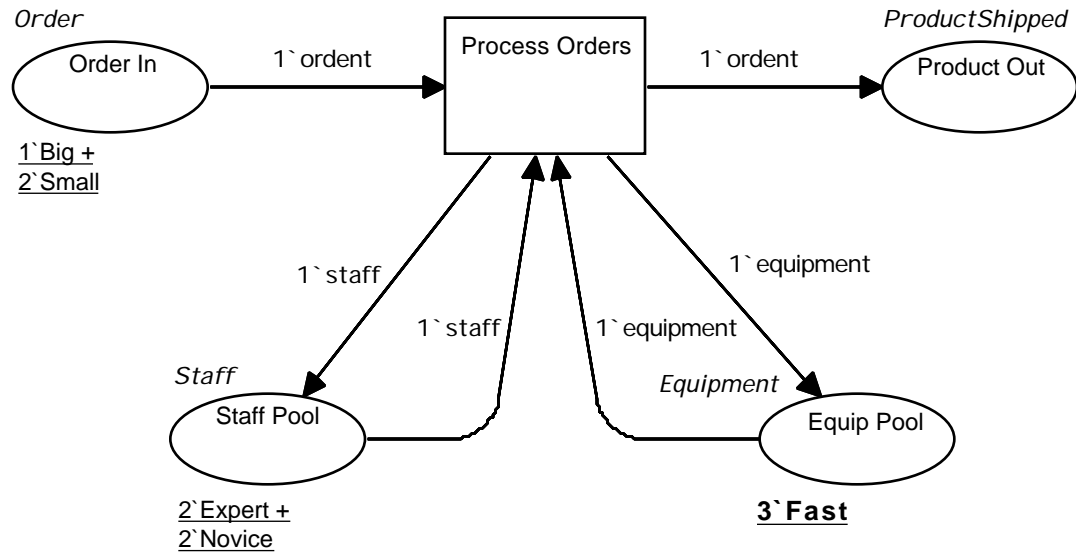
### Adding More Equipment

In order for SalesNet to execute concurrently, there must be enough equipment for more than one order to be processed at a time. In order for it to execute without conflict, there must be enough for all orders to be processed at the same time.

- Enter text mode and change the initial marking region of Equip Pool to 3`Fast.

The net should look like this (bolding excepted):

[if ordent = Big then staff = Expert else staff = Novice]



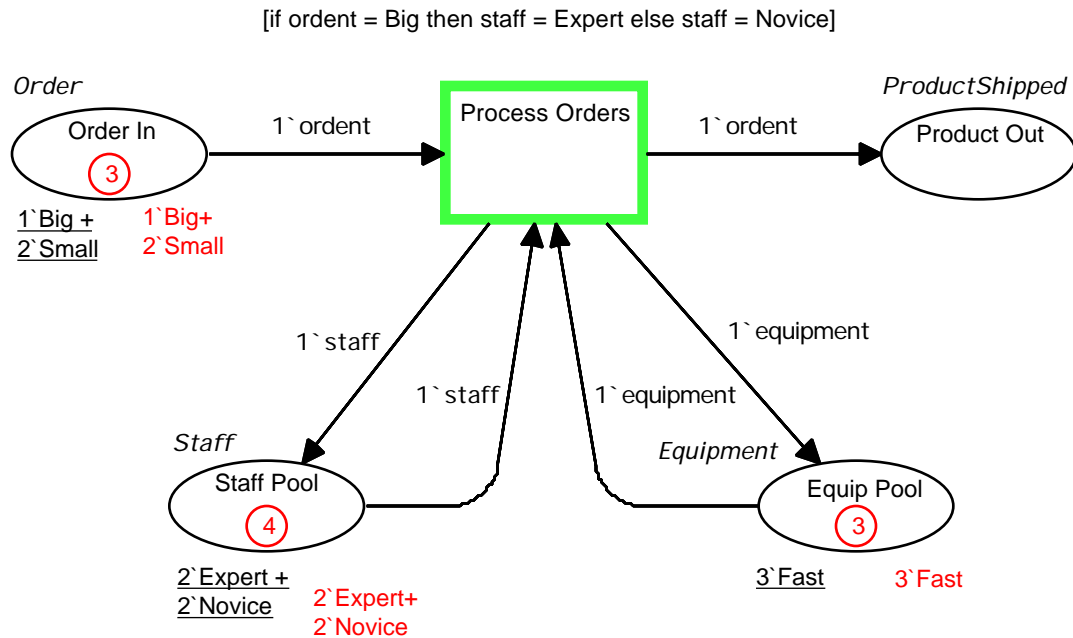
### Executing SalesNet

Now let's see how SalesNet executes. SalesNetDemo already has a prime page, so you don't need to designate one.

- Enter the simulator.

Breakpoints and the display of input and output tokens are already set, and the various simulation regions have already been positioned for good appearance:

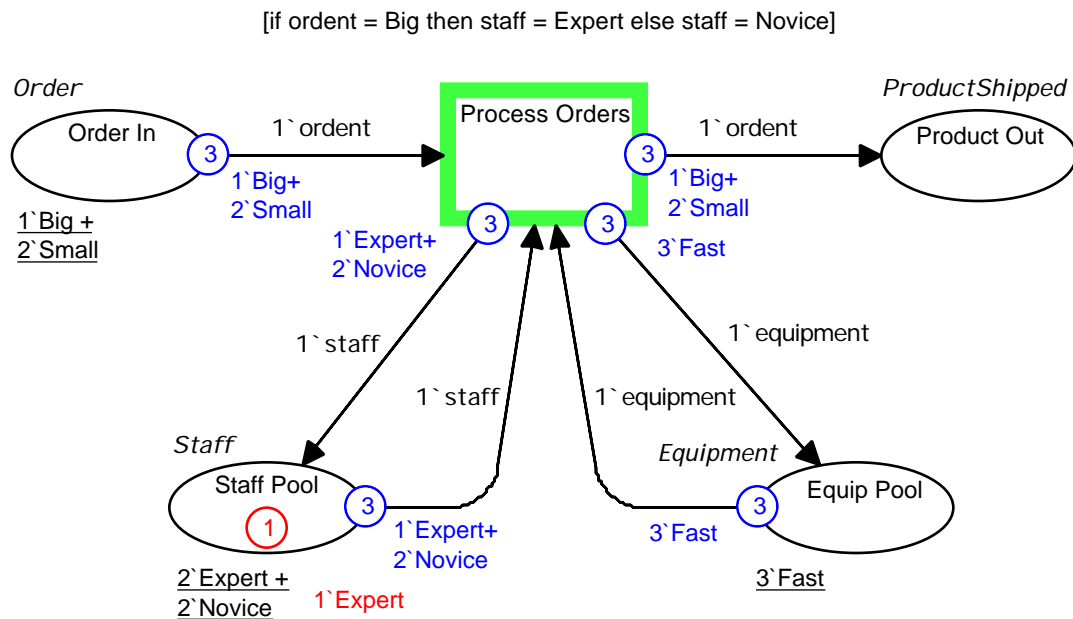




- Start simulation.
- Choose **Continue** until execution is complete.

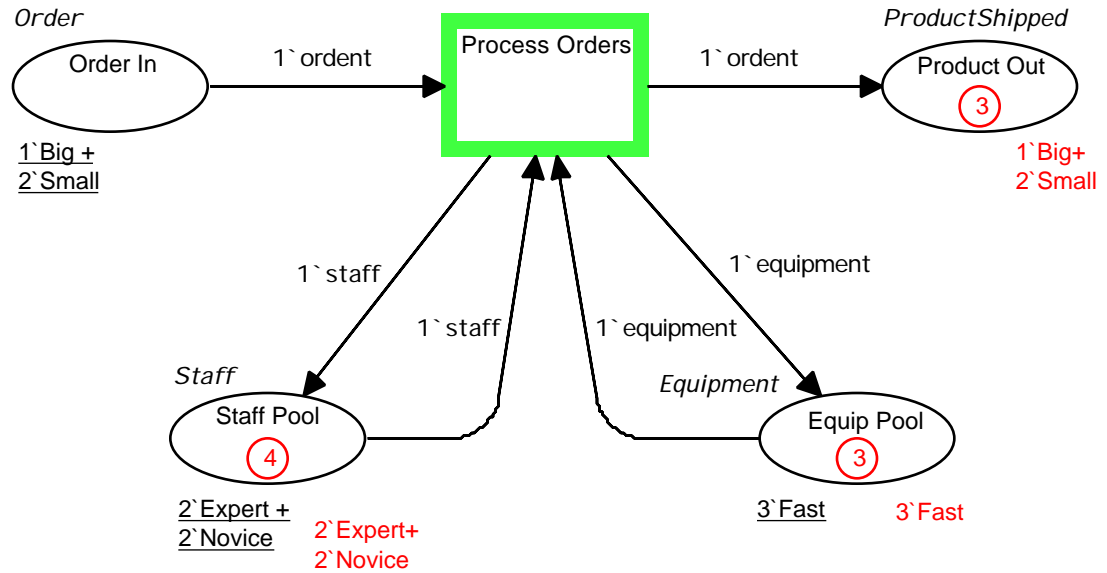
The net should go through the following states:

## Breakpoint 1: Beginning of Substep



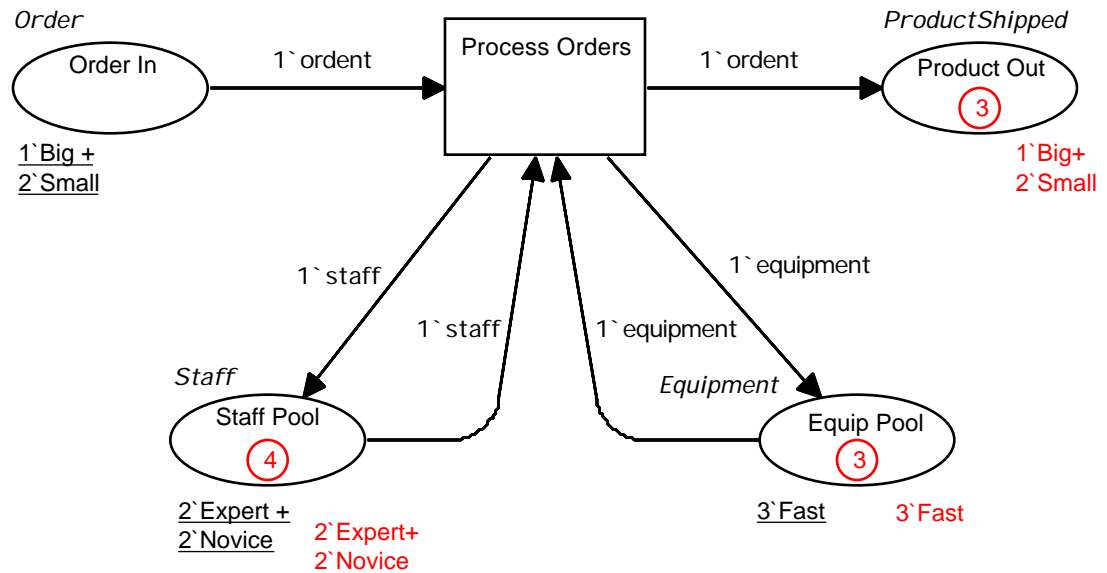
## Breakpoint 2: End of Substep

[if ordent = Big then staff = Expert else staff = Novice]



## Execution Is Complete

[if ordent = Big then staff = Expert else staff = Novice]



### Analysis of the Execution

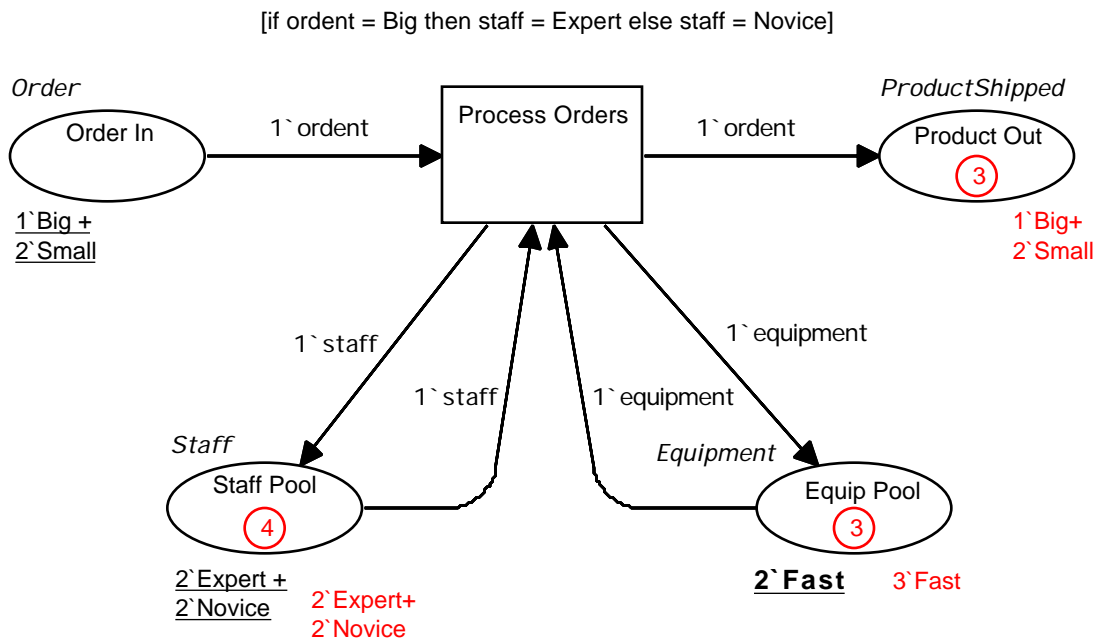
You probably didn't see much that was new in this execution. The reason is that there was no conflict: the three orders were processed concurrently in a single step, and that was it. The next execution of SalesNet will provide only two pieces of equipment, so that concurrency will be partially but not completely limited by conflict.

### Changing a Net in the Simulator

Let's liven things up in SalesNet by cutting back to two pieces of equipment and watching the orders compete for them. To accomplish this, you need only change the initial marking of Equip Pool. You don't have to return to the editor to make minor changes such as this. You make them directly in the simulator.

- Select the initial marking region of Equip Pool.
- Enter text mode.
- Edit the region to be 2`Fast.
- Leave text mode.

The net should now look as follows:



- Pull down the **Sim** menu and examine it.

Note that most of the commands are disabled, but that **Reswitch** is available. After you make a change in the simulator, the parts of the net that you have changed must be syntax checked again, and new executable code must be generated for them. This process is called *reswitching*.

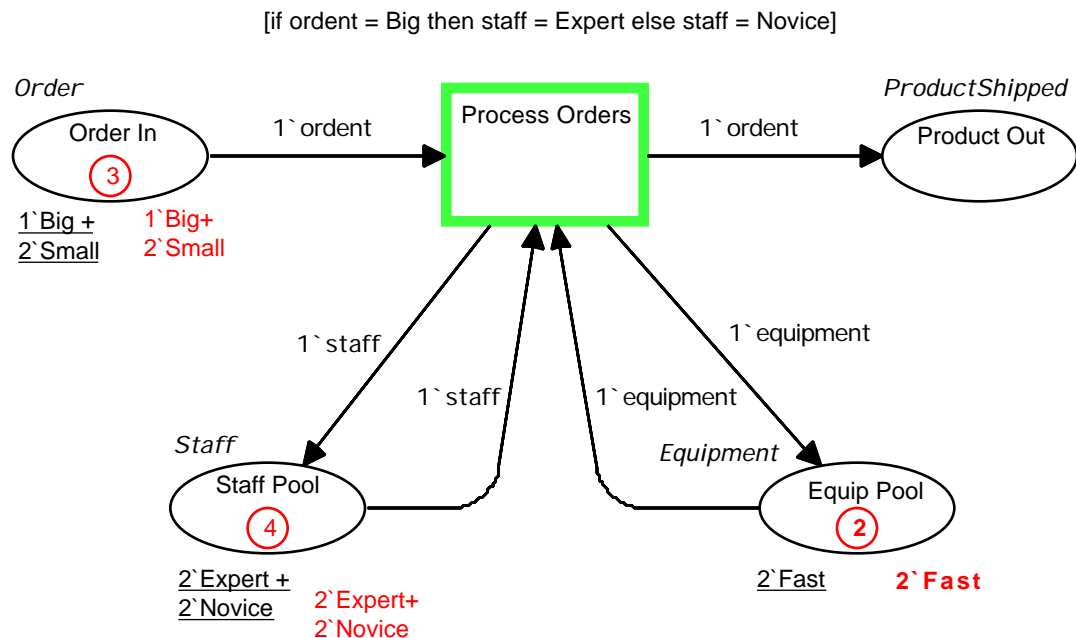
- Choose **Reswitch** from the **Sim** menu.

Keep an eye on the status bar: you will see it briefly display the same messages that it does when you are entering the simulator from the editor. The displays are brief because only the parts of the net that are affected by the change need to be rechecked.

Now look at the state of the net overall. The marking of `Equip Pool` has not changed because you changed its initial marking region. The reason is that the creation of initial tokens as specified by an initial marking region happens only when a net's initial state is established on entry to the simulator or via the **Initial State** command.

- Choose **Initial State** from the **Sim** menu.

When the new initial state has been established, the status bar displays Finished Initializing State. There are now two `Fast` tokens in `Equip Pool`:



You can now run the net just as if `Equip Pool`'s initial marking had always been 2`Fast. All effects of previous executions were erased when you executed **Initial State**.

### The Simulator's Execution Algorithm

In Chapter 8, when you first watched a CP net execute, you were asked to ignore anything the simulator did that did not make obvious sense, and you were promised that all would be made clear in due time. The time has come to keep that promise.

When a net is given to the simulator for execution, the simulator does the following:

1. Evaluate any initial marking regions and put tokens into places as the regions specify.
2. Scan the net and make a list of all transitions for which there exists at least one enabling binding. This is the *enabled list*.
3. Do the following:
  - 3A. Scan the enabled list and construct a list of transitions and enabling bindings such that there is no conflict among the bindings. Such a list is called an *occurrence set*. Its elements are called *binding elements*.
  - 3B. For every binding element in the occurrence set, fire the transition it indicates with the binding it indicates. Since the bindings are nonconflicting, all these firings can happen concurrently.
  - 3C. Update the enabled list by rechecking the enablement of all transitions whose input places were changed by the firings just completed. (Transitions whose input places have not changed do not need to be rechecked.)
4. If the enabled list is not empty after the update in 3C, repeat from 3A.
5. If the enabled list is empty, terminate execution. (You can also terminate execution at any time by pressing ESC.)

Each iteration of 3A, 3B, and 3C is called a *Step*. The firing of an individual binding element is called a *Substep*.

### Executing SalesNet With Conflict

Let's execute SalesNet now, and look at everything the simulator's execution algorithm does to carry out the execution.

#### 1: Establish Initial Markings

This has already been done by the **Initial State** command. The resulting markings are shown in the above figure.

#### 2: Put All Enabled Transitions on the Enabled List

This has already been done by the **Initial State** command. There was at least one enabling binding for `Process Orders`, so it is enabled and has been put on the enabled list. The simulator indicates the transition's enabled status by highlighting it, as shown in the above figure.

#### 3A: Scan the Enabled List and Construct an Occurrence Set

- Choose **Interactive Run** from the **Sim** menu

The status bar displays Constructing Occurrence Set. The simulator now examines all transitions on the enabled list, and constructs a list of transitions with enabling bindings such that there is no conflict among the bindings. This list is the occurrence set for the step that is currently underway.

When several transitions exist and have mutual conflicts, adjudicating the conflicts can become quite complex, but the situation is fundamentally no different when only one transition is involved: the same sorts of decisions have to be made, and the same rules are followed in making them. In the current case there is only one transition, `Process Orders`, and there are three enabling bindings for it. There is one binding:

```
ordent: Big
staff:  Expert
equip:  Fast
```

And there are two identical bindings:

```
ordent: Small
staff:  Novice
equip:  Fast
```

Any two of these bindings can fire concurrently, because there are two (and only two) `Fast` tokens available. There is no way to prefer one binding over another, so the simulator chooses any two at

random. This is the choice that must be made whenever conflict exists. Since it is made at random, the result is indeterminacy.

In this case the indeterminacy is of little import, due to the simplicity of the net, but a more complex net could behave very differently in future steps depending on what orders are processed now and what orders are forced to wait.

### 3B: Execute the Elements in the Occurrence Set

Chapter 8 described the algorithm for firing a transition as follows:

1. Rebind any CPN variables as indicated by the enabling binding.
2. Evaluate each input arc inscription. The result is a multiset of tokens called input tokens.
3. Evaluate each output arc inscription. The result is a multiset of tokens called output tokens.
4. Subtract each multiset of input tokens from each input place.
5. Add each multiset of output tokens to each output place.

It was convenient but imprecise to describe this as the algorithm for firing a transition. It is more correct to call it the algorithm for executing a binding element. When an occurrence set has only one element (as they all did in previous chapters) the distinction is unimportant, but now we can be more precise.

### Executing an Occurrence Set

Since the elements in an occurrence set are not in conflict, they can be executed in any order. On a multiprocessing computer they could all be executed simultaneously by different processors. This would be faster than executing them in some order, but neither the presence of some particular order nor the absence of any order makes any difference, because the net ends up in the same state in any case.

This lack of dependence on, or even need for, an ordering for binding element execution allows the simulator to intersperse such executions. This makes possible the breakpoints **Beginning of Substep** and **End of Substep**.

The algorithm for executing an occurrence set is:

1. For each element in the set:
  - 1A. Rebind any CPN variables as indicated by the enabling binding.
  - 1B. Evaluate each input arc inscription.
  - 1C. Evaluate each output arc inscription.
2. If Show Input Tokens is set, display the input tokens.
3. If Show Output Tokens is set, display the output tokens.
4. If the breakpoint **Beginning of Substep** is set, pause execution.
5. When execution continues:
  - 5A. Subtract each multiset of input tokens from each input place.
  - 5B. Add each multiset of output tokens to each output place.
6. If the breakpoint **End of Substep** is set, pause execution.

Execution of the occurrence set is now complete. When net execution continues, the simulator will recheck enablement, as described below.

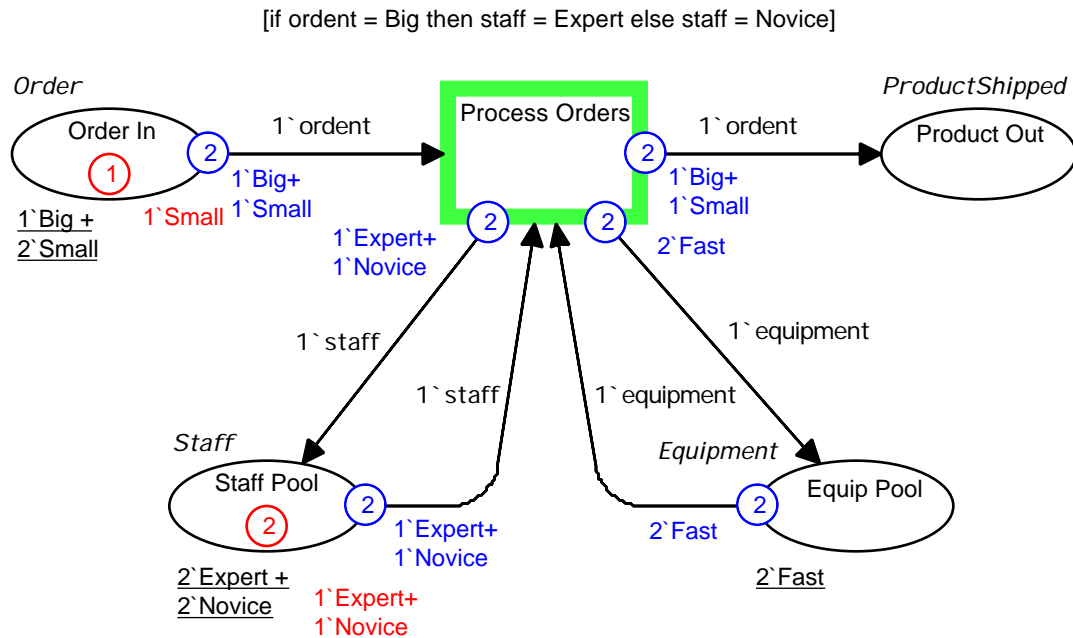
### **SalesNet's Appearance at Breakpoint 1**

When you chose **Interactive Run** above, the simulator constructed an occurrence set and began to execute it. Breakpoint 1 is set, so execution has paused after creating and displaying the input and output tokens.

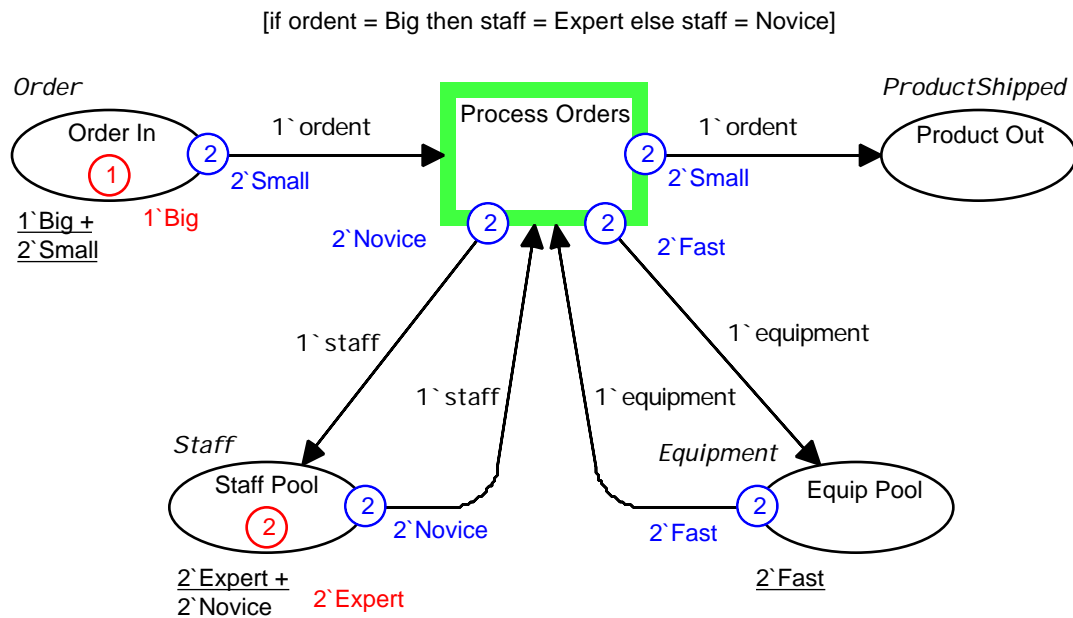
Suppose that the simulator created an occurrence set that contains two binding elements. One specifies firing `Process Orders` with `ordent = Big`, `staff = Expert`, `equip = Fast`; the other specifies `ordent = Small`, `staff = Novice`, `equip = Fast`. The net would look like this at Breakpoint 1:



## Design/CPN Tutorial



Your simulator may have constructed an occurrence set in which both elements specified `ordent = Small`, in which case the net you see will look like this:

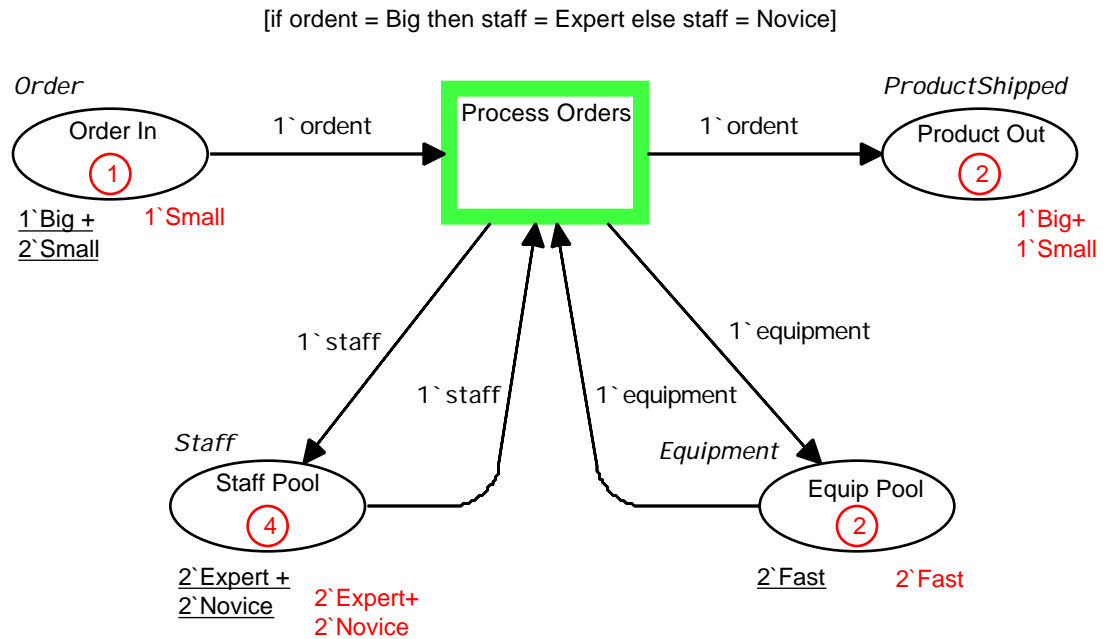


### SalesNet's Appearance at Breakpoint 2

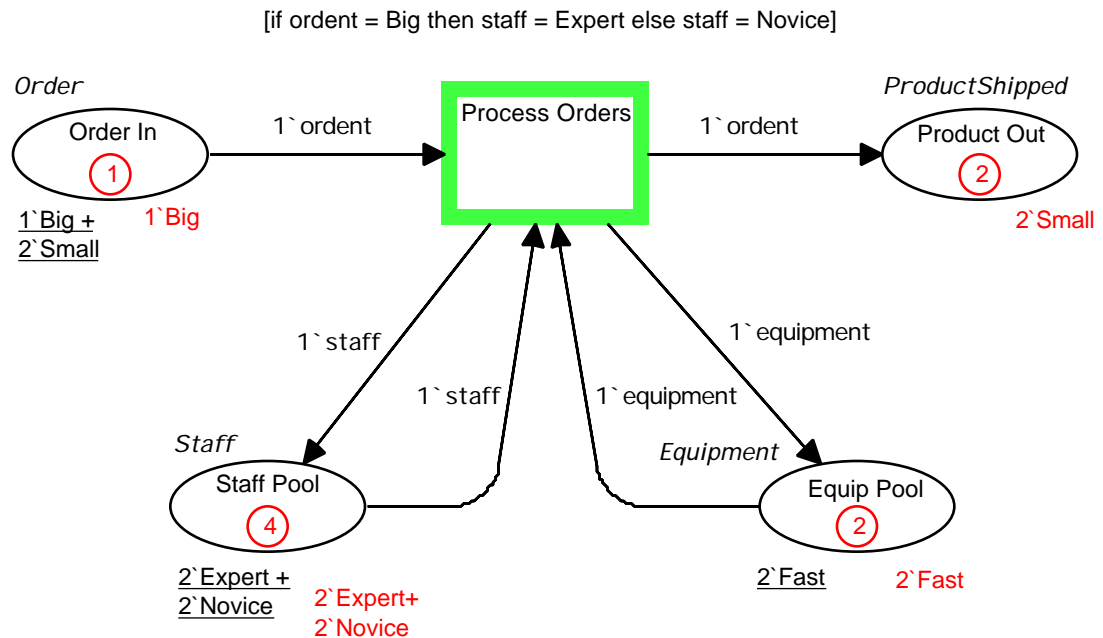
- Choose **Continue** from the **Sim** menu.

## Concurrency and Choice

The simulator proceeds with the execution of the occurrence set: it subtracts the input tokens, and it adds the output tokens. Since Breakpoint 2 is set, it then pauses execution once more. Assuming the first occurrence set described above, the net would look like this at Breakpoint 2:



If your simulator constructed an occurrence set in which both elements specified `ordent = Small`, in which case the net you see will look like this:



Execution of the occurrence set is now complete.

### 3c: Update the Enabled List

- Choose **Continue** from the **Sim** menu.

Transitions whose input places have not changed need not be rechecked for enablement, because their status cannot have changed. `Process Orders` has a changed input place, so it will be rechecked and found to be enabled. It is therefore again highlighted.

With the enabled list has been updated, the step is over. The break-point **Between Steps** is set, so the **Step Finished** dialog appears



### 4: Continue Execution

- Click **Cont**.

Since `Process Orders` is enabled, the simulator executes another step (3A-3C). This step involves no choices, since there is only one order to be processed, but the stages are the same.

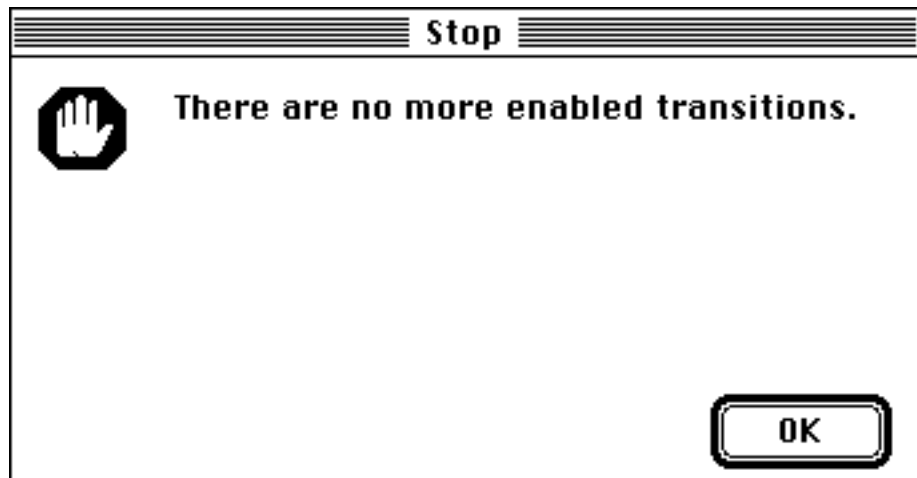
- Continue execution through the two breakpoints

The **Step Finished** dialog reappears.

### 5: Complete Execution

- Click **Cont**.

Now the situation is different. When the simulator rechecked enablement, it found that there is no enabling binding for any transition. It therefore displays a dialog that states this fact:



- Click **OK**.

## Experimenting With Concurrency and Conflict

There is only one way to become comfortable with the material this chapter has presented so far: experiment with it.

- Change one or more initial markings in some way that you think will produce interesting conflicts.
- Leave text mode. (You can't reswitch in text mode.)
- Execute **Reswitch**, **Initial State**, and **Interactive Run**.
- Track execution of the net, noting how it responds to the conflicts you have created.

Your goal should be not only to learn about concurrency and conflict, but to become familiar with the simulator's execution algorithm. The algorithm contains a fair amount of detail, but no great conceptual depth. With experience you will soon come to take it for granted.

- Perform additional experiments, using various initial markings, until you feel comfortable working with the simulator.
- Quit Design/CPN and take a break.

Congratulations. You have just survived the most difficult phase of learning about CP nets.



# Chapter 12

## CPN Hierarchical Decomposition

FirstNet and SalesNet, the nets we have worked with so far, each has all of its graphics on a single page. It would not be feasible to implement a large model in this way. Effective CPN modeling requires the ability to distribute a net across multiple pages, so as to divide it into modules small enough to keep track of. Such a module is called a *submodel*.

Distributing a net across multiple pages requires some mechanism for interconnecting the submodels on the various pages, so that the state of one can influence the state of another. Otherwise we would have several disconnected nets rather than one distributed net.

Design/CPN offers two mechanisms for interconnecting net structure on different pages: substitution transitions and fusion places. A *substitution transition* is a transition that stands for a whole page of net structure. A *fusion place* is a place that has been equated with one or more other places, so that the fused places act as a single place with a single marking.

Substitution transitions and fusion places together provide a very general capability for organizing a net into submodels. This capability is called *CPN Hierarchy*. A complete discussion of CPN hierarchy appears in Appendix A.

This chapter shows you how to use a substitution transition to create a one-level hierarchical decomposition. This use demonstrates only a few of the capabilities that substitution transitions provide. Appendix A demonstrates the rest.

### Definition of Hierarchical Decomposition

Consider SalesNet, the net you created in Chapter 10. SalesNet represents a very high-level view of the system it models. Such a view can be useful, of course; but it would also be useful to have more detailed information about how orders are processed, staff and equipment used, and products shipped. Ideally we would like to

represent this additional information without having to lose the simplicity of the high-level overview that SalesNet currently provides.

In order to add detail to a model without losing overview, a transition may have associated with it a separate page of CP net structure called a *subpage*. This page contains a more detailed view of the activity that the transition represents. Such a transition is called a *substitution transition*. The details on the subpage are called the *decomposition* of the transition. The page that holds the transition is called the *superpage*.

Transitions on a subpage may in turn have associated subpages, and so on. Since this method of representing details results in a hierarchy of subpages that contain decompositions, it is called *hierarchical decomposition*.

## Top-Down and Bottom-Up Development

There are two different ways to specify a decomposition for a transition:

1. Create a page containing a submodel, then link the submodel to the transition. The page then becomes a subpage, and the transition becomes a substitution transition on a superpage. This is *bottom-up development*.
2. Start with a transition, have Design/CPN create a subpage for it, then edit the subpage to create the submodel. This is *top-down development*.

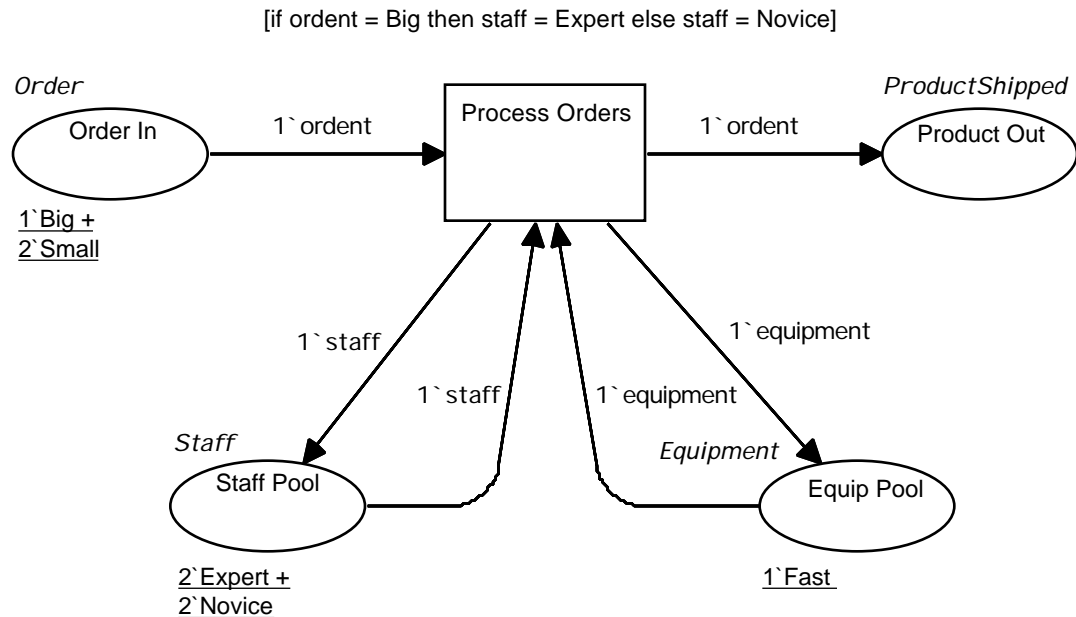
Neither of these methods is intrinsically preferable. The choice of which to use is largely a matter of how one prefers to work. This chapter gives an example of top-down development. Bottom-up development, and much more about top-down development, are discussed in Appendix A.

## Creating a Hierarchical Decomposition

Let's extend SalesNet by specifying a decomposition for the transition `Process Orders`.

- Open NewSalesNet, the diagram you created in Chapter 10 and saved in the NewTTDiagrams directory.

The net looks about like this:



## Designating the Transition to Decompose

- Select the transition `Process Orders`.

## Initiating Subpage Creation

- Choose **Move to Subpage** from the **CPN** menu.

You may wonder why the command is **Move to Subpage** rather than something like **Create Decomposition Page**. The reason is that the using a substitution transition to create a decomposition page for a single existing transition is just a special case of a much more general mechanism. Design/CPN allows you to designate a whole section of existing net structure and move it to a subpage, leaving behind a substitution transition to represent it all. When the section moved is itself just a single transition, as in this case, the effect is to create a decomposition page for that transition.

## Specifying the Substitution Transition's Location

Design/CPN does not assume that you want the substitution transition to appear at the location of the transition you have selected. It therefore enters a specialized editor mode: *substitution transition creation mode*. This mode is similar to ordinary transition creation mode, except that the result is a substitution transition. As with ordinary transition creation, moving the mouse during the creation process reshapes the transition at its current location, while depress-



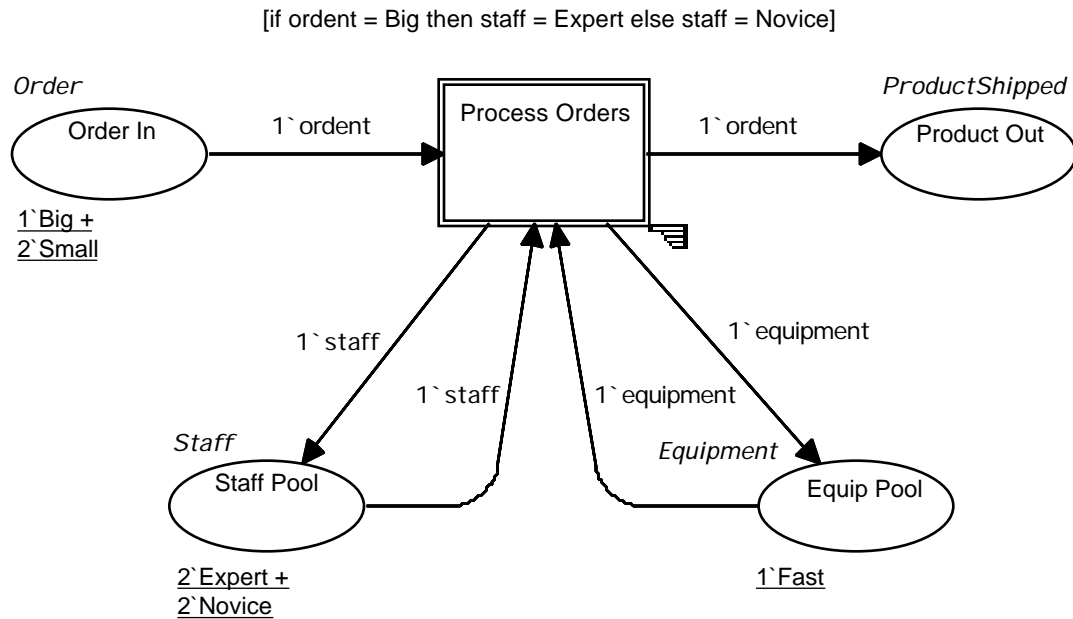
## Design/CPN Tutorial

---

ing SHIFT and moving the mouse moves the transition while preserving its shape.

- Use the mouse with the SHIFT key to create a transition that just surrounds the transition `Process Orders`.

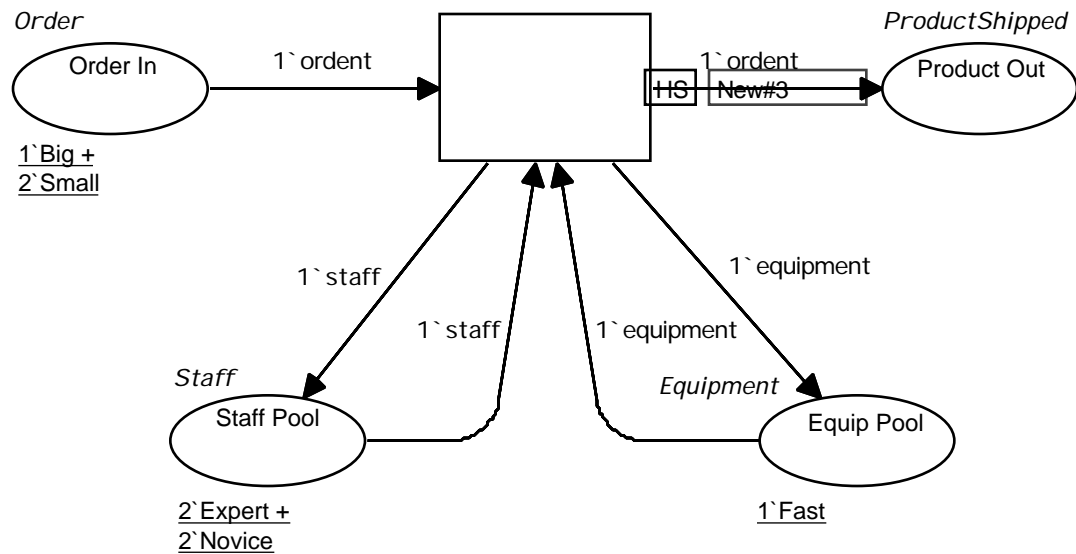
Just before you release the mouse button, `Process Orders` should look like this:



- Release the mouse button.

The adjustment tool disappears. Design/CPN moves `Process Orders`, to a new page that it creates for this purpose. That page is a subpage, and the page you have been working on is now a superpage. The superpage looks like this:

# Hierarchical Decomposition



Process Orders has been replaced by a substitution transition. To indicate this, a region called the *hierarchy key region* has been created. This consists of a box, **HS**. The box is currently partly obscured by the arc to Product Out. The next section shows you how to move this region to a better location.

Next to the hierarchy key region is another region, called the *hierarchy region*, containing the text New#3. This region indicates that the net structure that the substitution transition stands for is on the page New#3. It is a popup region, so you can make it appear and disappear by double-clicking on the key region.

## Naming the Substitution Transition

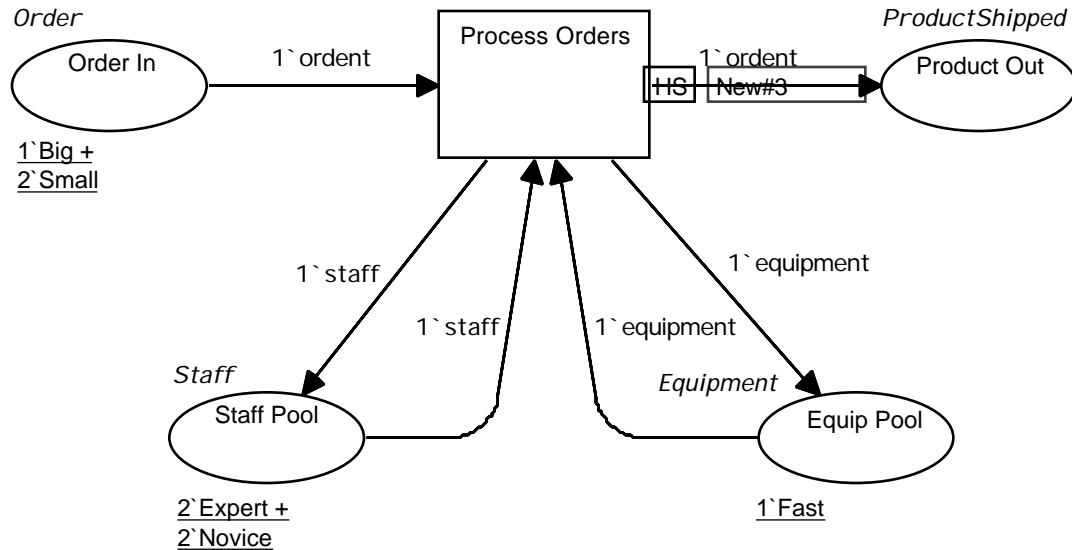
Design/CPN does not assume that you will want a substitution transition that replaces a single transition to have the same name as the replaced transition, so it does not name such a transition automatically.

When you create a substitution transition via **Move to Subpage**, Design/CPN automatically enters text mode. You could name the substitution transition immediately by typing in text, but you would not be able to reposition the text afterwards to improve its appearance, so:

- Leave text mode.

In this case, the substitution transition should have the same name as the original.

- Use **CPN Region** to name the substitution transition `Process Orders`. Position the name region in the top center of the transition:



### Improving the Substitution Transition's Appearance

The substitution transition would look better if the hierarchy regions weren't right on top of the arc. But moving the regions presents a problem: they are so small that it is difficult to grab onto them with the mouse without getting the arc instead. The hierarchy key region is particularly hard to get hold of.

To deal with situations like this, Design/CPN provides commands in the **Makeup** menu that allow you to select and move objects without using the mouse.

- Select the substitution transition.
- Choose **Child Object** from the **Makeup** menu.

The transition has only one child object, the hierarchy key region, so the region becomes selected. (If there were more than one child object, and **Child Object** did not select the needed object, you could use **Next Object** and **Previous Object** to select among the child objects.)

So much for selecting the region. In order to move it:

- Choose **Drag** from the **Makeup** menu.

The editor is now in *drag mode*. The mouse pointer becomes the *drag tool*:

## Hierarchical Decomposition

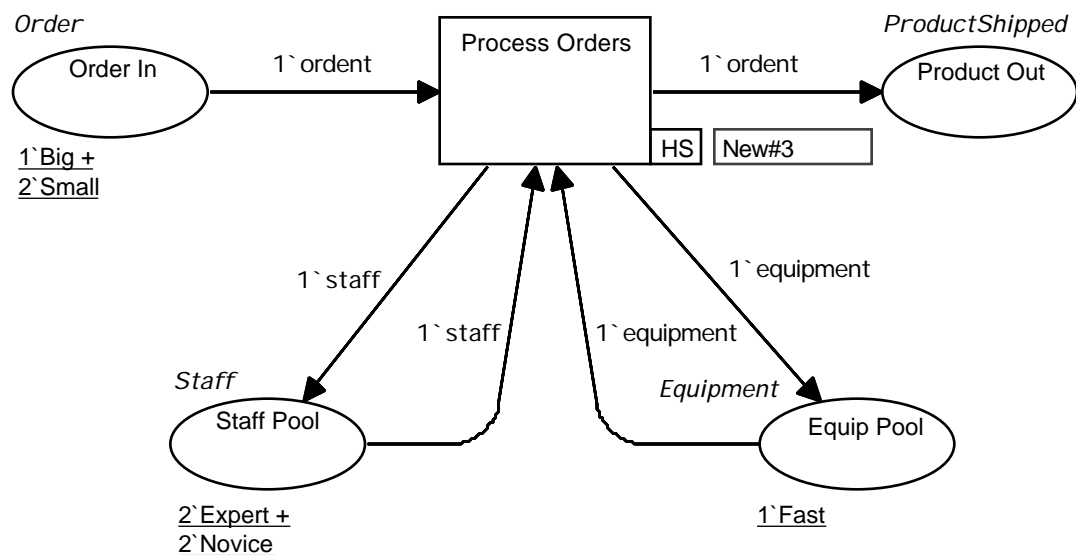


In drag mode, you can move objects without having to position the mouse pointer directly over them. If you depress the mouse button anywhere over the current window, then move the mouse, all selected objects will move along with the mouse.

- Move the drag tool to the general vicinity of the obscured hierarchy key region.
- Depress the mouse button.
- Move the mouse around.

Note how the key region tracks the movement of the mouse. Since the hierarchy region is a region of the key region, it follows the movement also.

- Position the hierarchy key region in the lower right corner of the substitution transition:



To exit drag mode:

- Press ESC.

The editor returns to graphics mode.

### Connecting Superpages to Subpages

The page now looks about as it did before, except that `Process Orders` is no longer an ordinary transition: it is a substitution transition, as indicated by the hierarchy key region HS that is associated with it. The name of the subpage is New#3, as indicated by the hierarchy key region.

The four places `Order In`, `Staff Pool`, `Equip Pool`, and `Product Out` have the same appearance as before, but their status has changed. Each of them is now a *socket*, and is linked with a place on the subpage called a *port*. This linkage is the key to understanding how Design/CPN implements hierarchical decomposition, and substitution transitions generally.

The introduction to this chapter briefly mentioned fusion places. A fusion place is a place that has been equated with one or more other places, so that the fused places act as a single place with a single marking.

Sockets and ports are a type of fusion place. When a place that is a socket has been equated with a place that is a port, the two are a single functional entity. They are not really two places any more: they are one place that has been drawn on two different pages.

This double representation allows us to see the place no matter which page we are looking at, and it provides the interconnecting capability that allows pages to be linked into hierarchical decompositions.

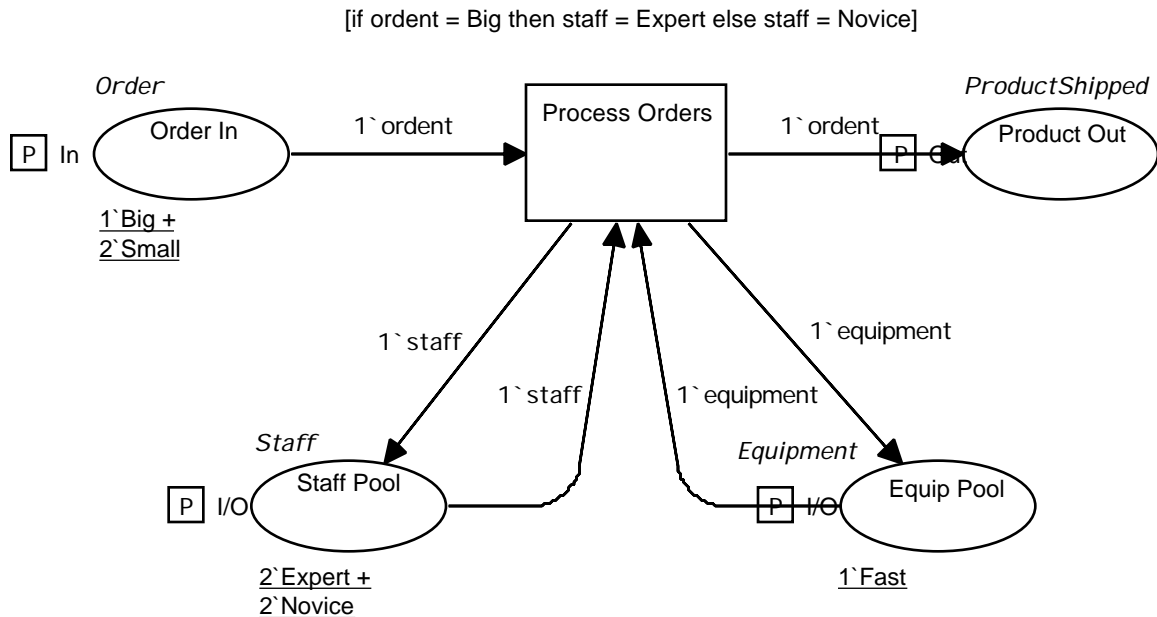
### Structure of the Subpage

Let's take a look at the subpage now.

- Double-click the substitution transition `Process Orders`.

The subpage appears:

# Hierarchical Decomposition



The surprise here is how little surprise there is. The subpage looks almost exactly like the superpage. The only difference is the appearance of the box **P** and an associated string next to each of the four places.

The **P** is called a *port key region*. It indicates that the associated place is a port. The associated string is contained in a region called a *port region*.

The similarity between this subpage and the superpage it was derived from is an artifact of the simplicity of the superpage, and of the way Design/CPN creates a decomposition.

## How Design/CPN Creates a Decomposition

When you use **Move to Subpage** to create a decomposition page for a single transition, Design/CPN does the following:

- Copy the selected transition to a newly created subpage
- Replace the selected transition with a substitution transition.
- Copy every place directly adjacent to that transition to the subpage.
- Copy all arcs between the transition and any of the copied places to the subpage.

- Equate each copied place on the superpage with its copy on the subpage, so that the original becomes a socket, and its copy becomes a port.

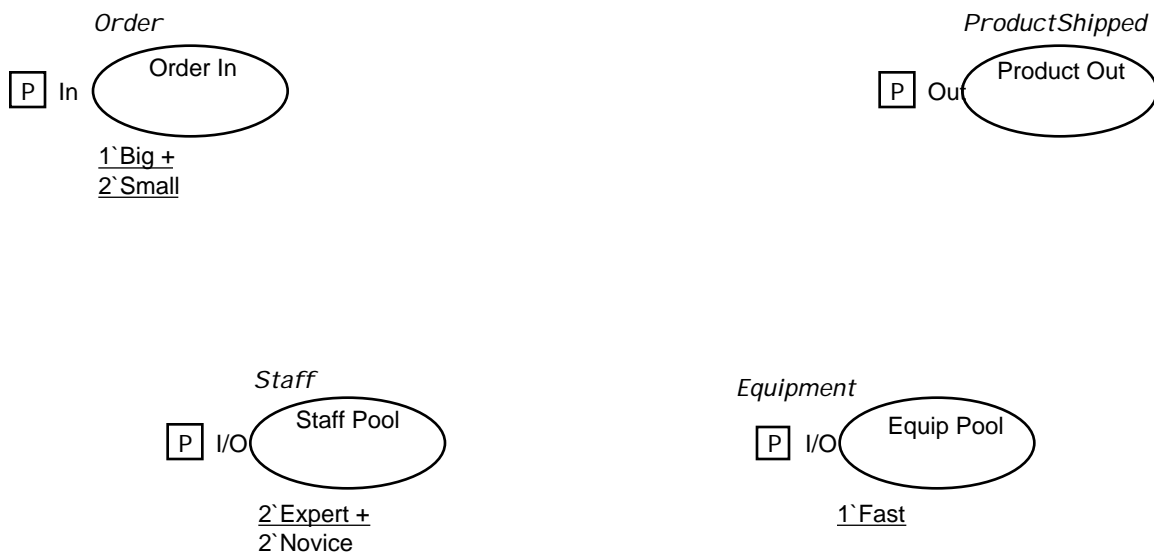
Most of this copying is just a convenience. Except for the port/socket matching, the copied entities have no functional connection with the originals: they could just as well have been omitted, leaving you to supply them yourself if appropriate. By copying them, Design/CPN guarantees that anything on the superpage that might also be useful on the subpage is already there. It thereby gives you the option of deleting things from the subpage if you don't want them, rather than requiring the more difficult course of recreating them if you do.

## Simplifying the Decomposition Page

In this case, the transition and arcs that Design/CPN copied to the subpage will not be helpful: adapting them to fit into the submodel that we will build on the subpage would take much more effort than just deleting them and starting over. Therefore:

- Select the copied transition `Process Orders` on the subpage.
- Delete the transition by pressing `DELETE`.

The transition and all connecting arcs are deleted, leaving nothing but the ports:



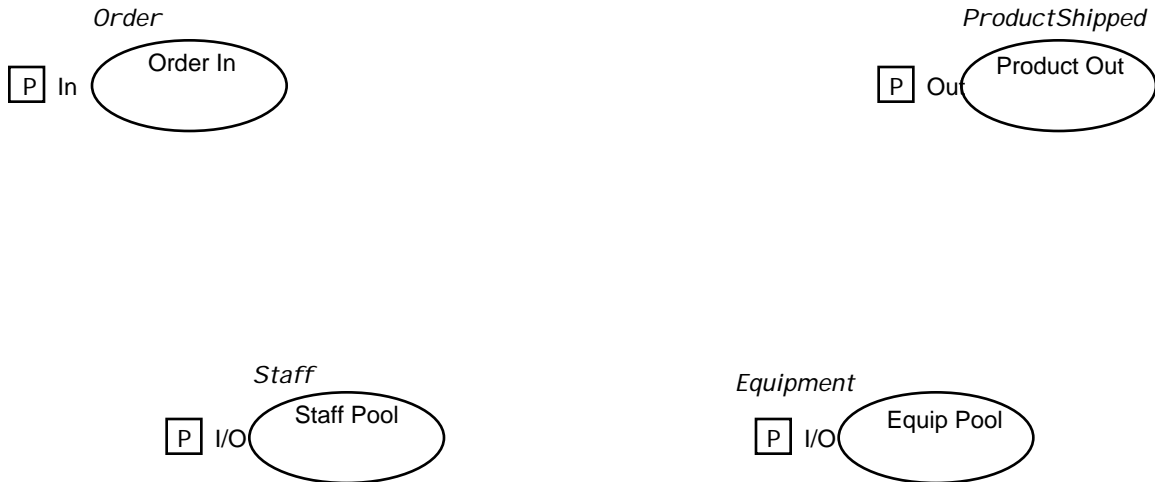
## Hierarchical Decomposition

---

The initial markings on the ports are redundant with those on the corresponding sockets, and aren't necessary for what we will be doing with the subpage, so:

- Make a group that includes all three initial marking regions.
- Delete them all by pressing DELETE.

The subpage should now look like this:



You can now fill in any net structure that you want, and connect it with these ports however you like, without bothering with anything nonessential that was copied from the superpage. Chapter 14 shows how this is done.

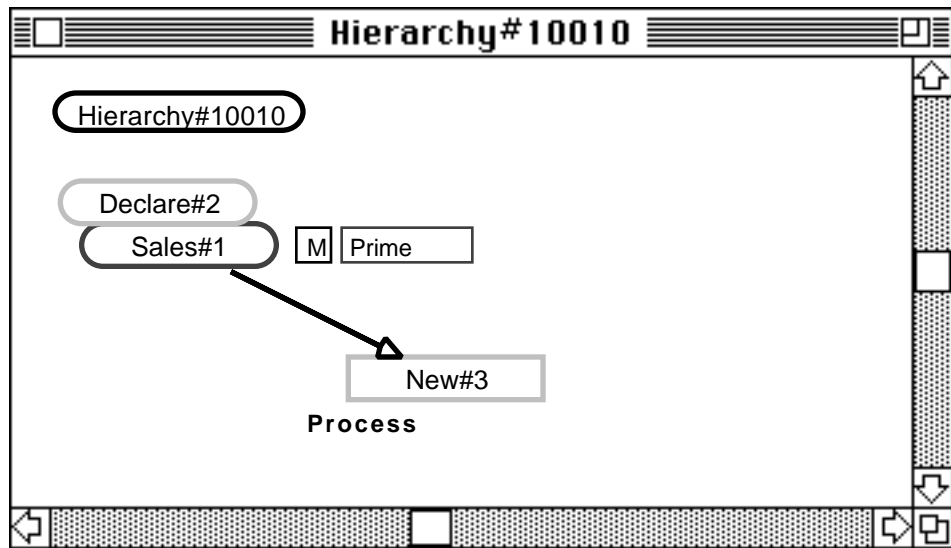
## Substitution and the Hierarchy Page

When you create a substitution transition, Design/CPN automatically updates the hierarchy page to reflect the change.

- Open the hierarchy page.

The page appears:





The arrow between Sales#1 and New#3 indicates that New#3 is a subpage that holds the decomposition of some transition on Sales#1. The label **Process** below New#3 is a *substitution tag region*. It indicates the name of the substitution transition on Sales#1, Process Orders. Names in substitution tag regions are truncated to avoid cluttering up the hierarchy page.

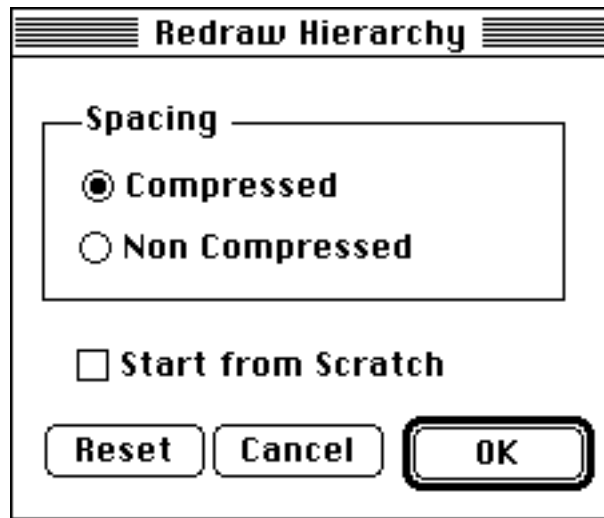
### Improving the Hierarchy Page's Appearance

When Design/CPN updates a hierarchy page, it does not attempt to do so intelligently, because such an attempt could destroy a hand-crafted hierarchy page organization that could not have been produced algorithmically. As always, it uses defaults, and as always, sometimes they work well and sometimes they don't. In this case they have not.

If the hierarchy page was in fact already hand-crafted, we would want to adjust it by hand. But it isn't, so we can tell Design/CPN to do the job automatically.

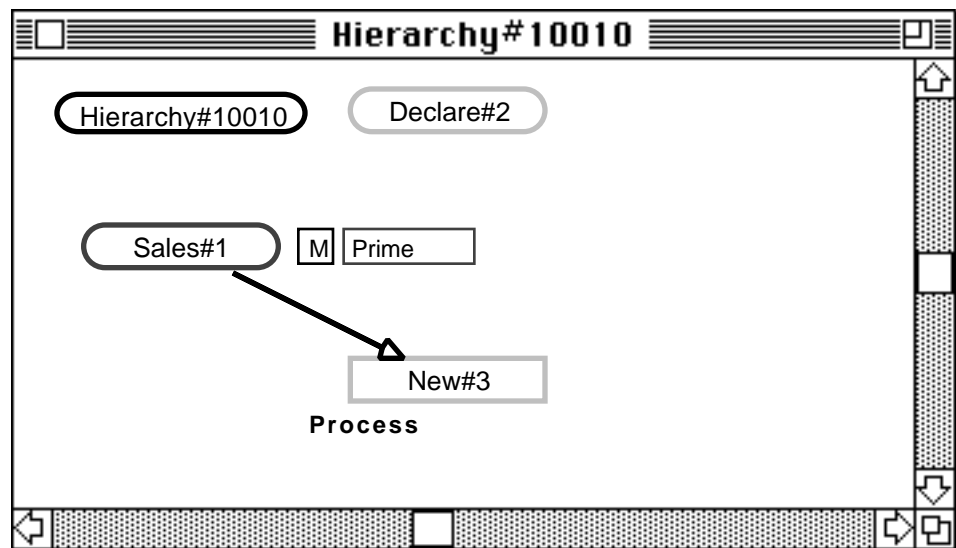
- Choose **Redraw Hierarchy** from the **Page** menu.

The **Redraw Hierarchy** dialog appears:



- Click **OK**.

The dialog disappears. Design/CPN redraws the hierarchy page:



This is a considerably better arrangement.

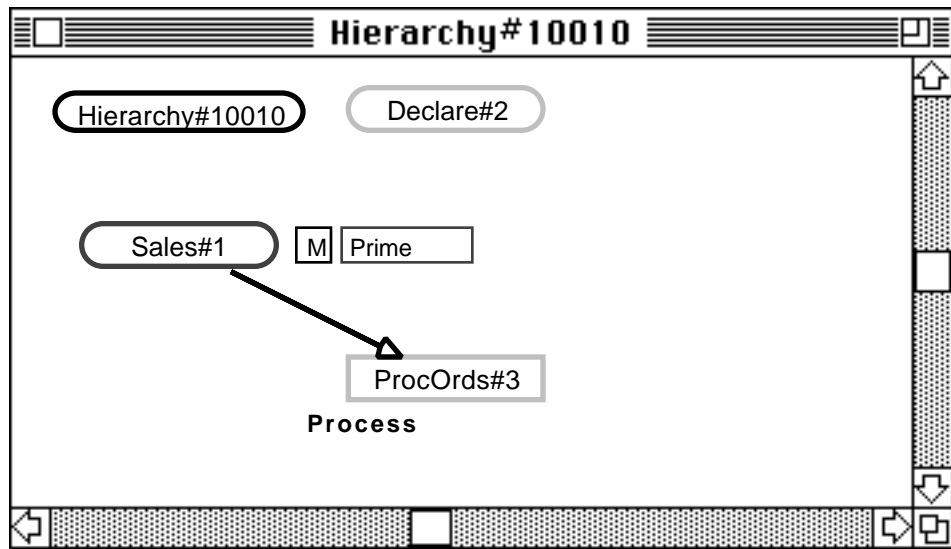
## Renaming the Page

New#3 isn't a particularly meaningful name for the decomposition page, so:

- Select the page node for New#3.
- Choose **Page Attributes** from the **Set** menu.

- Change the new page's name to ProcOrds#3

The hierarchy page should now look like this:



## Saving the Net

In Chapter 14, you will add additional net structure to the decomposition page you created in this chapter. The resulting net won't be SalesNet any more, so it needs a new name. Since it is destined to become a useful model, not just a net for making points about CPN, let's call it FirstModel, and save it in a diagram called NewFirstModel.

- Save NewSalesNet in NewTTDiagrams under the name NewFirstModel.

# Chapter 13

## Understanding a Simple Model

In the previous chapter you modified SalesNet (in the diagram NewSalesNet) by creating a subpage named ProcOrds#3 that contains the beginnings of a hierarchical decomposition for the transition Process Orders. You then saved the modified NewSalesNet in NewTTDiagrams under the name NewFirstModel.

In the next chapter you will add both global declarations and graphical structure to the subpage in NewFirstModel, resulting in a somewhat realistic model of an order processing system. This model is called FirstModel.

It would be of little use to build FirstModel without understanding it: this would just be a rote exercise in editor techniques. In this chapter we will take a very close look at FirstModel, both in itself and in relation to its antecedent, SalesNet.

As you go through this chapter, you may find it useful to open the diagrams that it references and look at them on your screen. The original SalesNet is in SalesNetDemo (in TutorialDiagrams); the current FirstModel, which has a decomposition page that contains nothing but ports, is in NewFirstModel (in NewTTDiagrams), and a completed version of FirstModel can be found in FirstModelDemo (in TutorialDiagrams).

### Overview of FirstModel

FirstModel is a moderately detailed model of a generic order processing system. The model does not specify what product is being ordered, and omits many of the details that would be part of a real order processing operation. The model represents entities of four types:

1. **Orders.** These are requests by customers to buy the product being sold. Orders come in two varieties, designated Big and Small.

2. **Staff members.** These are people who process orders. Staff members come in two varieties, designated Expert and Novice.
3. **Equipment.** These are pieces of machinery that are used by staff members to process jobs. Equipment comes in two varieties, designated Fast and Slow.
4. **Products.** These are the output of the system. There are two types, designated Big and Small. Thus a Big order is big because it is an order for a Big product, and similarly for a Small order.

FirstModel processes orders through a cycle of actions that occurs once for each order. Those actions are:

1. An order begins to be processed.
2. A staff person is assigned from a staff pool to handle the order. A Big job requires an Expert staff person; a Small job requires a Novice staff person.
3. The staff person enters the order into the system in some way.
4. The staff person obtains a piece of equipment from an equipment pool, uses it in some way to process the order, then returns the equipment to the pool. A Big order requires a Fast piece of equipment; a Small order requires a Slow piece of equipment.
5. The staff person ships the product that was ordered.
6. The staff person returns to the staff pool, and is immediately available to process another order.
7. Processing of the order is complete.

### **FirstModel and SalesNet Compared**

The entities in FirstModel are exactly the same entities as in SalesNet, and much the same things are done with them. The difference between SalesNet and FirstModel is entirely a matter of the level of detail.

All we know from SalesNet is that an order is processed using a staff person and a piece of equipment, resulting in the return of the staff and equipment to their pools and the shipping of a product. Such an overview can be quite useful in beginning to organize one's thinking about a system, but it does not constitute a precise description of system organization, and executing it cannot provide exact

measurements of system behavior. To obtain these, we need a more detailed model.

FirstModel begins to show some details of the operation by which orders are processed. It too could be much more detailed than it is, but for now it will suffice.

## Structure of FirstModel

Now let's look at the details of FirstModel, and see exactly how it models the order processing system.

### Data Declarations in FirstModel

FirstModel's data declarations are a superset of those in SalesNet:

```
color Order = with Big | Small;
color ProductShipped = Order;
color Staff = with Expert | Novice;
color Equipment = with Fast | Slow;

var ordent : Order;
var staff : Staff;
var equipment : Equipment;
```

```
color Order = with Big | Small;          (* Orders for products *)
color ProductShipped = Order;            (* Orders shipped *)
color Staff = with Expert | Novice;      (* Staff members *)
color Equipment = with Fast | Slow;      (* Pieces of equipment *)

color OrderEntered = product Order * Staff;  (* Orders entered but unprocessed *)
color OrderProcessed = OrderEntered;        (* Orders processed but unshipped *)

var order: Order;                        (* An order *)
var staff: Staff;                       (* A staff member *)
var equip: Equipment;                   (* A piece of equipment *)
```

(The second figure, containing FirstModel's declaration node, has been reduced in size to fit on the printed page. Many other figures in this and future chapters will be similarly reduced. The versions you actually work with on the computer will all be full-size.)

Order, Staff, Equipment, and ProductShipped have not changed, and are used the same way in SalesNet and FirstModel. The variables ordent and equipment have been replaced by order and equip.

The change from `ordent` to `order` is being made because FirstModel needs to distinguish between orders and entered orders in a way that did not apply to SalesNet, and the term `ordent` could lead to ambiguity under the new circumstances. The change from `equipment` to `equip` is being made to save space: a CP net page can easily become too crowded, and brief names can help to control this trend.

Name changes such as these are common in the development of a real model. Rarely does a model have its final form when first constructed. A real model typically begins with a sketch, such as SalesNet, and evolves by decomposition and other elaborations until it has the desired structure and detail. Some backtracking is inevitable when development proceeds in this way.

### Tuple Colorsets

There is only one new feature to FirstModel's colorsets: the colorset `OrderEntered`:

```
color OrderEntered = product Order * Staff;
```

`OrderEntered` is a *composite colorset*: a colorset that is constructed of other colorsets. CPN ML provides many composite colorsets: records, lists, tuples, and various others. `OrderEntered` is a tuple.

A *tuple* is a colorset that is a cartesian product of two or more other colorsets. Every member of a tuple colorset is an ordered sequence of values, each of which is drawn from the subsidiary colorset defined for its position in the tuple. Thus every member of `OrderEntered` is a sequence of two values, the first of which is of type `Order`, and the second of which is of type `Staff`.

A tuple value is denoted by listing the constituent values in parentheses separated by commas. `OrderEntered` consists of the four values:

```
(Big, Expert)
(Big, Novice)
(Small, Expert)
(Small, Novice)
```

These values are simply the cartesian product of the colorsets `Order` and `Staff`.

### Tuples in FirstModel

Tuples are often used to keep track of transient associations between entities. For example, in FirstModel entering an order into the system is accomplished by temporarily assigning a staff member to pro-

cess the order. This makes it convenient to represent an entered order with a token that consists of the order and the staff member assigned to it. The colorset `OrderEntered` provides such tokens.

The colorset `OrderProcessed` is just a duplicate of `OrderEntered`. In FirstModel, it is also convenient to represent an order that has been processed but not yet shipped with a token that consists of the order and the assigned staff member. We could just use `OrderEntered` tokens again, but the name `OrderProcessed` is more suggestive of the meaning of the tokens at that stage of order processing, so using it makes the model easier to understand.

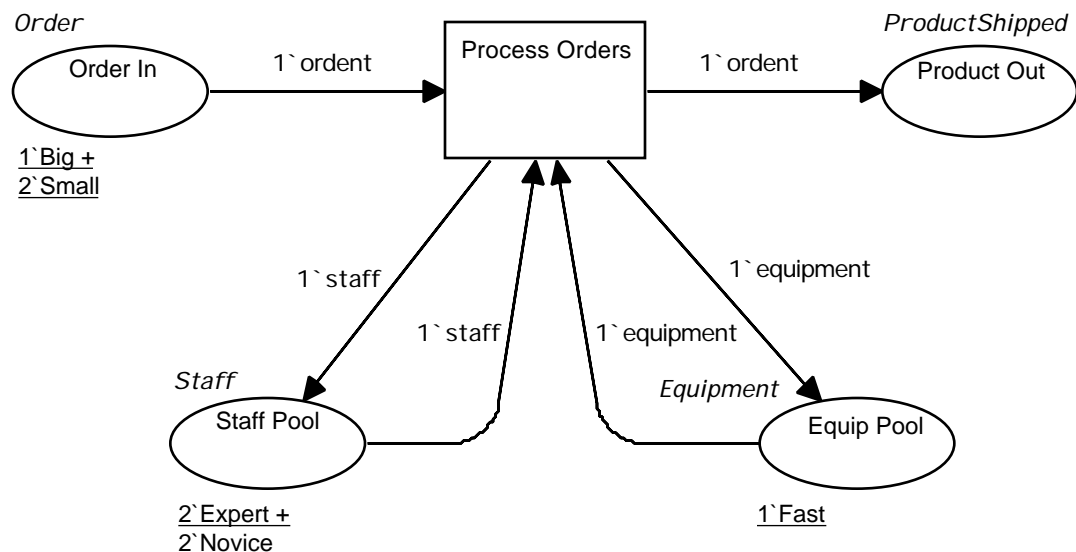
You may have noticed that the global declaration node declares variables for tokens of type `Order`, `Staff`, and `Equipment` (namely `order`, `staff`, and `equip`) but none for `OrderEntered` and `OrderProcessed`. We won't need variables for `OrderEntered` and `OrderProcessed`. The reason will be explained later in this chapter.

### The Superpage in FirstModel

FirstModel's superpage is structurally identical to SalesNet:

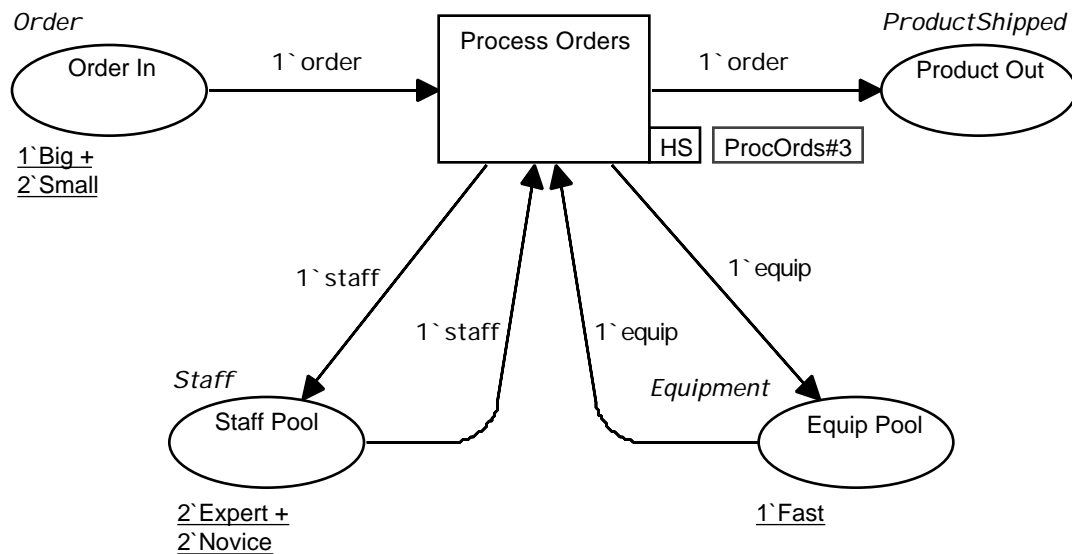
#### SalesNet

[if ordent = Big then staff = Expert else staff = Novice]





## FirstModel Superpage



### **Differences Between SalesNet and the FirstModel Superpage**

The only differences are:

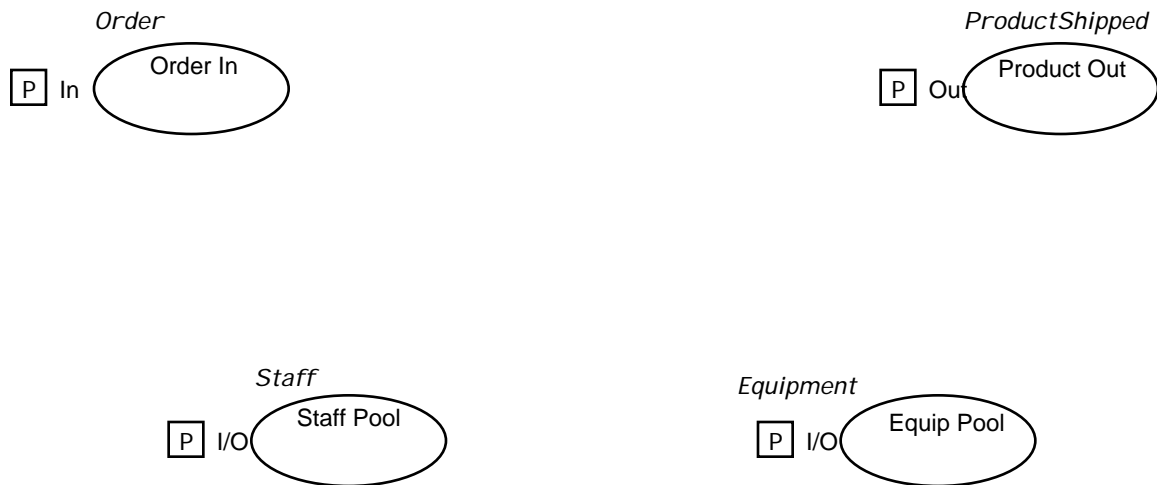
- The variable `ordent` has been replaced by `order`.
- The variable `equipment` has been replaced by `equip`.
- `Process Orders` has become a substitution transition.

The changes to variable names are not strictly necessary. By continuing to declare `ordent` and `equipment` in the global declaration node, we could continue to use them on the superpage, or anywhere else. But keeping them around would add clutter while providing no advantage: it is better to replace them with their updated equivalents, `order` and `equip`. A CPN model can build up areas that are outdated but still functional just as a conventional program can. It is best to prevent this from happening, or clarity will suffer.

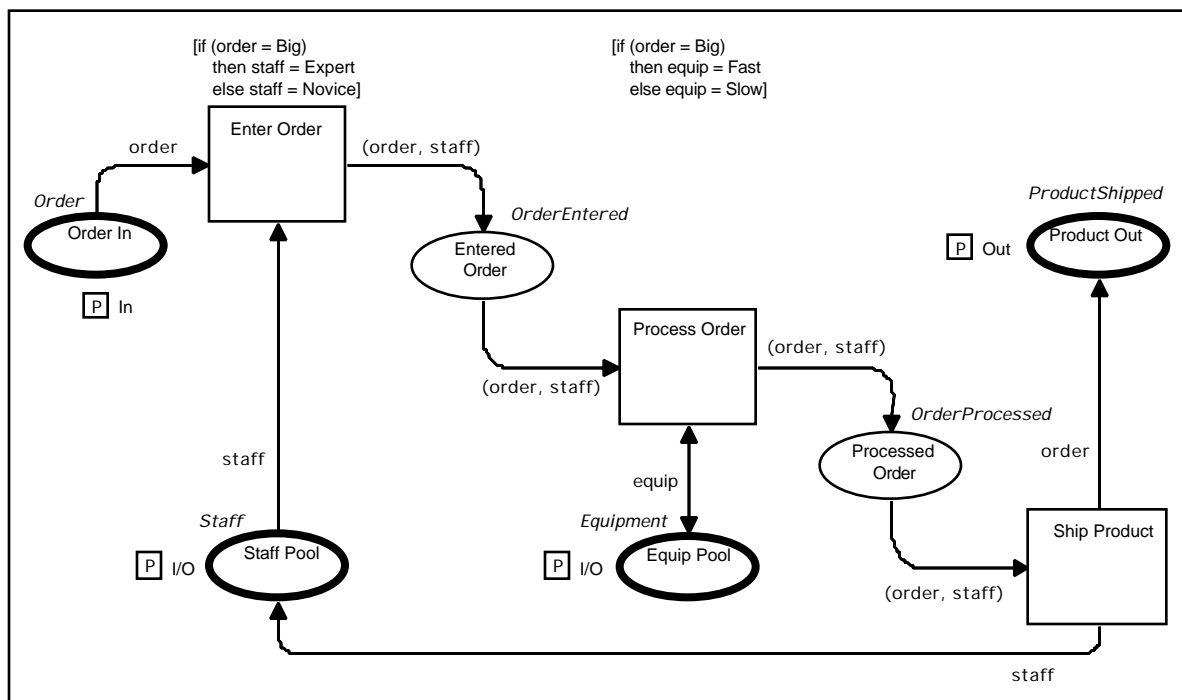
### **The Subpage in FirstModel**

The conversion of `Process Orders` to a substitution transition holds great promise, but it has no significance as yet, because there is nothing executable on the subpage: it contains nothing but a framework of ports. These provide an interface that links the subpage to the superpage:

# Understanding a Model



Soon we will fill in this framework so that it becomes the FirstModel subpage:

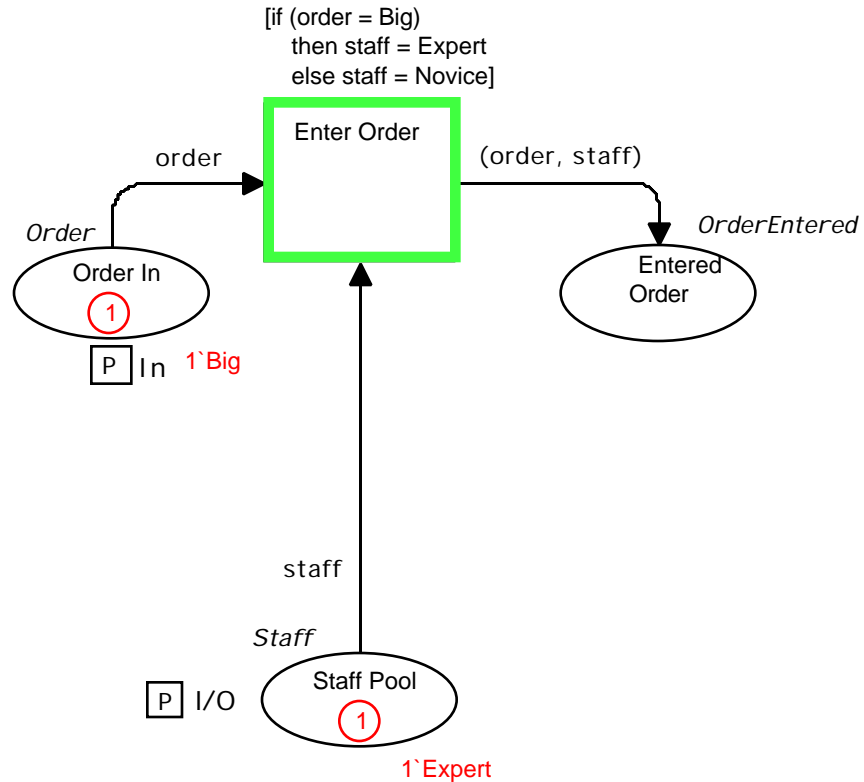


## Function of FirstModel

Now let's look at the details of how FirstModel operates. The simplest way to do this is to look at each of the three transitions and its environment in isolation from the others.

## Enter Order

Enter Order and its environment look like this:



Enter Order is enabled whenever there is a *Big* order in *Order In* and an *Expert* staff member in *Staff Pool*, or there is a *Small* order in *Order In* and a *Novice* staff member in *Staff Pool*. Enabling tokens exist, so the transition is enabled.

When *Enter Order* fires it will remove the *Order* and *Staff* tokens that produced the enabling binding from the input places. But what will it put into *Entered Order*?

## Tuple Constructors

The colorset of *Entered Order* is *OrderEntered*, which was declared as:

```
color OrderEntered = product Order * Staff;
```

That is, every member of *OrderEntered* is a tuple the first element of which has the value *Big* or *Small* (the elements of *Order*) and the second element of which has the value *Expert* or *Novice* (the elements of *Staff*).

The output arc inscription to `Entered Order` has the form of an `OrderEntered` tuple, but instead of using wired-in values of appropriate type, as in:

```
(Big, Expert)
```

it uses CPN variables of appropriate type:

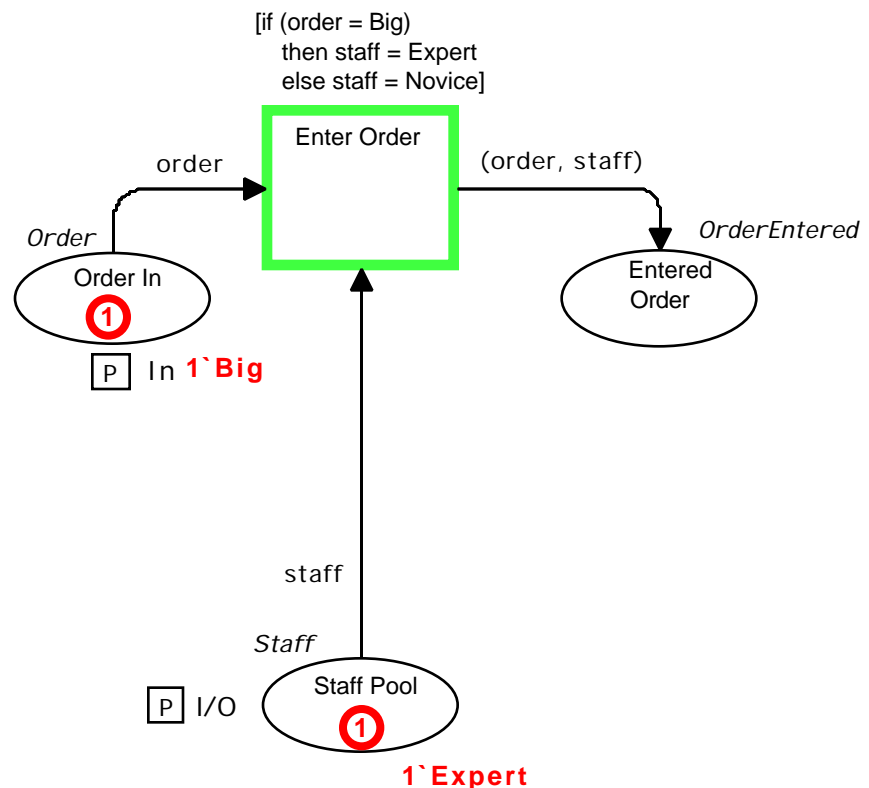
```
(order, staff)
```

This specification is called a *tuple constructor*. It is just a way of specifying a tuple value by using variables rather than constants. For example, if `order` is bound to `Big` and `staff` is bound to `Expert`, then `(order, staff)` evaluates to `(Big, Expert)`.

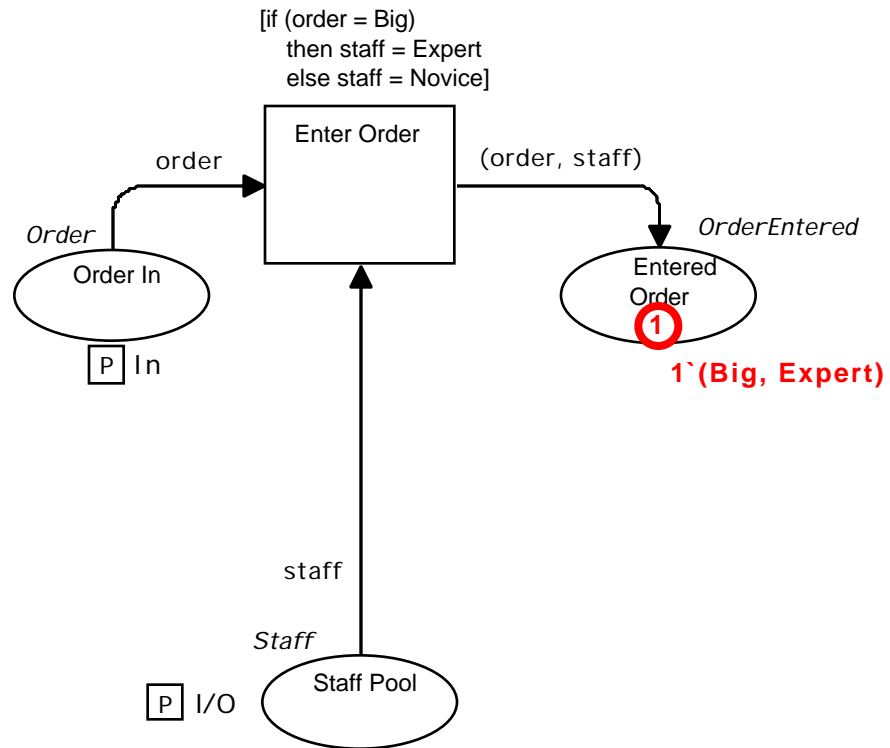
### Example of Tuple Construction

When a transition fires, each output arc inscription is evaluated with any variables used on input arcs bound to the values they have in the enabling binding, and the resulting multiset is put into the output place. When there is no count on an arc inscription, 1 is assumed, so that the inscription on the output arc to `Entered Order` is actually `1^(order, staff)`.

Suppose that `Enter Order` fires with `order` bound to `Big` and `staff` to `Expert`:

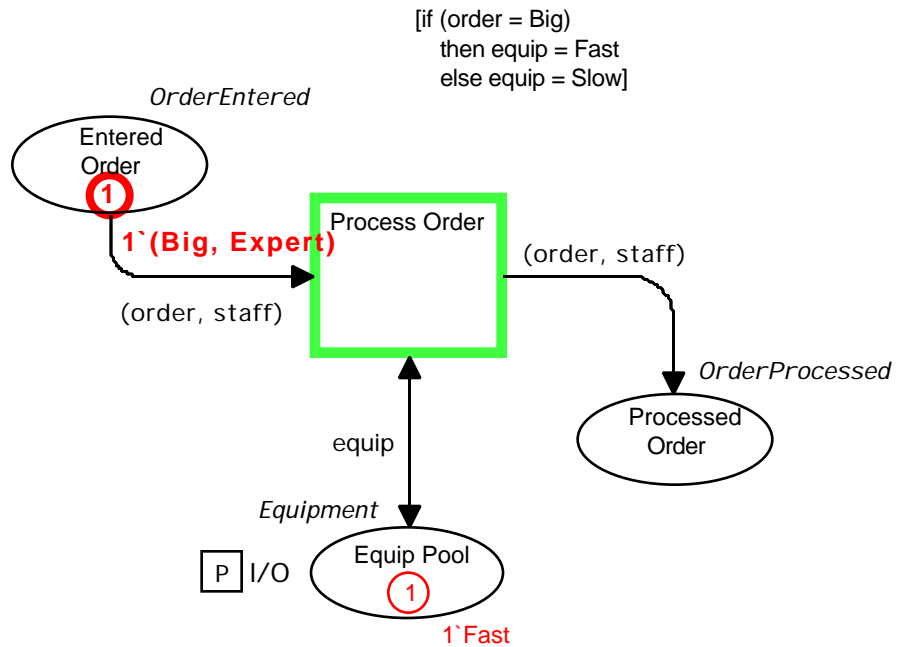


When the constructor  $(order, staff)$  is evaluated with these bindings, the result is the tuple  $(Big, Expert)$ . The output arc inscription thus evaluates to  $1 \cdot (Big, Expert)$ , and the firing of Enter Order will result in one  $(Big, Expert)$  token being put into the output place:



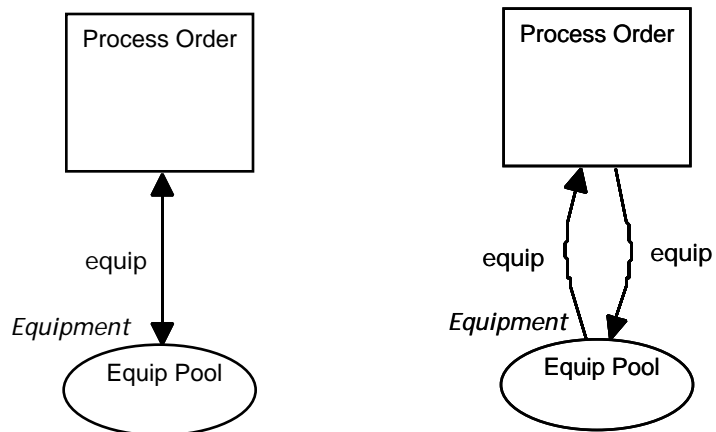
### Process Order

The transition Process Order takes up where Enter Order leaves off:



## Bidirectional Arcs

The arc between **Equip Pool** and **Process Order** has a property we have not seen before: it is bidirectional. Such an arc can be used (but does not have to be) whenever both an input and an output arc connect a given place and transition, and the inscriptions on the two arcs are identical. Thus these two structures are exactly equivalent:



Bidirectional arcs are commonly used when transition enablement requires the existence of a resource that firing does not consume, or of a condition that firing does not change.

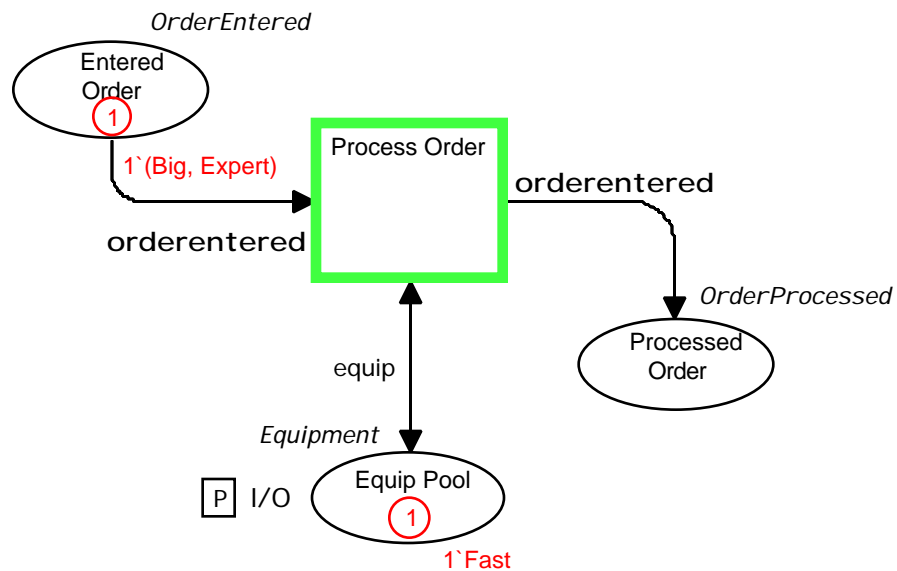
### Tuple Patterns

The key to understanding `Process Order` is the input arc inscription `(order, staff)`, and its interaction with the guard and the other inscriptions.

Suppose that `Process Order` did not care about the composition of the entered orders that it processes, but merely passed them through unexamined and unchanged. We could then declare a CPN variable of type `OrderEntered`:

```
var orderentered: OrderEntered;
```

and `Process Orders` could use this variable to input and output `OrderEntered` tokens:

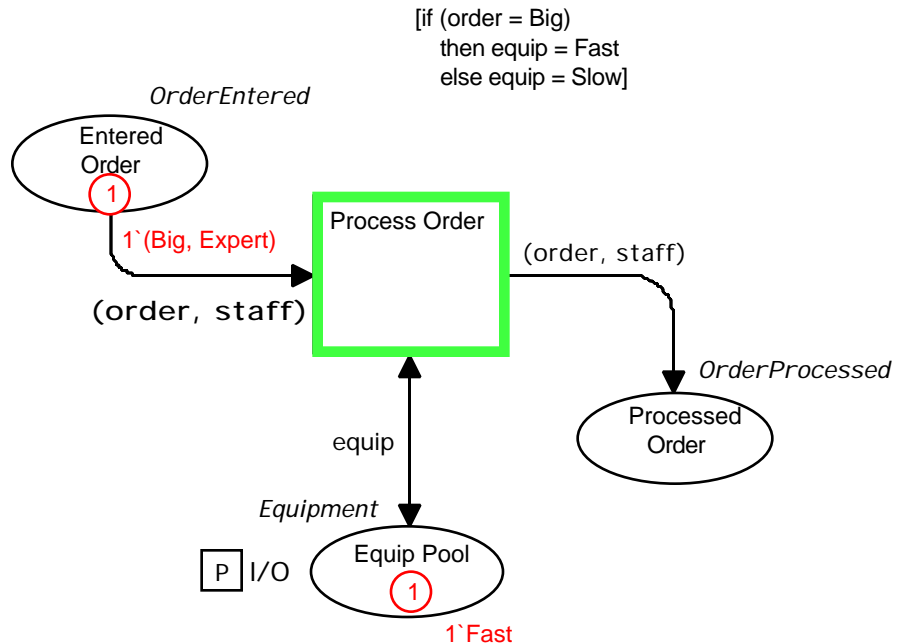


(We can use `orderentered` to put a token into `Processed Order` because `OrderProcessed` is a duplicate colorset of `OrderEntered`.)

Now `Process Order` is enabled whenever there is a token in `Entered Order`, and its firing puts an identical token into `Processed Order`. The fact that the token happens to have a substructure is now irrelevant: it is being dealt with only as a whole.

But there is a price. The substructure of the value bound to `orderentered` is not just irrelevant, it is unavailable. We have lost the ability to predicate the enablement of `Process Orders` on a relationship between order type and equipment type, because this ability requires accessing the constituent values of the tokens in `Entered Order` and testing one of those values in a guard.

There are several ways to access the constituent values of a composite token. The simplest is the one used in the original formulation of `Process Order`:



The specification on the arc between `Entered Order` and `Process Order` is called a *tuple pattern*. It is the inverse of a tuple constructor. Where a constructor takes bound variables and produces a composite value, a pattern takes a composite value and binds its constituents to variables.

When the simulator checks the enablement of a transition with that has a pattern on an input arc, it does not bind whole token values to variables. Instead it matches the value of each token that it looks at against the pattern, and binds the variables in the pattern to the components in the token value. The bound values are then available for use just as if the variables had been bound directly, as `order` and `staff` are in the context of the `Enter Order` transition.

### Enablement of `Process Order`

In the above figure, `Entered Order` contains one token whose value is `(Big, Expert)`. When the simulator checks the transition for enablement, it will match the value of that token against the pattern `(order, staff)`. This will bind `order` to `Big` and `staff` to `Expert`.

The guard is true if `order` is bound to `Big` and `equip` is bound to `Fast`, or `order` is bound to `Small` and `equip` to `Slow`. There is a `Fast` token in `Equip Pool`, so the simulator binds its value to `equip`. Now bindings have been found for all input arc inscription

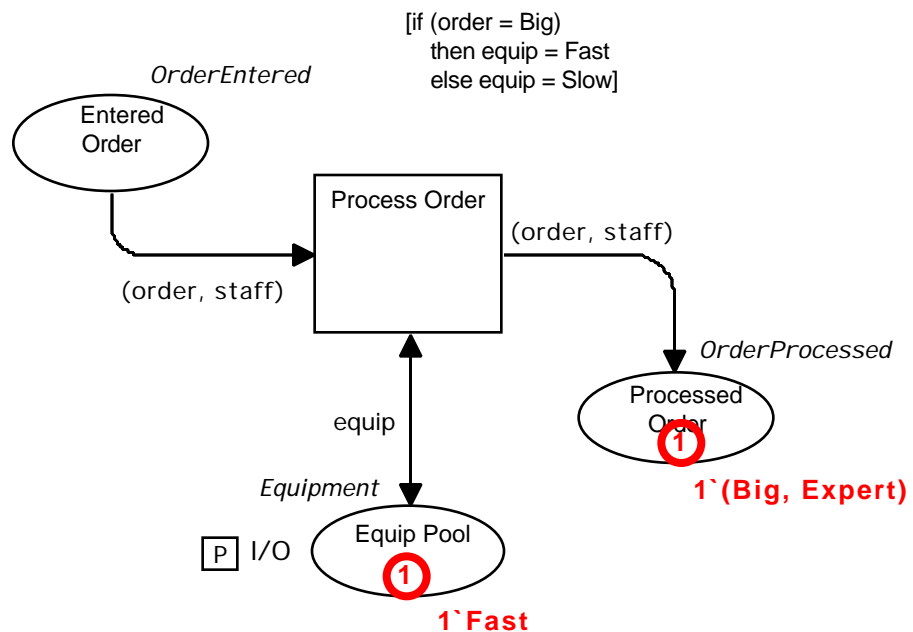


variables such that the multisets that the inscriptions specify exist in the input places, and the guard is true. `Process Order` is therefore enabled with the binding `order = Big`, `staff = Expert`, and `equip = Fast`.

### Firing of Process Order

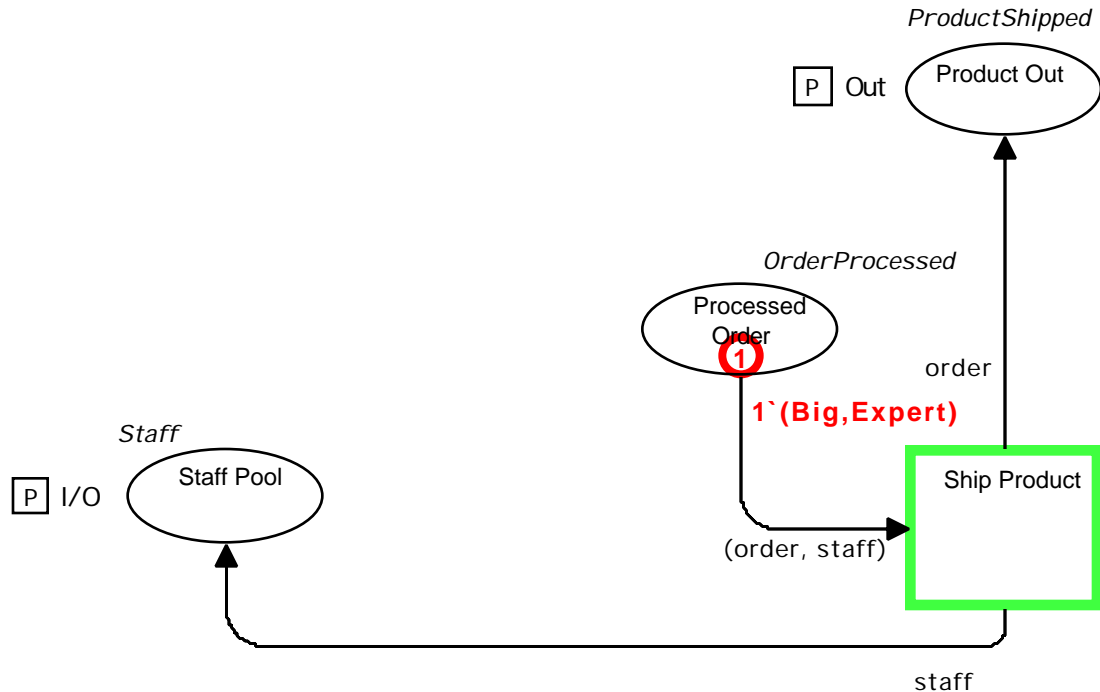
When `Process Order` fires, the enabling binding is restored to all arc variables. The input arc inscription `(order, staff)` then evaluates to `1` (Big, Expert)`, and `equip` to `1`Fast`. These tokens are subtracted from the input places.

Evaluation of the output arc inscription `(order, staff)` is the same as for the firing of `Enter Order`. Given the bindings `order = Big`, `staff = Expert`, the constructor `(order, staff)` evaluates to `(Big, Expert)`, and the inscription to `1` (Big, Expert)`. That token is created and added to `Processed Orders`. Since `equip` is bound to `Fast`, the output arc inscription to `Equip Pool` evaluates to `1`Fast`, and the `Fast` equipment is restored to the pool:



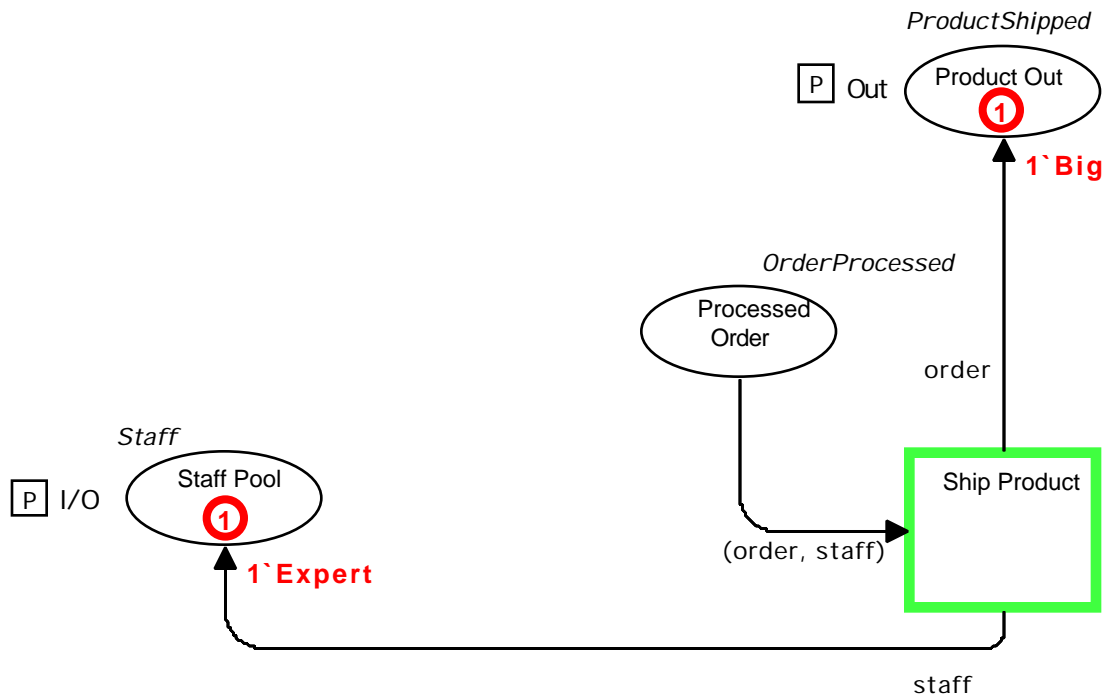
## Ship Product

Ship Product does contain any fundamentally new features:



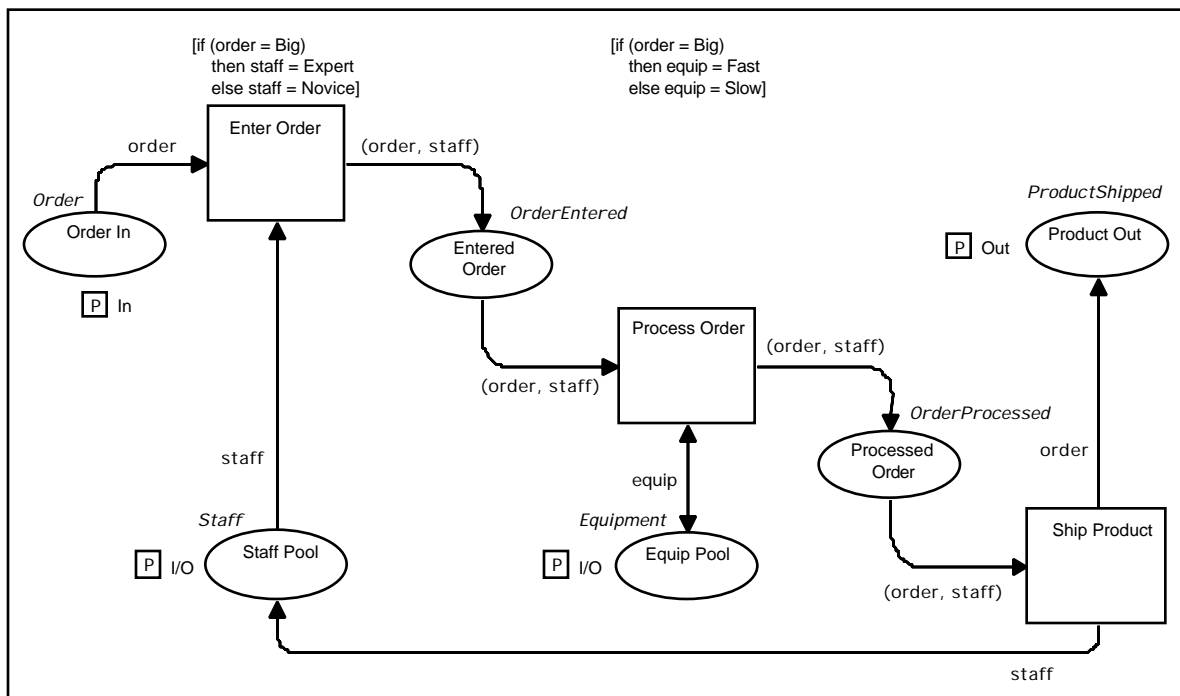
The transition does not specify a guard, so it has an implicit guard of **true**. Thus it is enabled whenever there is a token in *Processed Order*, irrespective of the constituent values of the token.

When *Ship Product* fires, it separates the components of whatever token enabled it and dispatches them to different destinations. It creates a *ProductShipped* token whose value is given by the first element in the enabling token's value, and puts it in *Product Out*; and it creates a *Staff* token whose value is given by the second element in the token's value, and puts it in *Staff Pool*. This makes the staff member available for processing additional jobs:



## Summary of FirstModel

We have now looked at FirstModel piece by piece. Here it is again with the pieces put together:



Doubtless this figure means a great deal more now than it did when you first saw it. Let's summarize what FirstModel does and how it does it.

### Entering an Order

When an order is received (there is an `Order` token in `Order In`), and an appropriate staff member is available (the order is `Big` and `Staff Pool` contains an `Expert` token, or the order is `Small` and `Staff Pool` contains a `Novice` token), the order is entered into the system. (`Enter Order` is enabled and fires. This subtracts the `Order` and `Staff` tokens from their places, and adds a tuple token containing their values to `Entered Order`.)

If there is no appropriate staff member (no token in `Staff Pool` that satisfies `Enter Order`'s guard given the binding of `order`) the order waits until one becomes available because a product has been shipped (until an appropriate token has been added to `Staff Pool` because `Ship Product` has fired).

### Processing an Order

When an order has been entered into the system (there is a token in `Entered Order`) and an appropriate piece of equipment is available (the order is `Big` and `Equip Pool` contains a `Fast` token, or the order is `Small` and `Equip Pool` contains a `Slow` token), the order is processed. (`Process Order` is enabled and fires. This subtracts the tokens from `Entered Order` and `Equip Pool`, replaces the `Equip` token, and adds to `Processed Order` a token identical to the one subtracted from `Entered Order`.)

If there is no equipment piece (no token in `Equip Pool` that satisfies `Process Order`'s guard given the binding of `order`) the order waits until one becomes available because an order has been processed (until an appropriate token has been added to `Equip Pool` because `Process Order` has fired).

### Shipping an Order

When an order has been processed (there is a token in `Processed Order`), a product is shipped and the staff member becomes available to process another order. (`Ship Product` is enabled and fires. This removes the enabling token from `Processed Order`, adds a `ProductShipped` token to `Product Out`, and adds a `Staff` token to `Staff Pool`. The values of the `ProductShipped` and `Staff` tokens are taken from the constituent values of the enabling token.)

### Concurrency in FirstModel

The preceding summary is complete in that it fully describes every piece of FirstModel, but it does not really capture the model's overall behavior. One reason is that it is essentially a sequential view.

It is true that FirstModel puts any one job through a sequence of disjoint steps. But the model as a whole does not necessarily execute sequentially. If there are sufficient orders and resources, it can execute concurrently in two respects.

First, each of its transitions can handle as many orders concurrently as the supply of resources allows. If there are ten jobs and ten staff members of the required types, all ten jobs will be entered in the same step by `Enter Order`; and if there is equipment of the required types, they will all be processed in the same step by `Process Order`. Lesser supplies of resources will of course result in reduced concurrency. This is the same phenomenon that we saw in the case of the single transition in SalesNet.

Second, FirstModel's transitions can execute concurrently with each other. If there are many orders, staff members, and equipment pieces, some orders can be being entered, others processed, and still others shipped, all in the same step.

Nothing special is needed to specify such concurrency among transitions: we just set up the model and the entities it processes, and any concurrency thereby implied naturally arises, both among transitions and for each transition individually. This is a general principle of CP nets.

In trying to understand FirstModel, or any CP net model, it is important to avoid assuming a sequential paradigm. Concurrency is the norm in CP nets. If you approach them by looking for concurrent operations, you will find them much easier to understand than if you concentrate on sequential flows.

Take a moment now to look at FirstModel and imagine it all happening at once. Focus on its behavior as a concurrently operating whole, rather than on the individual orders that it processes by executing a sequence of operations on each one. This shift in viewpoint, from focusing on sequences of transformations of objects to a focusing on concurrencies of processes in systems, is an essential part of learning to work with CP nets.

### Locality in CP Nets

At one level, looking at a model piece by piece gives a complete description of the model. By looking separately at each transition in a CP net and the places directly connected to it, in total isolation from the rest of the net, we can completely characterize the conditions that enable the transition and the result of firing it. Such characterization describes a transition completely: there is no more to a transition than *when it happens* and *what it does*.

Since nothing happens in a CP net except transition firing, a description of every transition is a description of the whole. Everything that the whole does is completely determined by what each transition does separately.

The fact that we can completely understand any transition in isolation from all others is an essential property of CP nets. Every transition constitutes a world of its own, handling inputs and outputs according to fixed rules, with no knowledge of where its inputs come from or where its outputs go. Transitions affect each other only indirectly, by putting tokens in each other's input places: they have no other relationship whatsoever. This mutual independence of transitions is called *locality*.

### Locality and Arc Inscription Variables

You may have wondered about the fact that FirstModel uses the same few CPN variables over and over in inscriptions on arcs connected to different transitions. If all the appearances of a CPN variable anywhere in the net had to have the same binding, useful execution would obviously be impossible.

Locality prevents any problem from arising. The use of a CPN variable by one transition has no connection at all with the variable's use by any other transition. Each transition establishes its own bindings for all variables used on arcs connected to it. Those variables may at the same time be bound to other values by other transitions. If so, the several bindings have no effect on each other whatsoever: each is restricted in scope to the separate world of a particular transition.

### Locality and Overview

Locality has an important general consequence: a CP net contains no inherent overview of itself. If an overview capability is needed, it must be explicitly created as part of the net. But it too will consist of transitions aware only of local information. The only difference will be that the information has global significance.

This property of CP nets might seem a limitation, but in fact it is both realistic and advantageous. Real-world systems have no overall viewpoint unless one has been explicitly created. Often there is no possibility of creating such a viewpoint.

When a human makes a mental model of a system, it is all too easy to inadvertently include in the mental model the effects of a global understanding that exists nowhere in the system itself. A CP net model does not permit this type of error: it is exactly what one makes it be, and no more. Thus modeling a system as a CP net will uncover any areas in which an intelligence has been assumed that does not actually exist in the system.

## Emergent Behavior in CP Nets

But there is another sense in which we definitely do miss something when we look at a CP net only in pieces. Though a complete definition of each piece of a net completely defines the whole, both statically and dynamically, complete understanding of the individual pieces of a CPN model of any complexity rarely sheds any light on what will happen when it executes.

The reason is that the properties that emerge when a CP net executes are functions of the whole net: they cannot be deduced from any part or parts taken separately. If there are only a few parts we may be able to grasp the whole and predict its behavior, but where there are many, as there are usually are when a system is complex enough to be worth modeling in the first place, human understanding cannot possibly predict the outcome of execution.

Thus CP nets allow us to model systems that we cannot possibly understand by looking at their parts, yet never require us to deal directly with anything larger than a part. We can build a model one transition at a time, taking advantage of locality to reduce the model-building task to the construction of small, tractable pieces, so that we never have to cope with the model, or the system, as a whole. When we execute the model, the simulator does the coping for us, leaving us free to concentrate on analyzing the results.

# Chapter 14

## Building a Simple Model

The time comes again to move away from theory and into practice. Let's build FirstModel (in this chapter) and execute it (in the next chapter). The results will clarify many things, and open the way to many more.

In Chapter 12 you modified SalesNet (in the diagram NewSalesNet) by creating a subpage named ProcOrds#3 that contains the beginnings of a hierarchical decomposition for the transition Process Orders. You then saved the modified NewSalesNet in NewTTDiagrams under the name NewFirstModel.

- Open NewFirstModel in the NewTTDiagrams directory.

In order to turn the diagram in NewFirstModel into FirstModel, we must do three things:

1. Add some global declarations, on page Declare#2.
2. Modify the superpage, Sales#1, to reflect the new declarations.
3. Build the model itself, on the subpage ProcOrds#3.

The first two are trivial, but the third should be more interesting. Let's get the easy changes out of the way first.

### Adding Global Declarations

Doing this is just a matter of editing the global declaration node.

- Open the page Declare#2 (if it is not already open).

The global declaration node looks like this:



```
color Order = with Big | Small;
color ProductShipped = Order;
color Staff = with Expert | Novice;
color Equipment = with Fast | Slow;

var ordent : Order;
var staff : Staff;
var equipment : Equipment;
```

- Reshape the global declaration node to fill most of the window.
- Enter text mode.
- Edit the global declaration node so that it looks like this:

```
color Order = with Big | Small;           (* Orders for products *)
color ProductShipped = Order;             (* Orders shipped *)
color Staff = with Expert | Novice;        (* Staff members *)
color Equipment = with Fast | Slow;        (* Pieces of equipment *)

color OrderEntered = product Order * Staff; (* Orders entered but unprocessed *)
color OrderProcessed = OrderEntered;       (* Orders processed but unshipped *)

var order: Order;                         (* An order *)
var staff: Staff;                         (* A staff member *)
var equip: Equipment;                     (* A piece of equipment *)
```

You may be tempted to skip typing in the comments. After all, this is “just an exercise”. But don't omit them: comments are as important in a CP net as in a conventional program.

- Leave text mode.
- Reshape the global declaration node so that it fits its contents.

Take a moment to be sure you understand the meaning of these declarations. If you have any questions, review the previous chapter before proceeding.

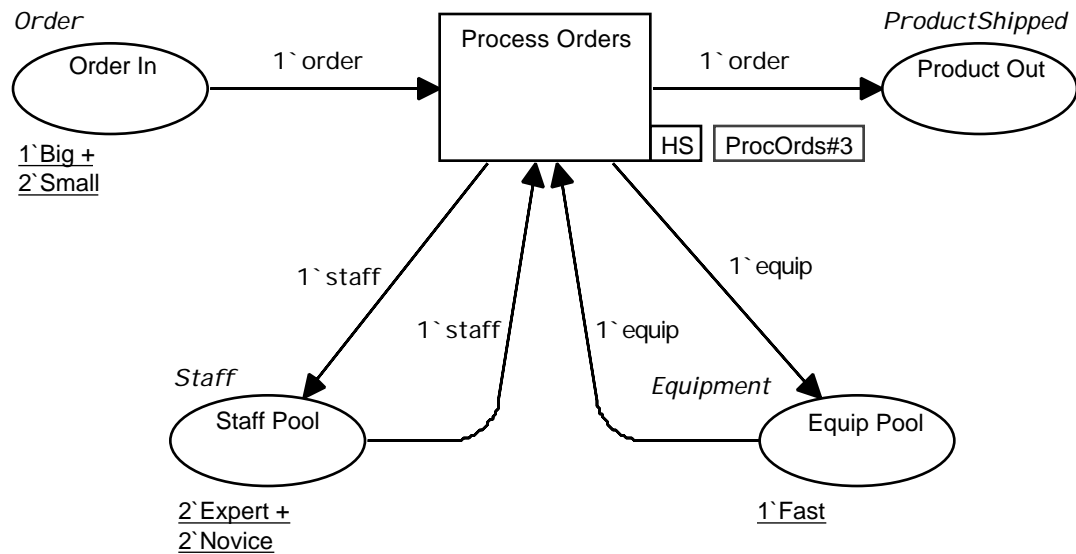
## Modifying the Superpage

Only trivial changes are needed.

- Open the superpage Sales#1.

The page looks like this:

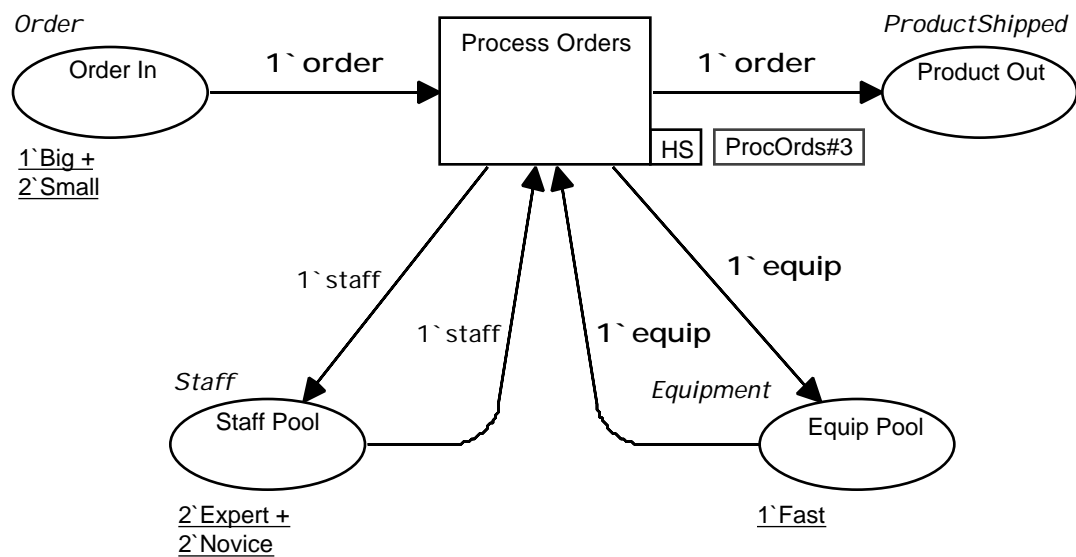
## Building a Model



All we need to do is replace the variable *ordent* by *order*, and the variable *equipment* by *equip*.

- Enter text mode.
- Edit each of the inscriptions that need updating to use the new names.
- Leave text mode.
- Reposition the inscriptions as needed for clarity.

The superpage should now look like this (bolding excepted):



That's it for the superpage.

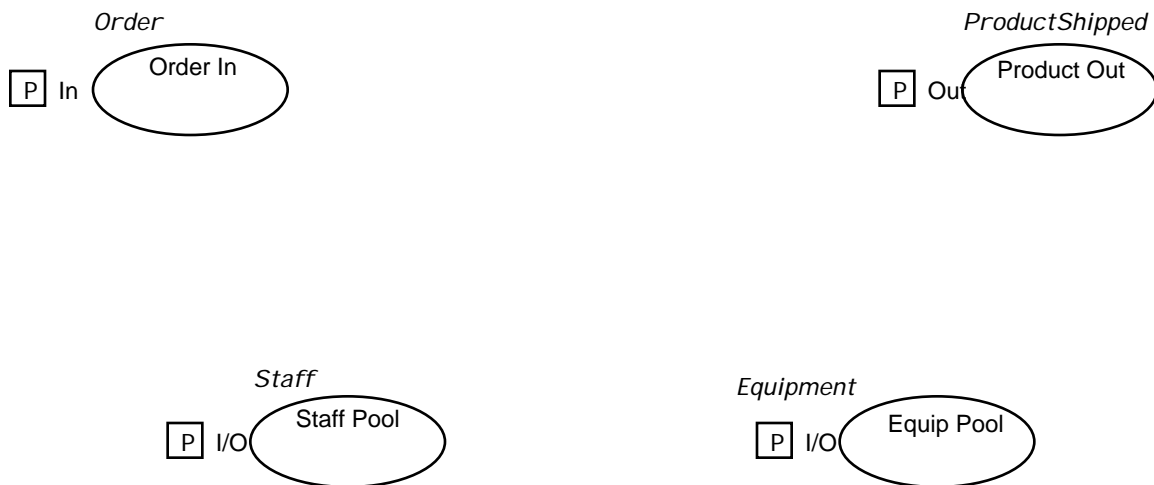
### Building FirstModel on the Subpage

Let's look at ProcOrds#3 as it is and as it will be.

- Open the subpage ProcOrds#3.

#### The Current Subpage

The subpage currently looks like this:

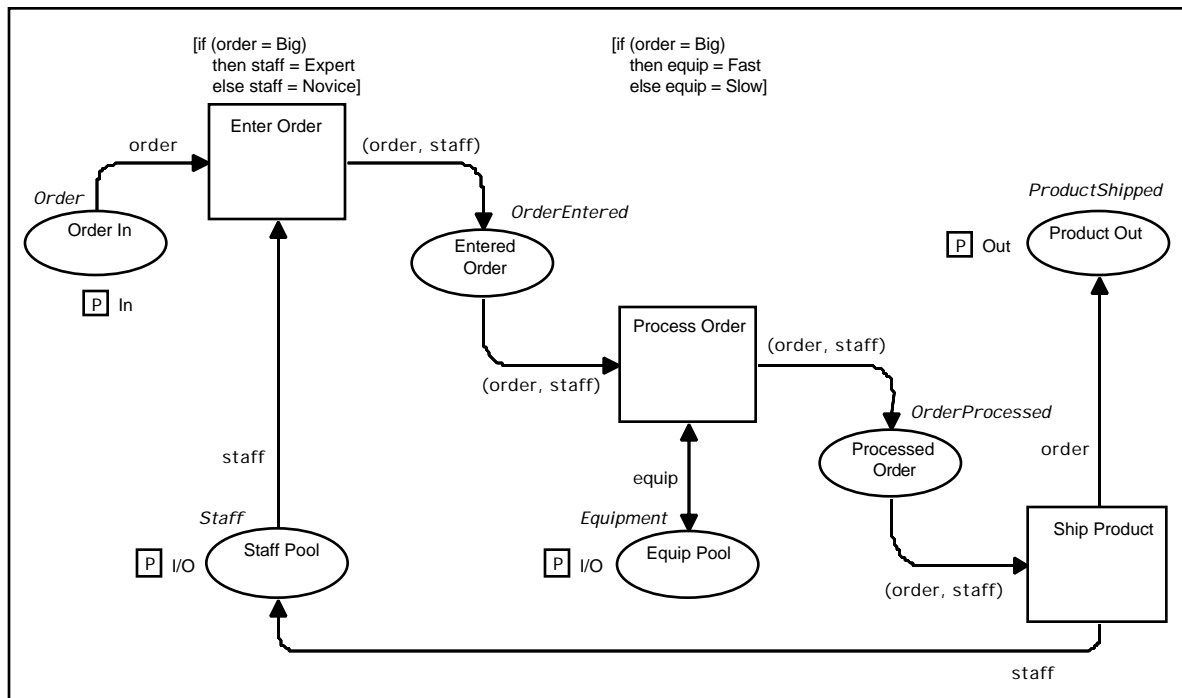


As described in Chapter 12, these four places are ports. Each is equated with the place of the same name on the superpage. The places on the superpage are sockets. A port and its socket are functionally the same place: the marking of one is the marking of the other. This property provides the link between superpage and subpage.

On the superpage Sales#1, the four sockets are connected to a single transition: the substitution transition `Process Orders`. On the subpage, the four ports will connect to a considerably more complex piece of net structure. This structure will not do anything fundamentally different from what `Process Orders` does. Rather, it will express the functionality of `Process Orders` in a more detailed way: one that is more likely to be useful in analyzing the sales order system that FirstModel will represent.

#### The Future Subpage

When you have completed this chapter, ProcOrds#3 will look like this:



This is exactly the net that we looked at in detail in the previous chapter. If you are unclear on any aspect of it, review that chapter before proceeding.

For brevity we will often describe the net structure to be built on page [ProcOrds#3](#) as [FirstModel](#). The model actually includes the global declarations and superpage as well, but we will rarely be concerned with these, so it will often be convenient to ignore them and just look at [ProcOrds#3](#).

This is a common phenomenon in CP net modeling: the global declarations, and the high-level pages, tend to stabilize and become taken for granted; the interesting activity mostly occurs on the sub-pages, where detailed changes are often made and specific execution events can be observed.

## Editing the Subpage

You already know most of the editor techniques you will need in order to build [FirstModel](#). Various additional techniques will be introduced throughout this chapter.

Many of the techniques covered in this chapter relate to improving the graphical appearance of net. These are as important as the basic net construction techniques. Though net appearance has no functional significance, it is not just a matter of aesthetics: it can make the difference between clarity and total incomprehensibility.

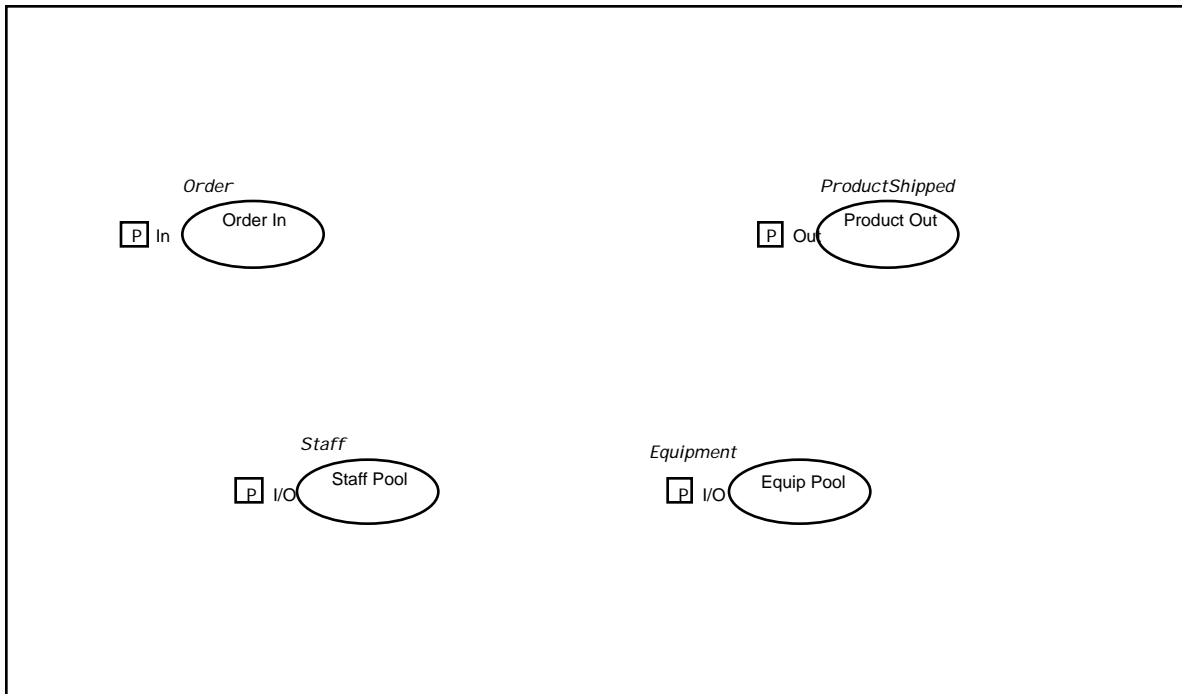
The sequence we will follow to create FirstModel will be:

1. Rearrange the ports on the subpage.
2. Create the transitions.
3. Name the transitions.
4. Create and name the places.
5. Give the new places their colorsets.
6. Align net components.
7. Connect ports, transitions, and new places by drawing arcs.
8. Create the arc inscriptions.
9. Create the transition guards.

This sequence represents only one of many ways in which FirstModel could be created. There is no one best way to create a net: it is a matter of individual preference. The approach used in this chapter has been selected to allow various net-creation techniques to be demonstrated one at a time; it is not necessarily an approach you would want to use generally.

### The Starting Point

All the figures that depict ProcOrds#3 in this chapter are reduced to fit on the printed page. To give you an idea of the scale of reduction, the following shows the ports now on ProcOrds#3:



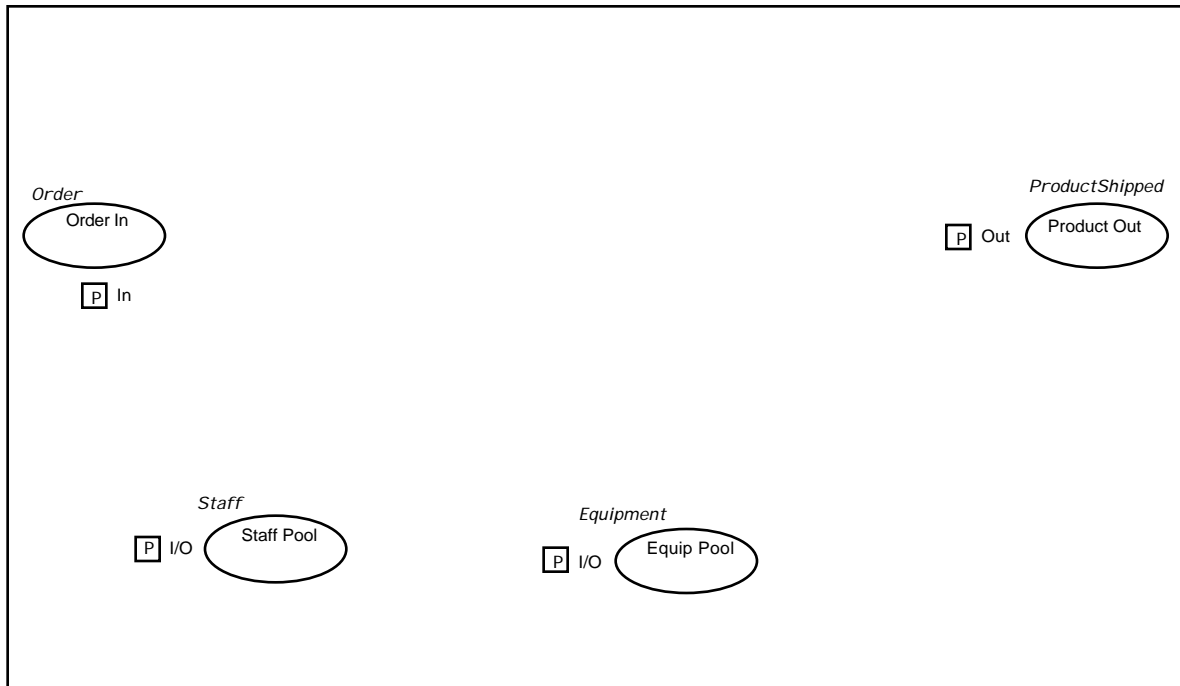
- Adjust the size of the ProcOrds#3 window so that it provides at least as much room relative to the ports as the above figure indicates. If you want a larger window relative to the ports, and your monitor is big enough, make the window as much larger as you like.

## Rearranging the Ports

- Drag the port key region **P** for Order In so that it is immediately below the port.

The port region tracks its parent.

- Drag the port key regions for the other three ports a little to the left, to provide more clearance between the places and the port regions.
- Drag each of the ports so that their positions relative to the window are as shown:

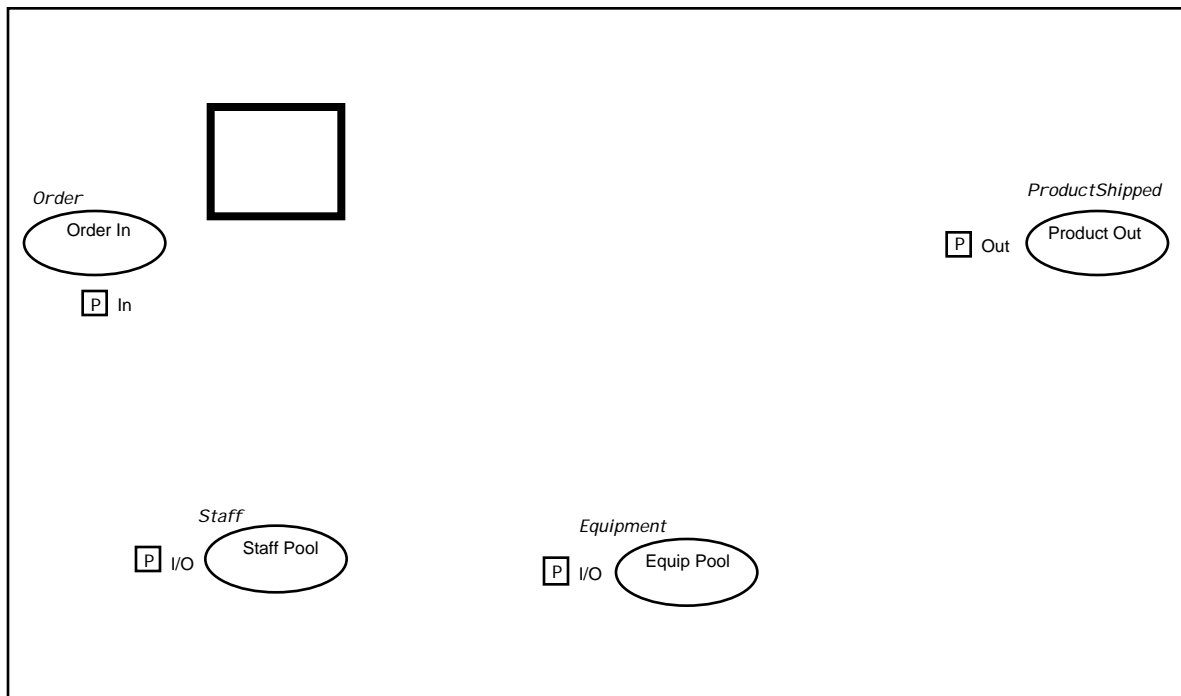


You have now created room for the additional net structure you will be drawing on the subpage. Of course, you could not be so clear about just where to put the ports if you were designing the page from scratch; you would have to experiment somewhat, moving them as required to make room for additional net structure.

As you proceed with these instructions, feel free to make whatever adjustments are needed to create the correct overall result. Don't worry about exactly matching the appearance of the figures: anything essentially equivalent will do.

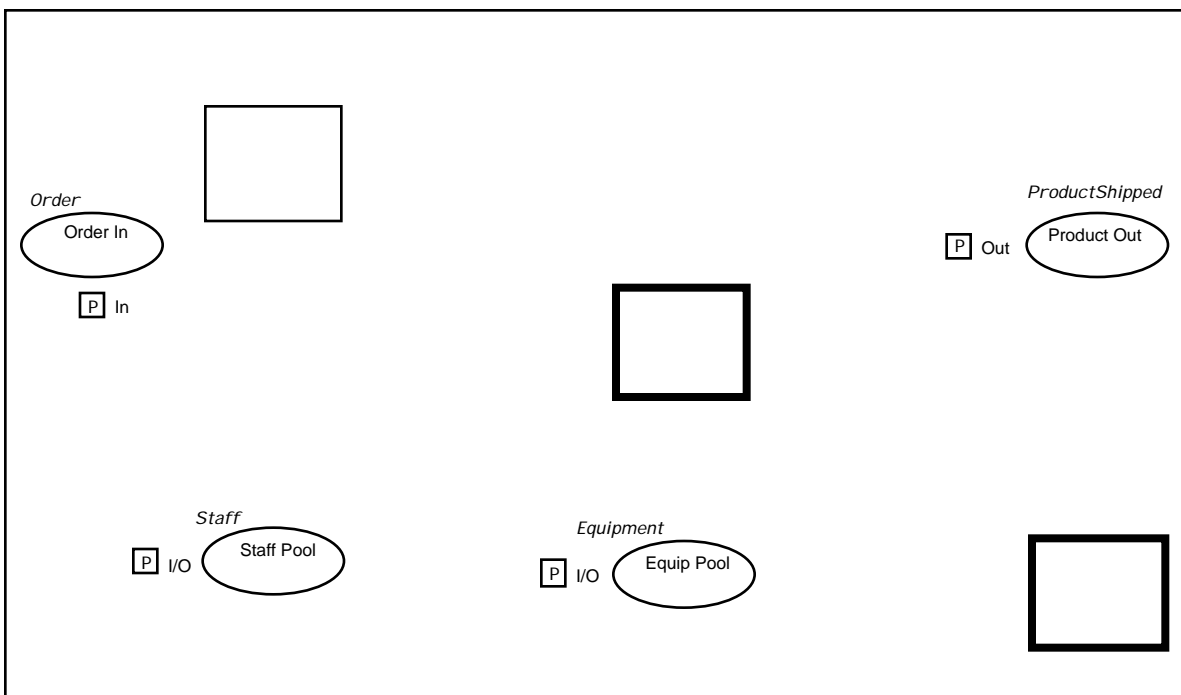
## Creating the Transitions

- Enter transition creation mode (Choose **Transition** from the **CPN** menu)
- Create a transition as shown:



Once you are in transition creation mode, you can create as many transitions as you like:

- Create two more transitions:



- Leave transition creation mode (Press ESC).



- Move the transitions as needed to match the above figure.

### Matching the Transition Sizes

It is difficult to get different graphical objects to have exactly the same size and shape using the mouse. But having similar net components look the same can do a lot to cut down on visual clutter. Design/CPN therefore provides an easy way to make one graphical object look exactly like another. Let's use it to make all three transitions the same size:

- Reshape one of the transitions so that it has the size and shape you want them all to have.
- Make a group that contains the other two transitions.
- Choose **Change Shape** from the **Makeup** menu.

The status bar displays: Select Node or Region as a model for shape change.

- Move the mouse over the transition that has the desired shape.

The transition's border flashes on and off.

- Click the mouse.

The other two transitions are reshaped to be identical to the one that you clicked on.

### Naming the Transitions

- Choose **CPN Region** from the **CPN** menu.

The **CPN Region** dialog for transitions appears.

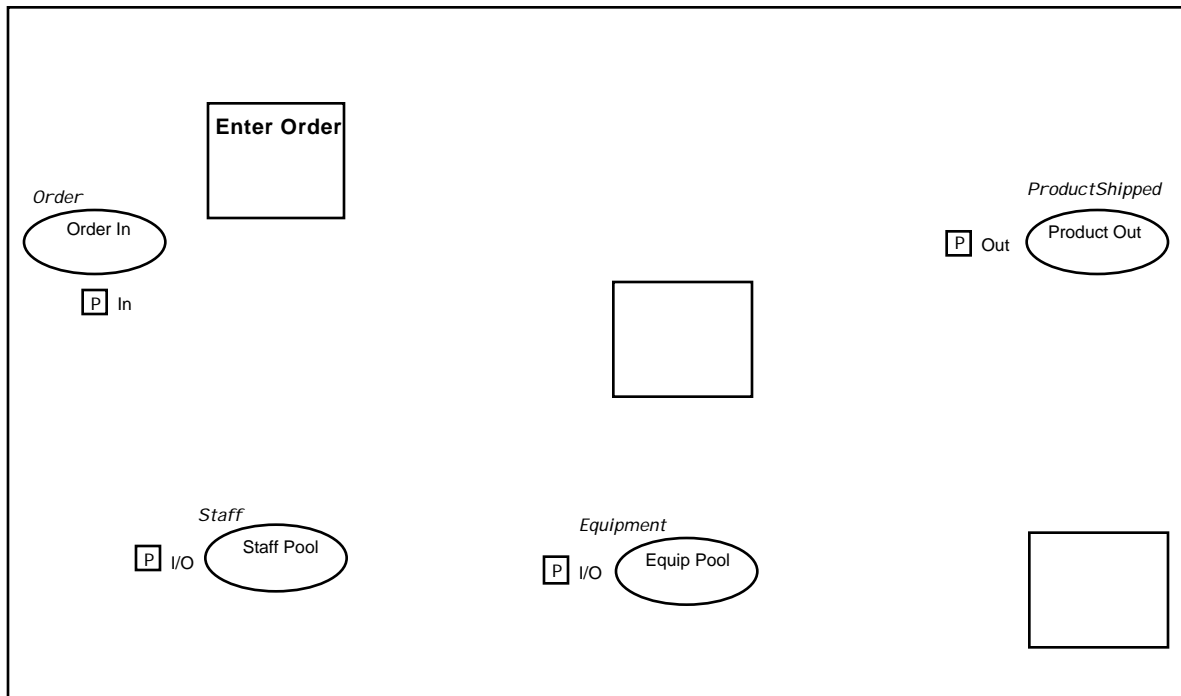
The default is **Name**, which is what we want, so:

- Click **OK**.

The dialog disappears. The editor is now in name region creation mode. The mouse pointer becomes the region tool.

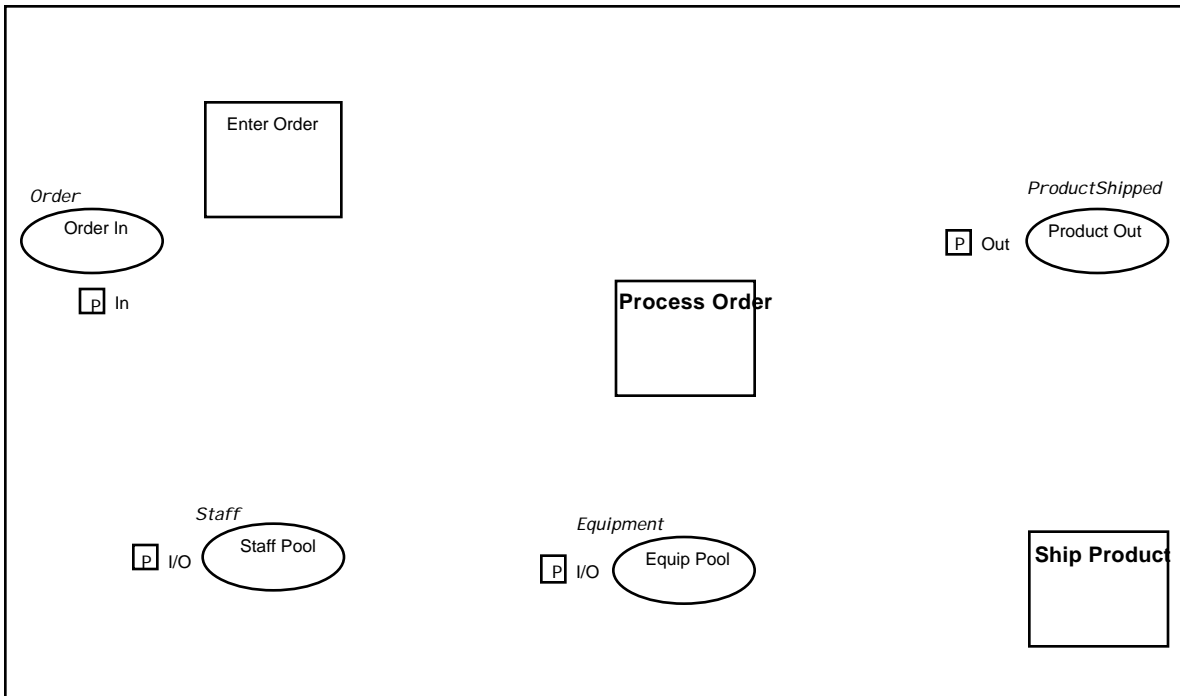
- Move the tool to the inside of the top left transition and click the mouse.
- Type "Enter Order".

The page should look like this, except that the transition name probably won't be so well positioned:



Once you are in name region creation mode, you can name as many transitions (and also places) as you like without leaving the mode.

- Click the mouse on the middle transition.
- Name the transition `Process Order`.
- Name the lower right transition `Ship Product`.
- Leave the creation mode (ESC).
- Use the mouse to position all three names at the tops of the transitions:

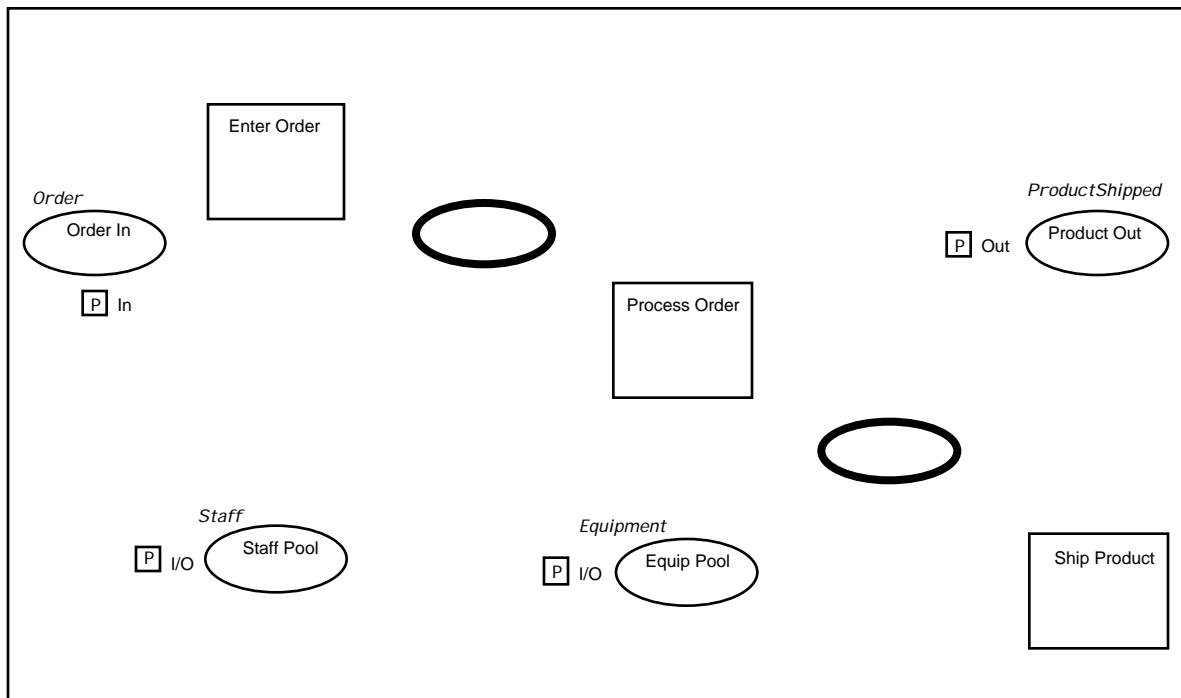


You may need to reshape the transitions so the names will fit inside them.

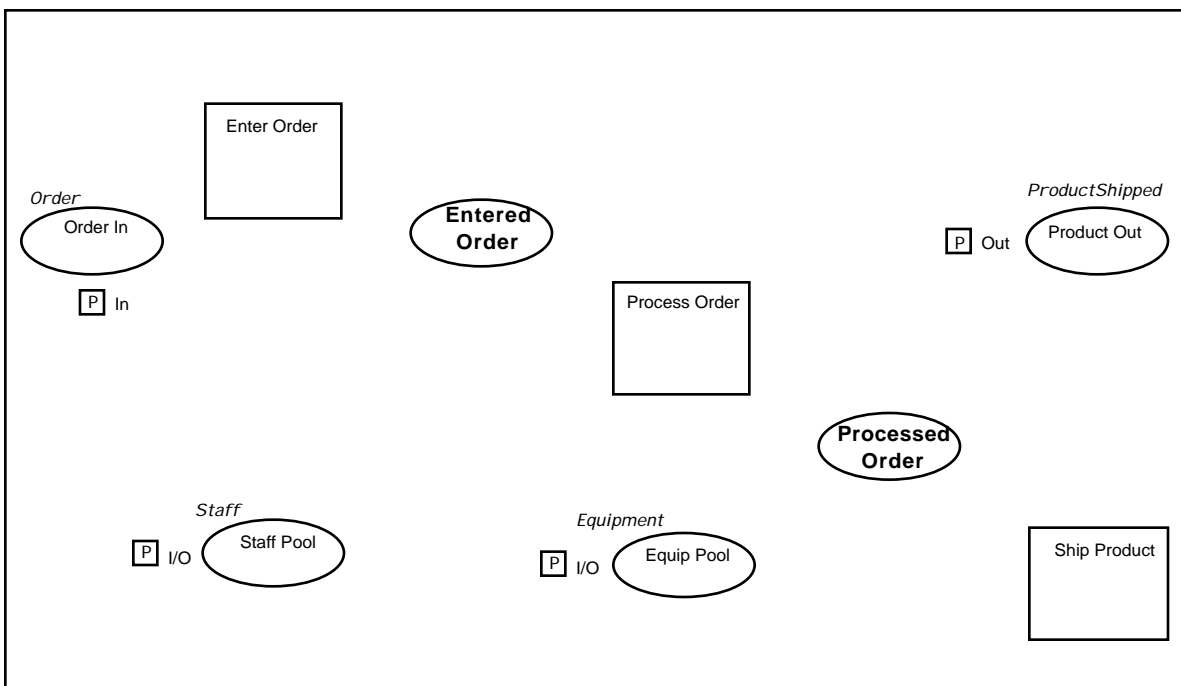
## Creating and Naming the Places

The steps are the same as for transitions, except that you start by selecting **Place** rather than **Transition** from the **CPN** menu.

- Create two places, positioned about like this:

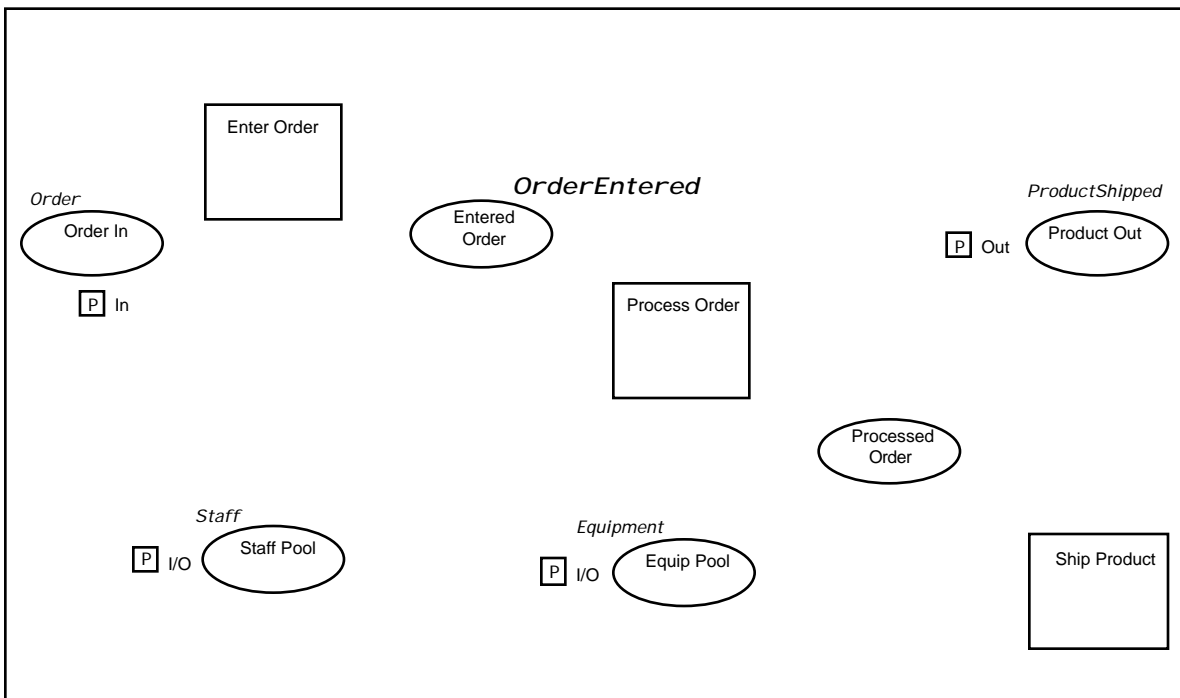


- Use **Change Shape** to make the two new places, and all four ports, the same size.
- Name the places Entered Order and Processed Order. Use linefeeds to make the names break as shown, unless you are making very large places and don't need to, and put the names at the tops of the places:

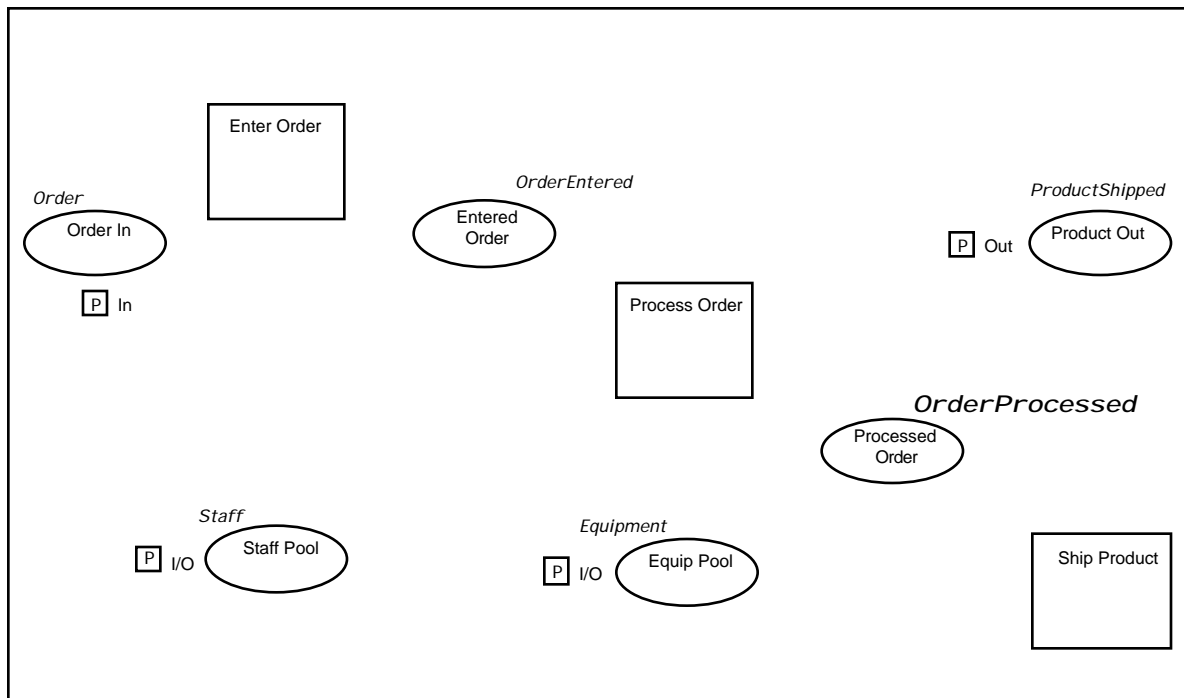


### Give the New Places Their Colorsets

- Select `Enter Order`
- Choose **CPN Region** from the **CPN** menu.
- Click **Color Set**.
- Click **OK**.
- Give `Enter Order` the colorset `OrderEntered`:



- Click on the place `Processed Order`.
- Give `Processed Order` the colorset `OrderProcessed`:



- Leave the mode.
- Adjust the colorset region positions as needed.

## Aligning Net Components

The figures we have been using to illustrate the growing net have everything lined up fairly nicely, but your version is probably less regular. Let's correct that now by doing the following:

1. Align the new transitions and places (nodes) into a diagonal line.
2. Align *Staff Pool*, *Equip Pool*, and *Ship Product* into a row.
3. Move *Staff Pool* directly under *Enter Order*.
4. Move *Equip Pool* directly under *Process Order*.

The changes made in this section are largely a function of the details of the net you are building, and are likely to be too small to show up in a reduced figure anyway, so we won't try to illustrate them.

### Diagonally Aligning the New Nodes

The five new nodes would be in a diagonal line if they were evenly spaced both horizontally and vertically. To accomplish this, we will use two commands from the **Align** menu: **Horizontal Spread**, which evenly spaces graphical objects horizontally between two objects you designate, and **Vertical Spread**, which does the same vertically.

#### Horizontal Spread

- Select all five nodes. Be sure nothing else is selected.

The status bar displays: Group Of 5 Nodes.

- Choose **Horizontal Spread** from the **Align** menu.

You can now designate two nodes, called *reference nodes*, between which the selected objects will be evenly spread. The status bar prompts for the first reference node by displaying: Please select node as reference for alignment.

- Move the mouse pointer through the interiors of various nodes.

A dark border flashes on and off around each node while it contains the mouse pointer: the flashing node is a prospective reference node for the horizontal spread. Note that the reference node does not have to be one of the nodes in the selected group.

- Click the mouse on `Enter Order`.

The status bar prompts for the second reference node by displaying: Please select second node.

- Click the mouse on `Ship Product`.

The five nodes are now evenly spaced horizontally, but not vertically.

#### Vertical Spread

Frequently a horizontal (or vertical) spread alone gives the result is needed, but in this case we need both. As with a horizontal spread, the reference nodes for a vertical spread don't have to be members of the group that is affected. Such membership is optional.

- Remove `Enter Order` and `Shipped Product` from the selected group
- Choose **Vertical Spread** from the **Align** menu.

- Click the mouse on Enter Order.
- Click the mouse on Ship Product.

The five nodes are now evenly spaced both horizontally and vertically, and so are in a diagonal line.

### Aligning Nodes Into a Row

The nodes Staff Pool, Equip Pool, and Ship Product would be more aesthetically arranged if they were aligned into a row.

- Create a group that contains all three nodes.

The status bar displays: Group Of 3 Nodes.

- Choose **Horizontal** from the **Align** menu.

This command aligns nodes horizontally, so that the centers of the nodes lie exactly in a row.

- Click the mouse on Ship Product.

The three nodes (specifically, their centers) are now in a row. The row passes through the center of the reference node Shipped Product. This node need not have been a member of the group to be aligned.

### Aligning Nodes Into a Column

You probably noticed **Vertical** in the **Align** menu. It has the same effect on vertical alignment that **Horizontal** does on horizontal alignment.

Lets put Staff Pool directly under Enter Order, and Equip Pool directly under Process Order. Things will look a lot better later if we do.

**Align** menu commands work the same way on single nodes as they do on groups:

- Select Staff Pool
- Choose **Vertical** from the **Align** menu
- Click on Enter Order.

Staff Pool is now directly below Enter Order.



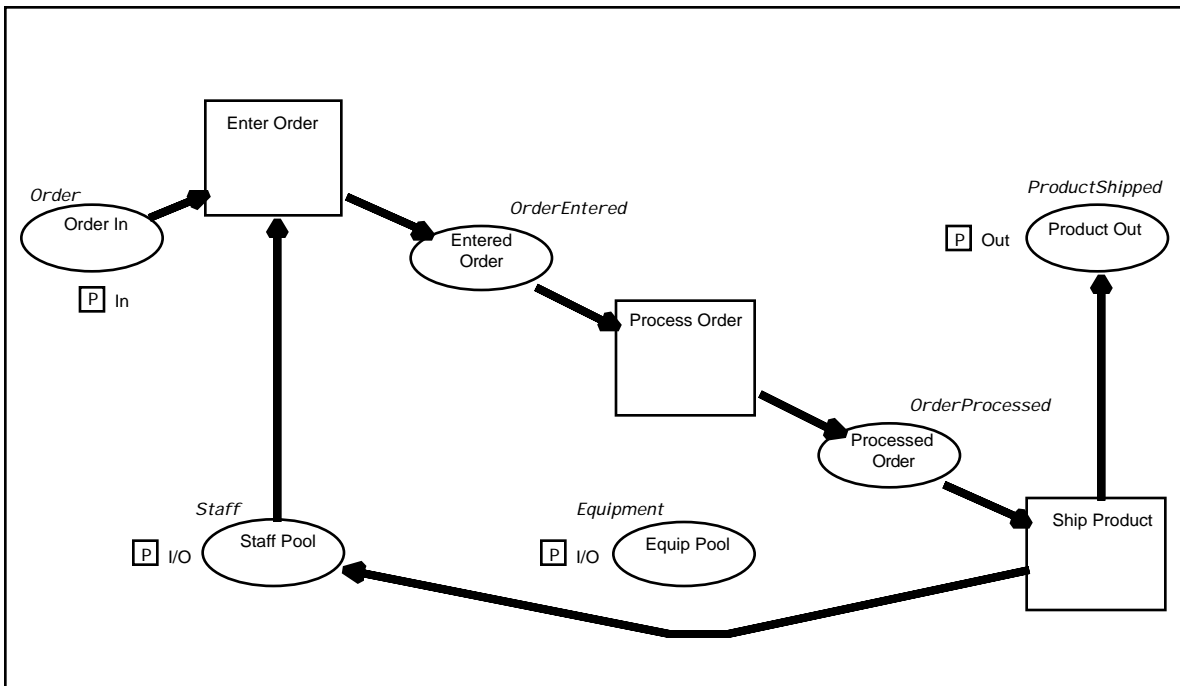
- Align Equip Pool directly below Process Order.

### Other Adjustments

If there are other adjustments that would make your particular net look better, perform them now. If you don't like the results of any command in the **Align** menu, you can undo the operation by choosing **Undo** from the **Edit** menu.

### Connect the Net Components With Arcs

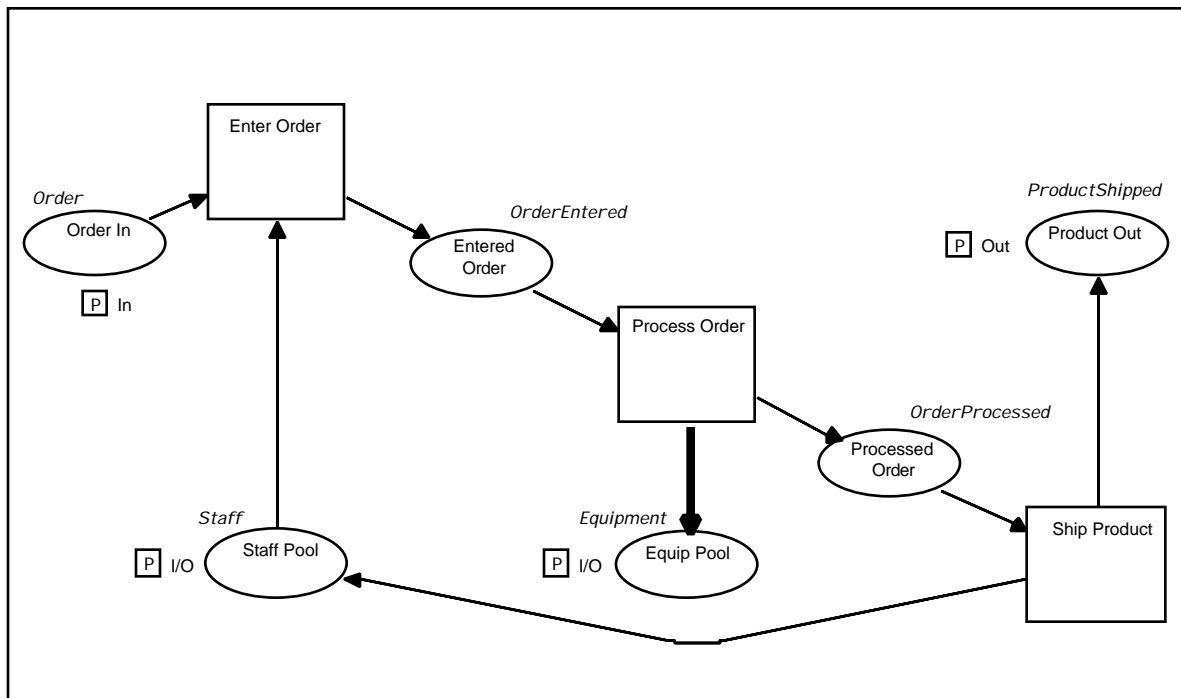
- Select **Arc** from the **CPN** menu.
- Draw the arcs shown:



### Drawing a Bidirectional Arc

The arc from Process Orders to Equip Pool is unusual in that it is bidirectional. To draw it:

- Draw an ordinary arc from Process Order to Equip Pool:

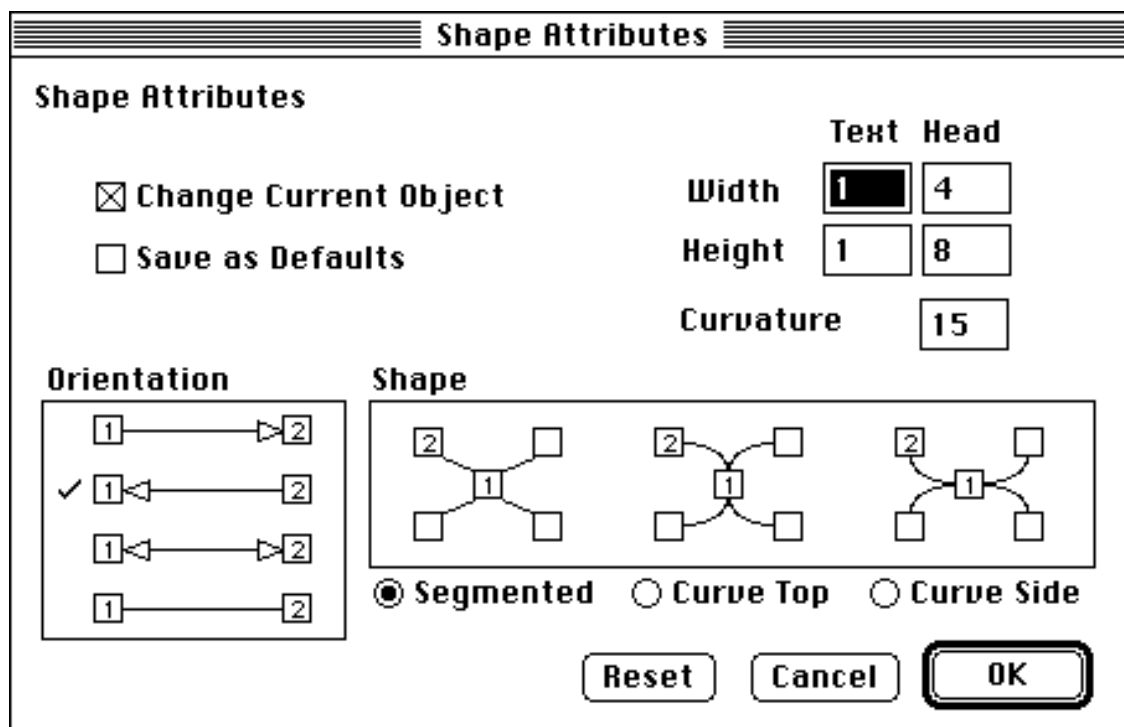


- Leave arc creation mode.

The new arc is the current graphical object.

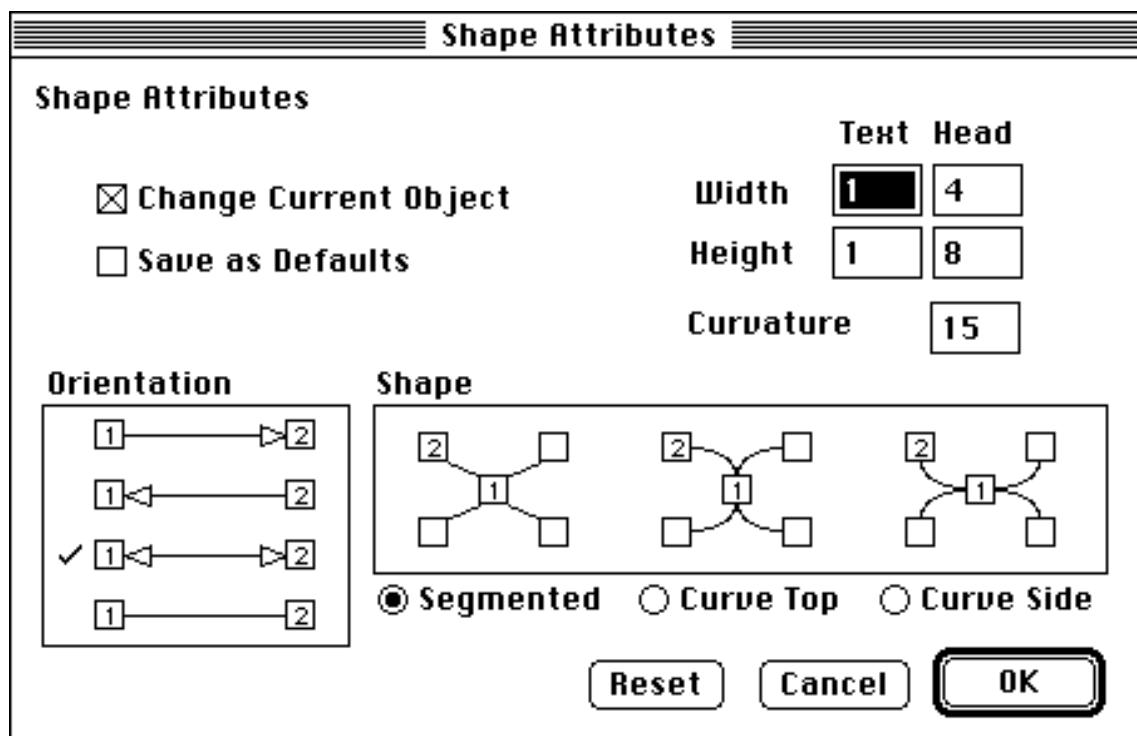
- Select **Shape Attributes** from the **Set** menu.

The **Shape Attributes** dialog for arcs appears:



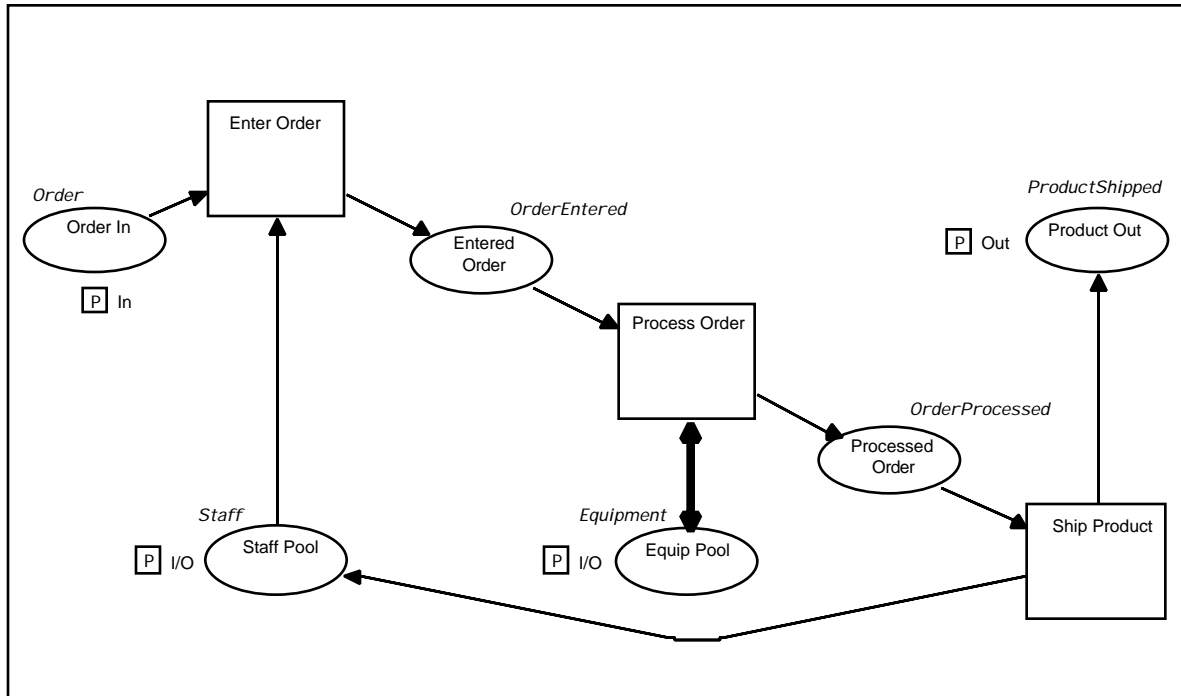
- Under **Orientation**, click on the third option, the double-headed arrow:

The option becomes selected:



- Click **OK**.

The arc between `Process Order` and `Equip Pool` is now bi-directional:



### Adjusting Arc Appearance

The arcs as now drawn are functionally correct, but they would look better, and provide more room for attaching arc inscriptions, if they followed right-angled paths.

Two tricks are often helpful in adjusting arcs:

1. To make a handle of an arc the center of a straight line, press **SPACEBAR** while you are dragging the handle.
2. To make a handle the vertex of a right angle, drag the handle until the “jaggies” caused by the granularity of screen pixels disappear from the arc segments adjacent to the vertex.

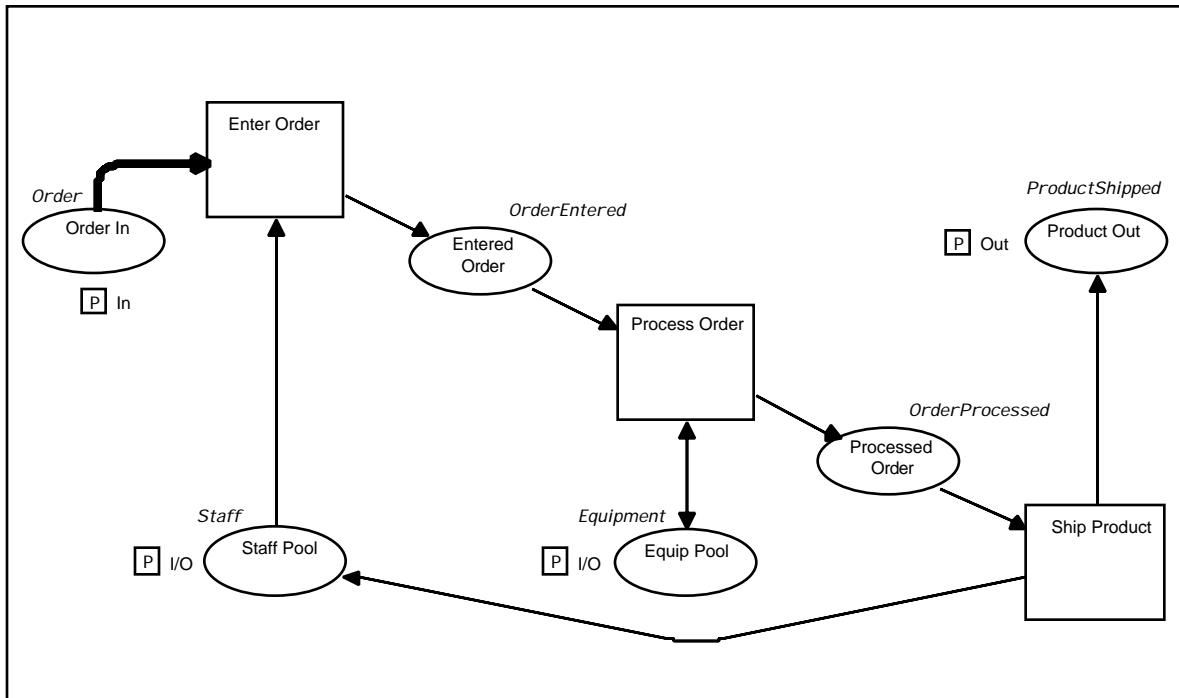
First adjust the arc leading from `Order In` to `Enter Order`:

- Select the arc.
- Position the mouse pointer over the handle at the center of the arc.
- Depress the mouse button.

## Design/CPN Tutorial

- Make the arc exactly right-angled by eliminating the “jaggies”.
- Release the mouse button.

The arc is now right angled:

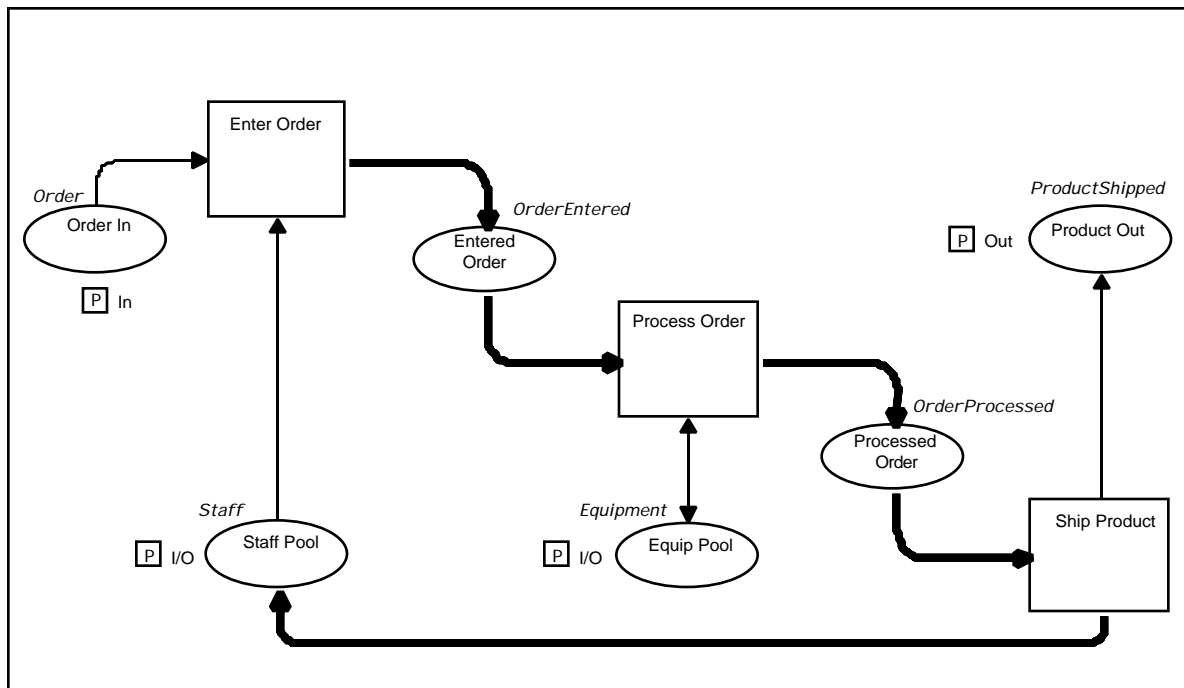


Just for practice:

- Position the mouse pointer over the adjustment point at the center of the arc.
- Depress the mouse button.
- Press SPACEBAR .

The arc is now a straight line.

- Make the arc right-angled again.
- Adjust the other arcs as needed to make the net look like this:



## Creating the Arc Inscriptions

You can create a series of arc inscriptions just as you can create a series of transition names. The technique can be used with all types of CPN text region.

- Select the arc between `Process Order` and `Equip Pool`.

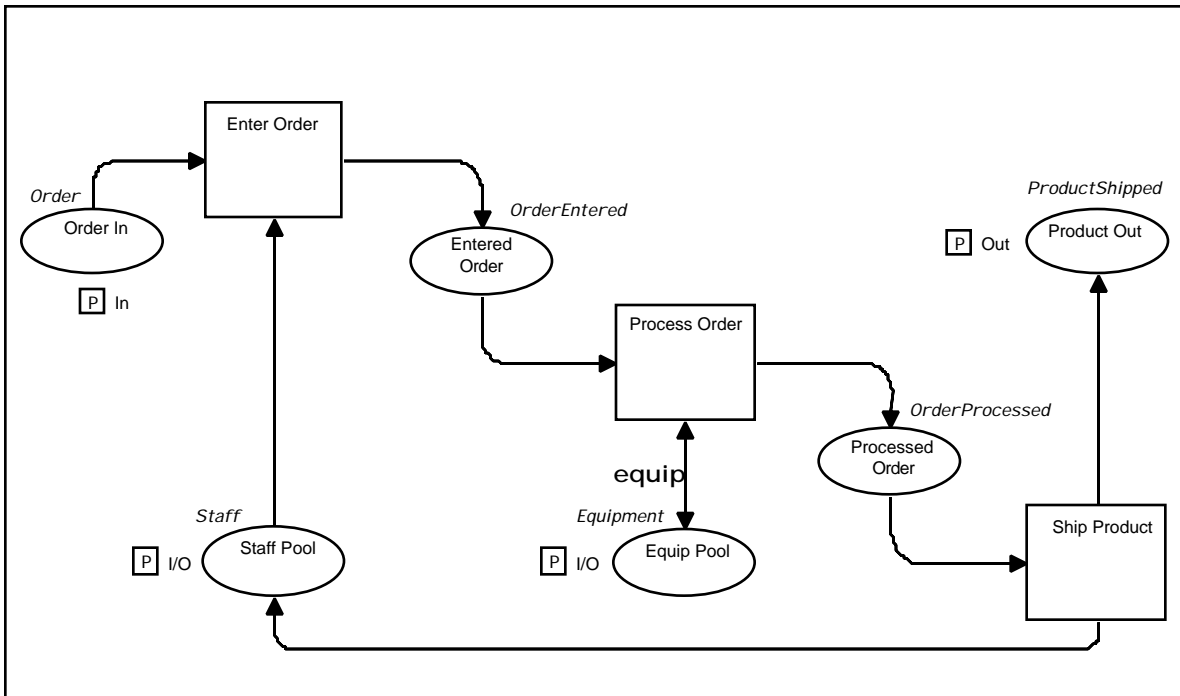
The fact that an arc is bidirectional makes no difference to the process of giving it an inscription.

- Choose **CPN Region** from the **CPN** menu.

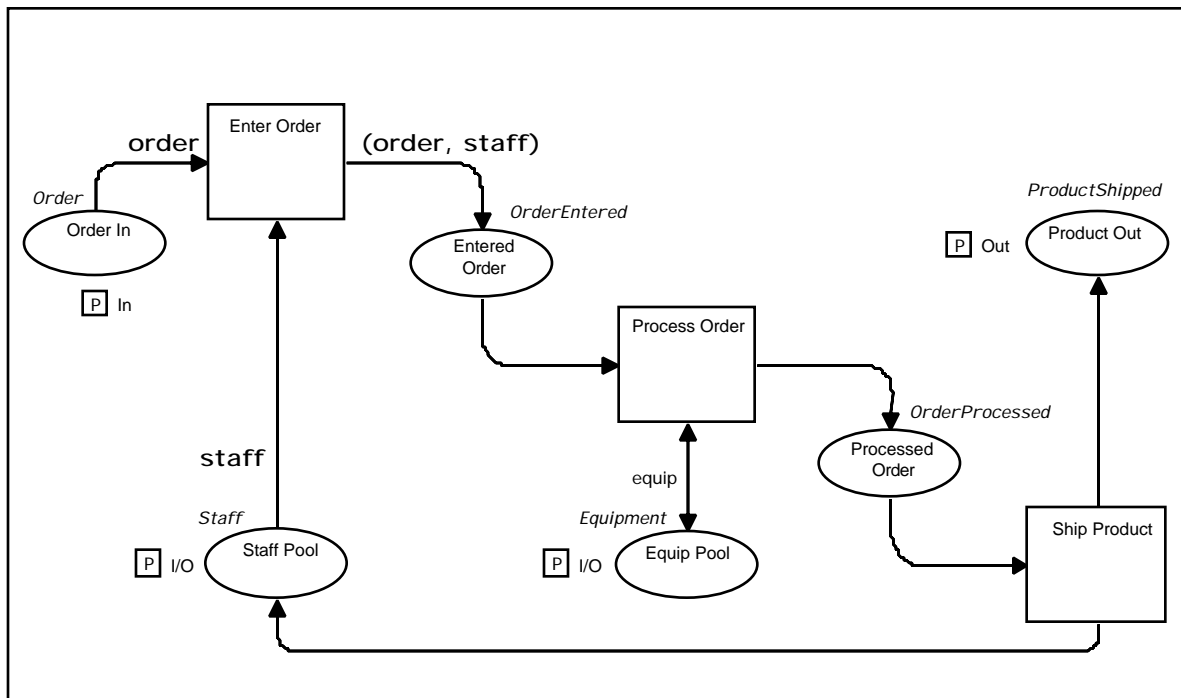
The editor enters arc inscription creation mode; the mouse pointer becomes the region creation tool.

- Click the mouse to the left of the selected arc.
- Give the arc the inscription `equip`.

The net now looks something like this:



- Give the arc between **Staff Pool** and **Enter Order** the inscription *staff*.
- Give the arc between **Order In** and **Enter Order** the inscription *order*.
- Give the arc between **Enter Order** and **Entered Order** the inscription *(order, staff)*:



- Leave arc inscription creation mode.
- Reposition the new arc inscriptions as needed.

## Copying and Pasting Text Regions

All the other arc inscriptions that FirstModel uses are copies of inscriptions that already exist. When a net needs more than one copy of the same region, you don't have to type them in individually. You can copy the region that already exists, and paste it into as many locations (of appropriate type) as you like.

This technique works with all types of CPN text region. Let's use it now to clone some arc inscriptions.

- Select the inscription on the arc between *Order In* and *Enter Order*.
- Execute **Copy** (via the **File** menu or a keystroke shortcut).
- Execute **Paste** (via the **File** menu or a keystroke shortcut).

The editor pastes a copy of the inscription next to the original. This location is only temporary. The mouse pointer becomes the pointer tool, and the status bar displays: Select Arc for next arc inscription region.

- Move the mouse over various arcs.

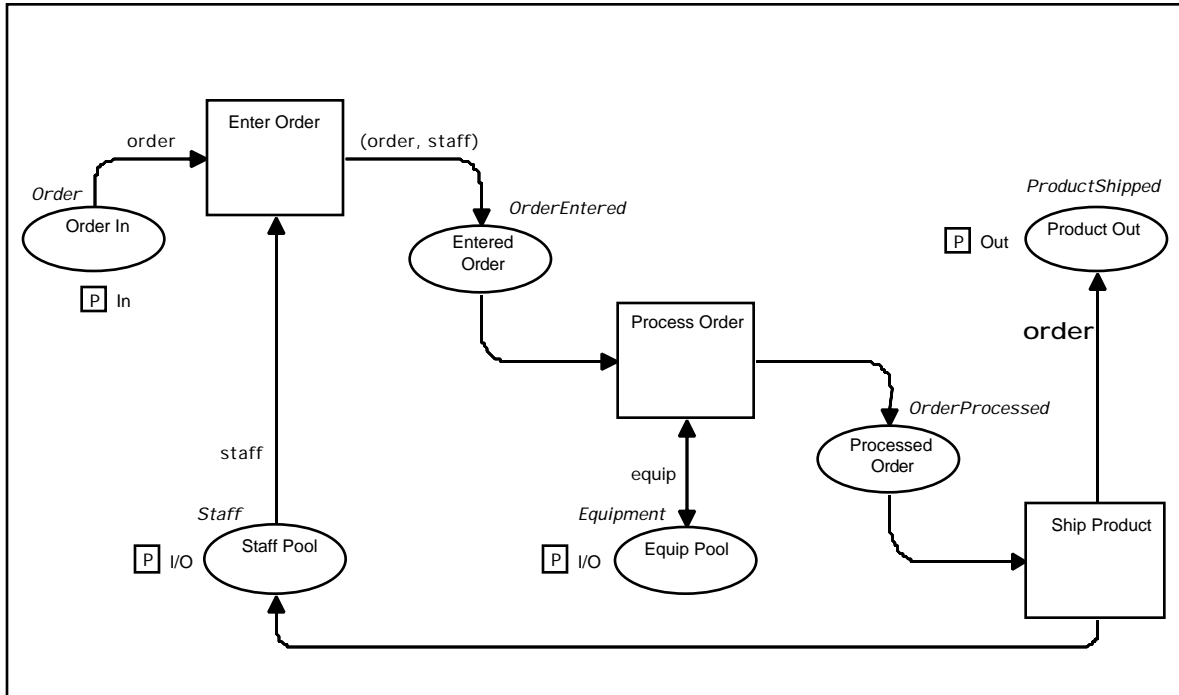


## Design/CPN Tutorial

Whenever the mouse is over an arc, the arc is highlighted. If you click the mouse while an arc is highlighted, the pasted inscription will become the inscription of the arc, and will be moved to a location near it.

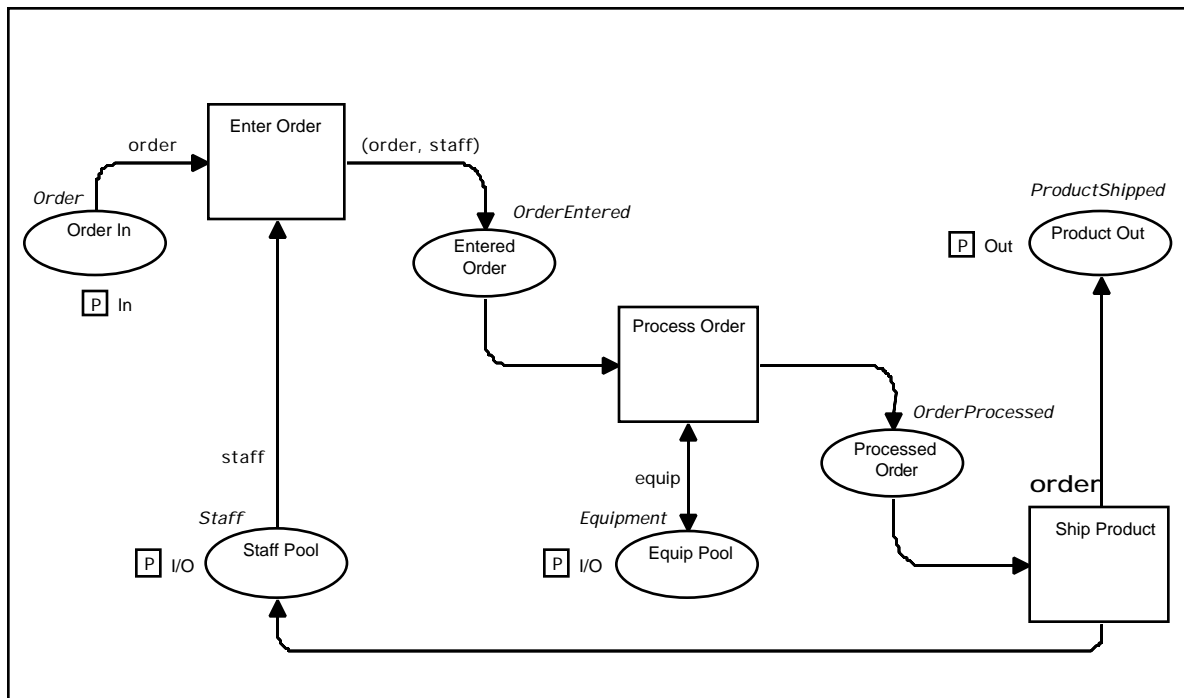
- Click the mouse on the arc between `Ship Product` and `Product Out`.

The copied inscription moves, but not to an ideal location:

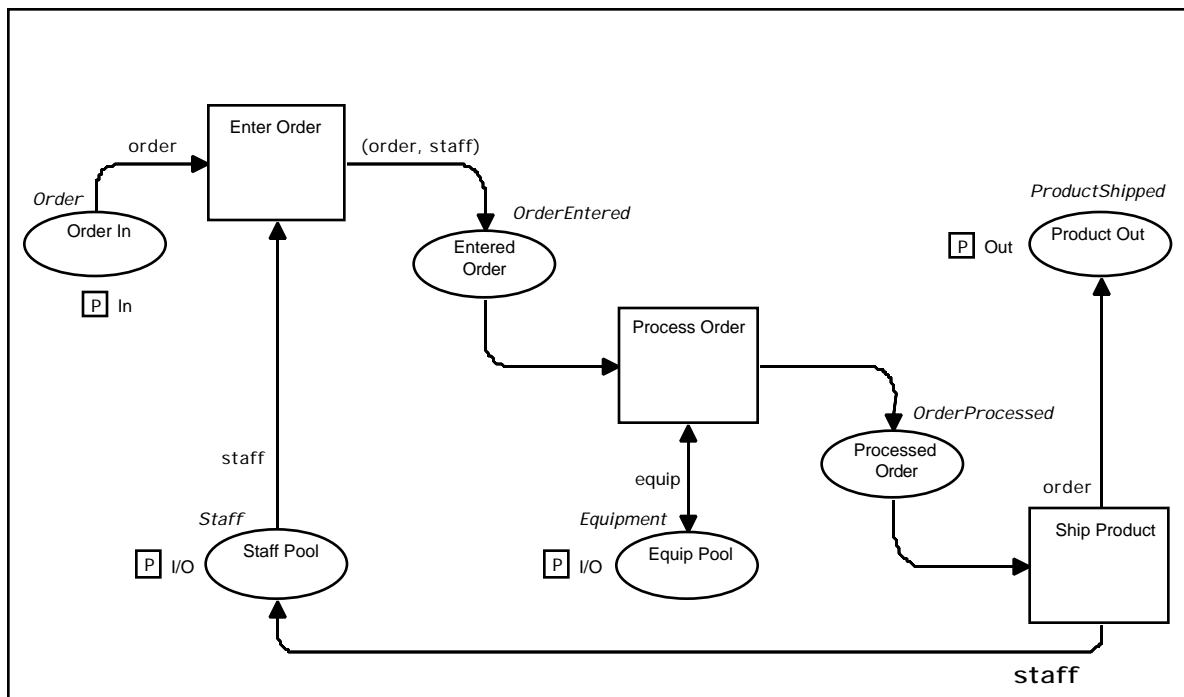


The problem of where to paste a cloned region has no reliable algorithmic solution. Therefore manual adjustment is sometimes necessary, as in this case:

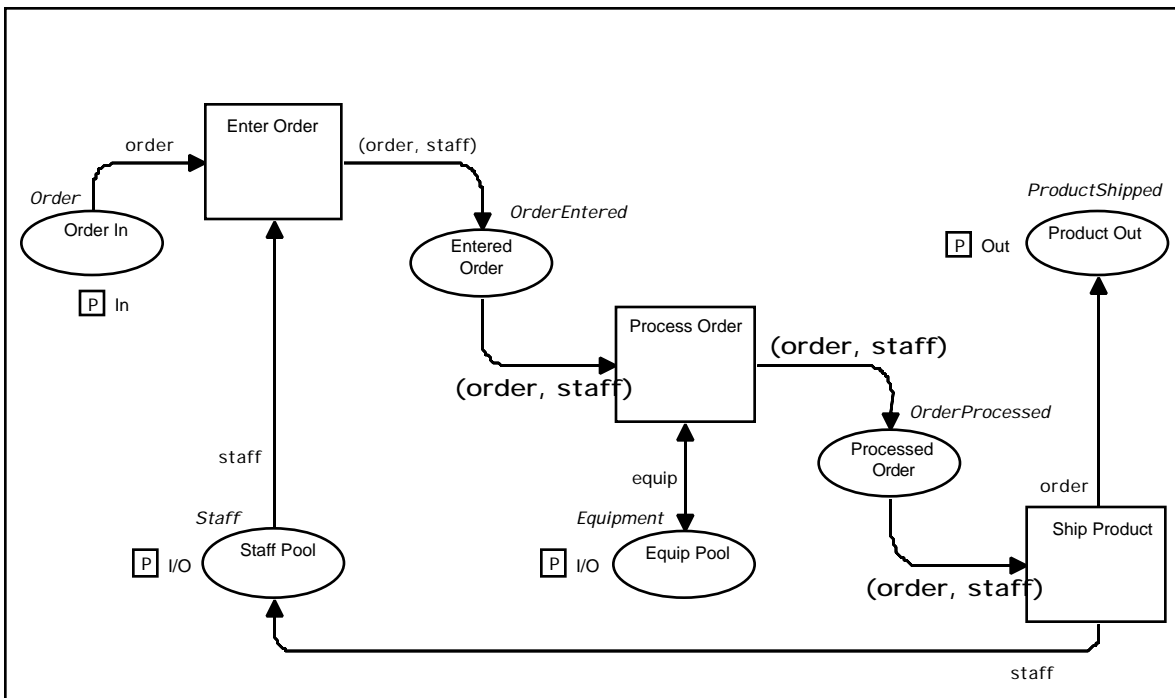
- Move the pasted arc inscription region to the correct position for it:



- Use the same techniques to clone the inscription *staff* onto the arc between *Ship Product* and *Staff Pool*:



- Clone the inscription *(order, staff)* onto all the arcs that use it:



## Creating the Transition Guards

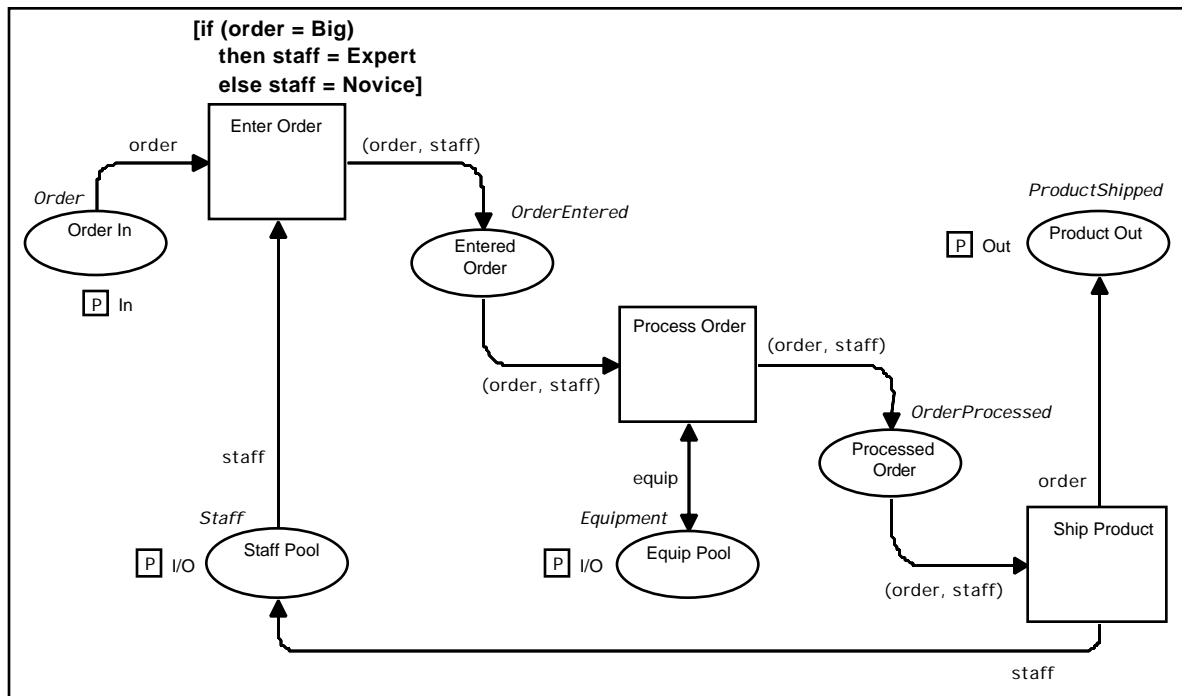
This is the last thing we need to do in order to create the net:

- Select the transition **Enter Order**.
- Choose **CPN Region** from the **CPN** menu.
- Choose **Guard**.

Creating a guard is exactly like creating any other kind of region.

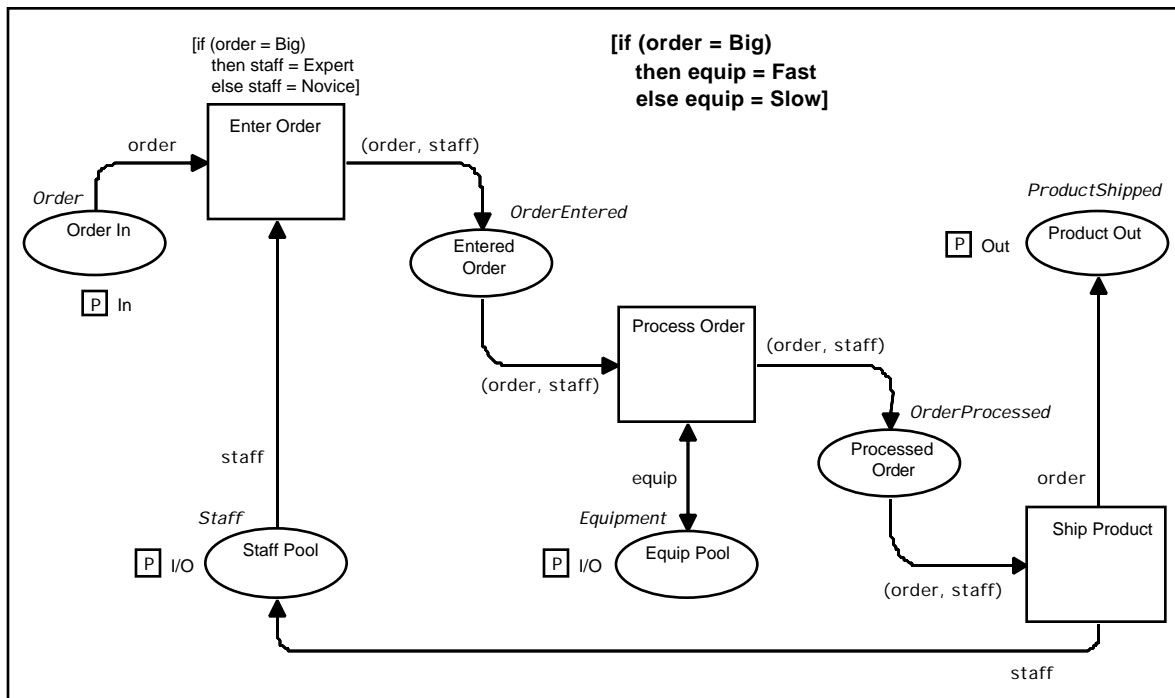
- Give **Enter Order** the guard:

```
[if (order = Big)
  then staff = Expert
  else staff = Novice]
```



- Remain in guard creation mode and click on **Process Order**.
- Give **Process Order** the guard:

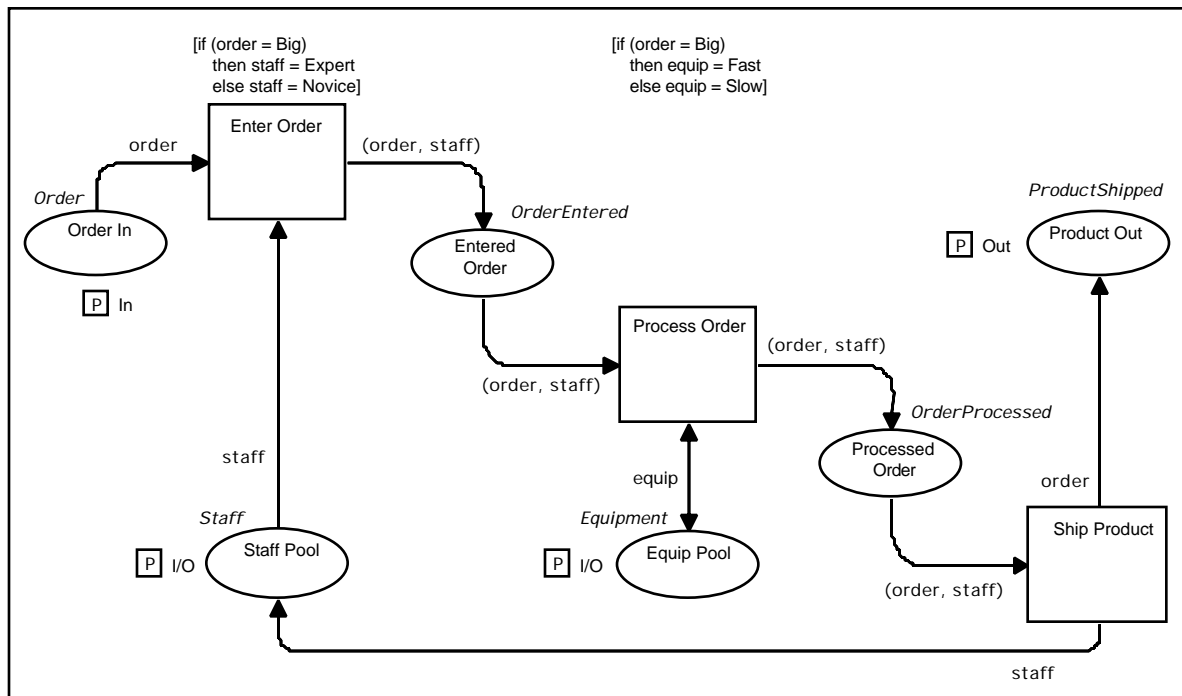
```
[if (order = Big)
  then equip = Fast
  else equip = Slow]
```



- Leave guard creation mode.
- Adjust the guards as needed.

## Final Adjustments to the Net

The final appearance of the net should be about like this:



Adjust the net as needed so that it generally matches this figure. Do anything you like that improves the appearance of the net. Check all the text regions for typographical errors.

You are finished with the construction of FirstModel . . . almost.

## Performing a Syntax Check

A model isn't finished until it is syntactically correct.

- Choose **Syntax Check** from the **CPN** menu.

If you have followed the directions in this chapter carefully, and there are no typos in any text regions, there should be no errors. If any errors are found, track them down using the techniques described in Chapter 9, then redo the syntax check. When no errors remain:

- Save NewFirstModel in NewTTDiagrams.

It's OK to overwrite the existing version of NewFirstModel. We will not need it further, and you can always recreate it from SalesNet.



# Chapter 15

## Executing a Simple Model

In this chapter you will execute FirstModel, the model that you built in Chapter 14. If you experience any problems with the mechanics of net execution, review Chapters 8 and 11 as needed. If you have problems that lead you to believe that your net, though syntactically correct, differs from what this chapter expects it to be, compare it with FirstModelDemo in TutorialDiagrams, and correct any variances.

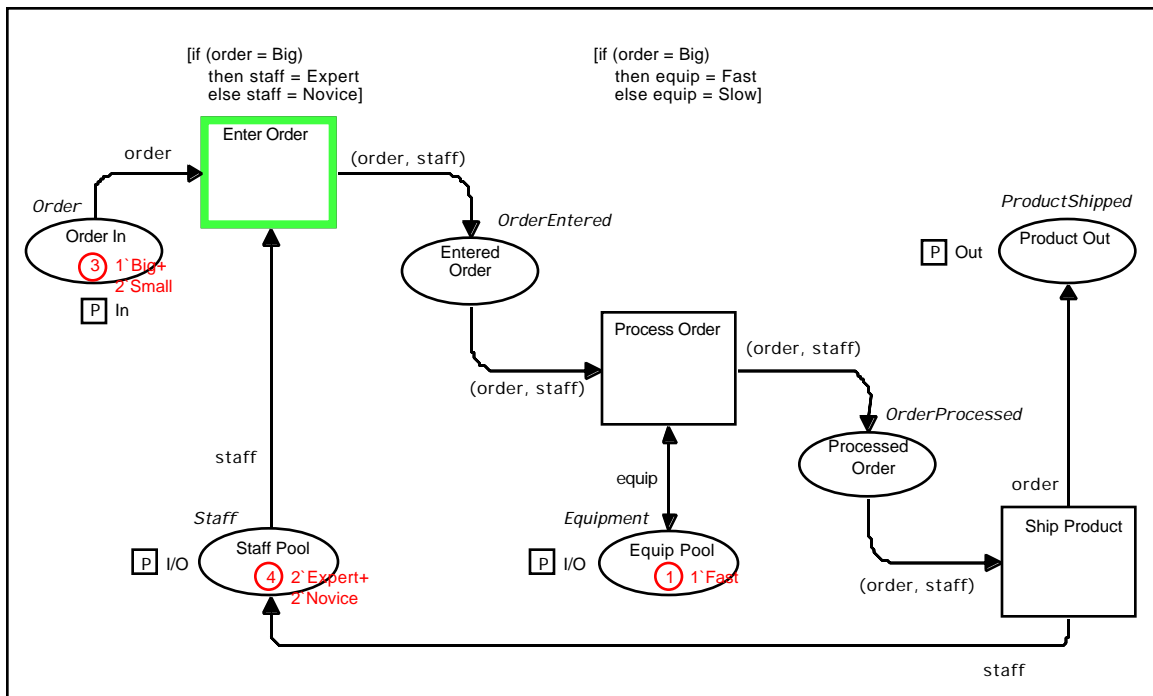
After executing the net in various ways, we'll look at some of topics that relate to net execution generally: studying it, understanding it, controlling it, debugging it, doing it faster, and saving the results of it.

### Executing the Net

- Open NewFirstModel in the NewTTDiagrams directory (if it is not open already).
- Enter the simulator.

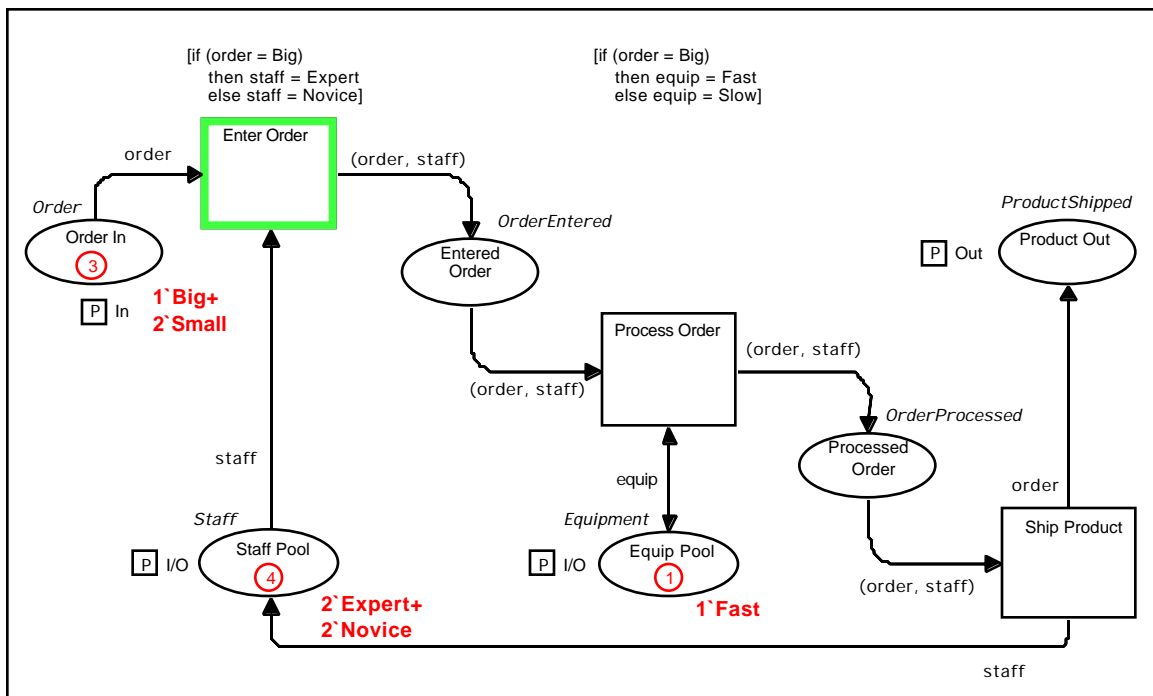
The net should look like this:





As always, marking regions need to be repositioned for best appearance.

- Reposition the marking regions of Order In, Staff Pool, and Equip Pool as shown:



We'll want to look at execution in detail, so:

- Choose **Interactive Simulation Options** from the **Set** menu.
- Set the breakpoints **Beginning of Substep**, **End of Substep**, and **Between Steps**, and the update of graphics **During Substeps**.

Previously when you executed a net, this tutorial showed the changes in the net step by step. This practice was useful in describing exactly what the simulator does when it executes a net, but at this point it would not provide any information you can't get by just looking at the screen, so we will forgo reproducing every simulator action in print. Exact reproduction is not possible anyway, because the simulator makes random choices when conflict exists, so there is often no way to predict exactly what you will see.

- Choose **Interactive Run** from the **Sim** menu.

Execution stops at Breakpoint 1.

- Move and reshape the input and output token regions as needed to produce a good appearance.

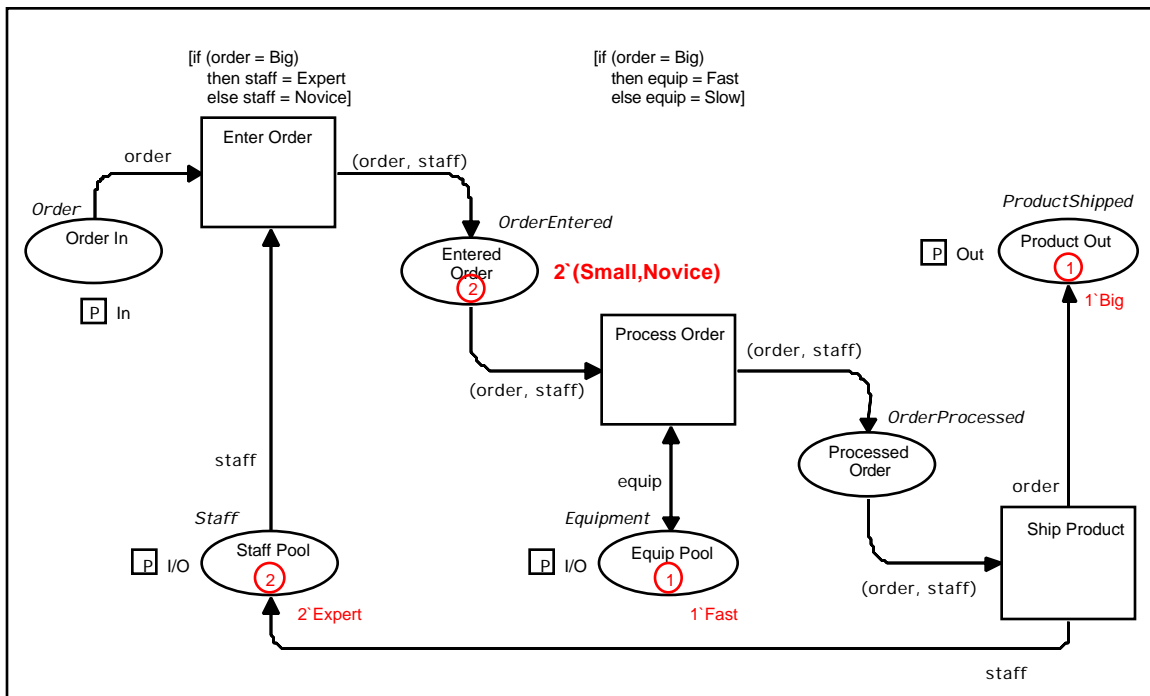
More such regions will appear as tokens propagate through the net. Adjust them as needed at each breakpoint.

- Continue execution until there are no more enabled transitions.

### Analysis of Execution

The things that happened as the net executed should not have presented any surprises. We have already looked at all the essential principles that govern CP net execution. These principles never change; the same sorts of things therefore happen over and over when CP nets execute.

Take a look at the place `Entered Order`. It contains two `Small` tokens, even though net execution is complete.



They have been left stranded because the guard on **Process Order** specifies that a **Small** job must use a **Slow** piece of equipment, and there are no **Slow** pieces of equipment in **Equip Pool**: there are only a single **Fast** pieces of equipment. Consequently there is no enabling binding possible for a **Small** token in **Entered Order**, and the **Small** tokens could not be processed.

Obviously a sales order department should be able to process all types of order, so a model of it should also. Let's change FirstModel so it can do that.

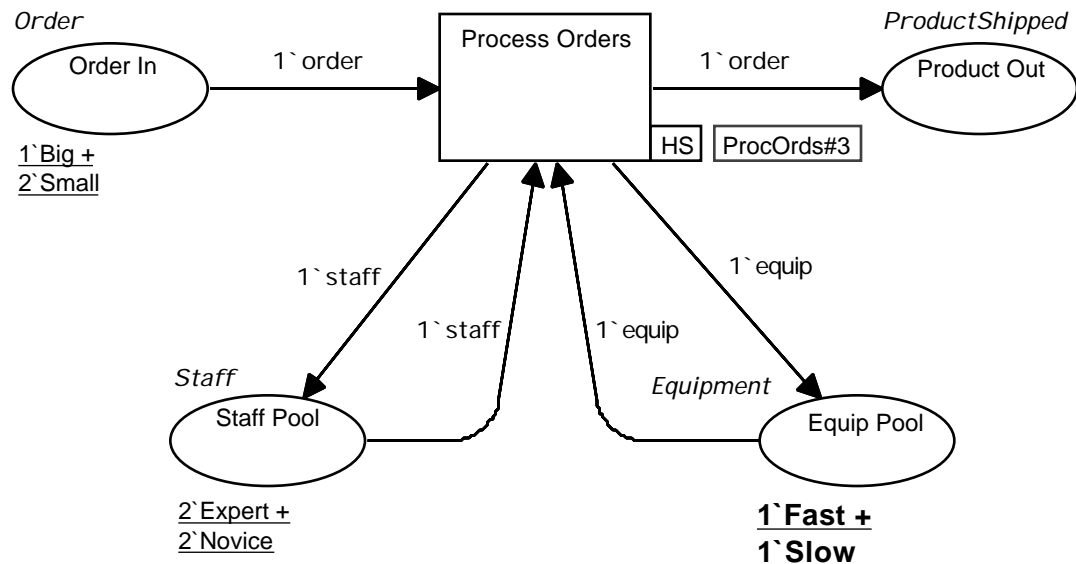
## Subpages and Initial Markings

The way to provide more equipment is to change the initial marking on **Equip Pool**. But that marking is not on the subpage ProcOrds#3; it is on the superpage Sales#1. Why is it not defined on the page where it is actually used?

The reason is that a CP net subpage is not restricted to being the decomposition of a single substitution transition: it can be used repeatedly in a net. Each usage is independent of all other usages. The effect is similar to the use of subroutines in a conventional program. The same subroutine can be used in many different places in a program, and each use is separate from all other uses. A subpage can similarly be used at many different places in a net. See Appendix A for details.

It would obviously be undesirable to wire the values of the arguments to a subroutine into the subroutine. Similarly it would be undesirable to wire the initial markings of the ports on a subpage into the subpage. The subpage then could be used only with those markings, which would greatly restrict its usefulness.

- Open the superpage Sales#1.
- Select the initial marking region of Equip Pool. (You can't enter text mode in the simulator unless you have selected something editable.)
- Change the initial marking of Equip Pool to 1`Fast + 1`Slow:



- Return to page ProcOrds#3.
- Leave text mode. (You can't reswitch in text mode.)
- Choose **Reswitch** from the **Sim** menu.

The changed marking region is compiled into the executable code for the diagram.

- Choose **Initial State** from the **Sim** menu.

The state of the net is initialized, and now reflects the change to the marking region.

- Execute the net.

This time no tokens are left behind, because there is appropriate equipment for any order.

### Experimenting With FirstModel

So far we've just used FirstModel primarily to demonstrate CP net techniques. Let's use it to find out something about the system it models. How would the sales order system respond if there were more jobs than resources, so that the demand on the resources is high, and the compositions of the two resource pools are not well matched to each other? Specifically:

- Open the superpage Sales#1.
- Change Order In's initial marking region to 10`Big + 10`Small.
- Change Staff Pool's initial marking region to 4`Expert + 2`Novice.
- Change Equip Pool's initial marking region to 2`Fast + 4`Slow.
- Leave text mode.

### How to Do Experiments

There is no way for this tutorial to anticipate how much or how little you will want to see of a particular experiment. In general you should continue executing and/or re-executing until you see the general pattern of execution, and then cancel execution.

You may find that you don't always want to examine the net at all of the breakpoints that are set, or even at any of them. If the instructions specifically say to set or clear some breakpoint, be sure to do it, as there will be some specific point to be observed. But when the task is just to execute a net and see what can be learned, you should feel free to set and clear breakpoints as needed to produce the level of observability you want.

If you find that net execution has run away because you have no breakpoints set, press ESC. When the current step is complete (nothing will seem to happen until then) the simulator will offer you a chance to cancel execution.

With these comments in mind:

- Return to the subpage, execute **Reswitch** and **Initial State**, then execute the net.

### Analysis of Execution

This is the first time we have executed a model under heavy load. There is not enough staff or equipment to process all the orders, so orders wait in `Order In` for staff members, and in `Entered Order` for equipment. Due to the overload, some of the orders must wait through many steps before their turn comes, but eventually all get through.

The asymmetry in resource availability produces a corresponding asymmetry in waiting behavior. `Small` jobs tend to remain in `Order In` through more steps than `Big` jobs do, because there are more `Fast` than `Slow` staff members, but `Big` jobs tend to remain in `Entered Order` through more steps than `Small` jobs do, because there are more `Slow` than `Fast` equipment pieces.

The heavy load, and the increased `Staff` and `Equipment` pools, also made obvious the concurrency in `FirstModel`. During execution there was usually concurrent activity of transitions both with themselves and with other transitions. Many things were happening in parallel, with all the details handled automatically by the simulator. All we had to do is watch.

### Complicating FirstModel

The behavior shown by `FirstModel` in the previous experiment consisted of many operations, but its overall wasn't really that surprising. You could probably have predicted about what would happen when you executed the model, and quite possibly you did.

One reason that `FirstModel` is so easy to predict is that the realms of `Fast` and `Slow` jobs have no effect on each other. From a modeling viewpoint, there is really no need to have them both. Anything learned by running `FirstModel` with some distribution of `Big`, `Expert`, and `Fast` tokens, and another of `Small`, `Novice`, and `Slow` tokens, could just as well have been learned by using only `Big`, `Expert`, and `Fast` tokens, and running the model twice, once with one distribution and once with the other.

Let's change `FirstModel` so that its behavior isn't so obvious. Let's say that a `Big` job must have an `Expert` staff member and a `Fast` piece of equipment, while a `Small` job can have any available staff member and any available piece of equipment. This will cause the two types of order to compete and interfere with each other in such a way that system performance becomes a highly nonlinear function of the exact compositions of the set of orders to be processed and of the staff and equipment pools.

### Using a Guard to Create a Partial Constraint

The simplest way to implement the new rules is to change the guards on `Enter Order` and `Process Order`.

The guards we have used so far have been very simple. Their only effect has been to predicate enablement on the existence of tokens with fixed values or combinations of values. This is an extremely common use of guards, but it barely scratches the surface of their capabilities.

The definition of a guard is very general. It is a boolean expression that must evaluate to **true** in order for a transition to be enabled. Nothing requires a guard to constrain tokens to fixed values. Nothing requires a guard to constrain tokens at all! A guard can be anything we want it to be. If it evaluates to **true**, the transition can be enabled; if it evaluates to **false**, the transition cannot be enabled. The rest is up to us.

What we want now is a guard that will enforce a constraint in some cases, and no constraint in others. The requisite guard has a **then** clause that requires a particular binding, and an **else** clause that is always **true** irrespective of any binding. Specifically:

- Change `Enter Order`'s guard to be:

```
[if (order = Big)
  then staff = Expert
  else true]
```

- Change `Process Order`'s guard to be:

```
[if (order = Big)
  then equip = Fast
  else true]
```

- Leave text mode.
- Execute **Reswitch** and **Initial State**.

### Executing the Net

Would you care to predict how this net will execute? It is almost impossible. It is also unnecessary. The whole purpose of modeling and simulation is to help us to deal with systems whose behavior cannot be predicted by the unaided human mind, but can be computed if the structure of the system can somehow be represented as an executable model.

- Execute the net.

### Analysis of Execution

What did you see when the net executed? How exactly did it perform? A lot of things went on, but what was the overall trend?

The fact is, you probably aren't sure. Many things happened, but there was no way to see what the overall pattern was. No doubt the execution created much information about system behavior and performance, but that information is not necessarily obvious from watching the flow of tokens.

Observing the individual events that constitute the execution of a model, as we have been doing, is not the best way to gather information about behavior. It is useful primarily for developing and debugging a net. Let's look at some of the tools that Design/CPN provides to facilitate this process. Then we'll go the other way, and look methods for deriving information from a model without watching any tokens at all.

### Controlling the Appearance of Concurrency

One thing that made it difficult to see what FirstModel does is the fact that it does so many things at once. Concurrency, as Chapter 11 pointed out, is a distinguishing feature of Petri nets. But concurrency can get in the way when the need is for a very detailed and localized view of exactly what a net is doing. This is particularly true when a net is being debugged. When dozens of transitions are concurrently enabled, each with dozens of enabling bindings, so much can happen in one execution step that debugging is almost impossible.

Therefore Design/CPN provides a way to control how concurrency is presented to the observer. By setting some parameters, you can control how much, or how little, is done in each step of the simulator's execution algorithm. Such control has no effect on the meaning of the net: it has the same structural and behavioral properties no matter how its execution is made to appear.

By controlling the appearance of concurrency you can in effect look at net execution through a microscope. You can see each individual microevent of net execution, without interference from other things that are happening concurrently. This is exactly what is needed when it is unclear exactly what a net does, and/or why it is not doing what it is supposed to do.

The appearance of concurrency is controlled by tuning the algorithm that Design/CPN uses to construct occurrence sets. Let's see how this works, then use it to look at FirstModel's execution from various perspectives.



### Review of Occurrence Sets

In Chapter 11, we looked in detail at the simulator's execution algorithm. Part of that algorithm is the construction of an occurrence set. An *occurrence set* is a list of elements called *binding elements*, each of which specifies a binding for a particular transition.

The simulator changes the state of a net by executing the elements in an occurrence set. Executing a binding element consists of rebinding the arc inscription variables of the transition that the element indicates to the values in the binding that the element indicates, and then firing the transition.

By definition, the elements in an occurrence set are not in conflict: the transition firings that the set indicates can all occur in the same step without attempting to subtract more tokens from any input place than currently exist in the place. That is, the firings can occur concurrently.

### Constructing an Occurrence Set

When it constructs an occurrence set, the simulator has many decisions to make. Even SalesNet, which has only one transition, provided the possibility of several different occurrence sets. When there are many transitions with many bindings that conflict in many ways, the number of possible occurrence sets may be very large.

There is no one best algorithm for constructing an occurrence set. One algorithm might result in very fast execution, but obscure the details of net execution by doing too many things at once. Another might execute slowly but illuminate every detail. A third might accomplish both these ends very well, and a fourth very poorly. An algorithm that gives the desired results with one net might be ineffective with another.

Therefore CP nets do not require that occurrence sets be constructed in any particular way, or have any particular property other than the nonexistence of conflict among the constituent binding elements. In particular, nothing requires the simulator to construct the largest possible occurrence set, or one of the largest possible. If we want to look at very small increments in the behavior of a model, we might want very small occurrence sets.

The obvious question is: how can we be sure that changing the occurrence set algorithm will not change the meaning of the net itself? How can we be sure that the behavior of a model is invariant of the way in which occurrence sets are constructed? To answer this question, we need a precise definition of concurrency.

### What Is Concurrency?

Informally, concurrent activities are those that happen “at the same time”. But what exactly does that mean? If we are to have a useful modeling paradigm, it cannot mean “in the same instant,” because no real activity is instantaneous; there is always some duration. Nor can it mean “occurring during exactly the same interval,” because rarely if ever could this property be guaranteed for different activities at different physical locations.

CP nets define concurrency as follows: A collection of activities is *concurrent* if the result of their occurrence is unaffected by the presence or even the existence of any particular occurrence order. That is, the defining property of concurrency is not that activities *do* overlap in time, but only that they *can*. It makes no difference whether they actually do or not, and if they do, it makes no difference just how they overlap. When all are complete, the result will be the same regardless of all such details.

As a result of this definition, we have complete freedom to construct occurrence sets in any way we like. We know that all the events we might put into an occurrence set can happen concurrently, because they are guaranteed not to be in conflict. Therefore, by the definition of concurrency, no ordering of events that results from this or that choice of occurrence sets makes any difference, because the ordering does not matter at all.

Similarly it makes no difference whether we execute a set of concurrent events by creating a single occurrence set or several of them. At the extreme we could require that every occurrence set consist of just one binding element, and it would make no difference: the effect would be the same as if we concatenated binding elements in all the sets into a single set, and then executed that set; or into several sets; or in different orders into one or several sets. It makes no difference at all to the outcome.

### Occurrence Set Parameters

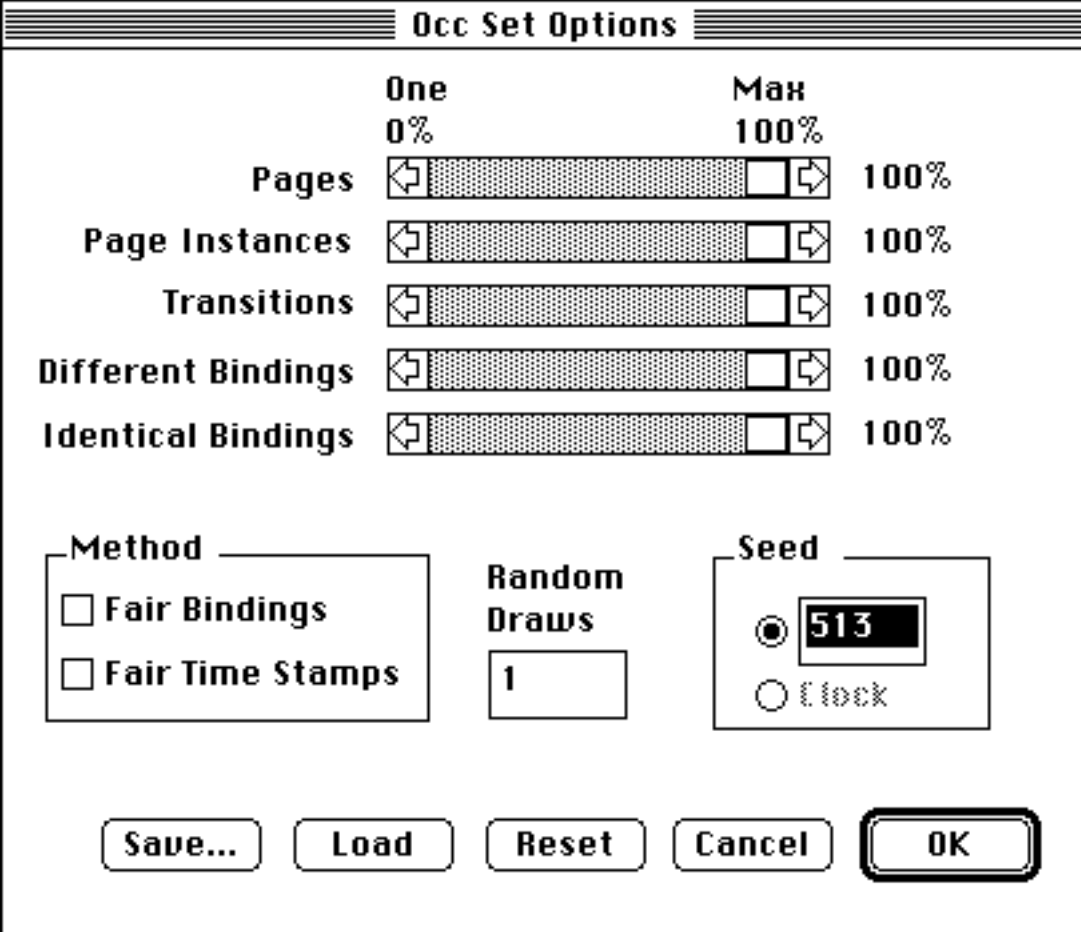
Most features of the simulator's execution algorithm are determined by the rules of CP net dynamics (Chapter 7), and so cannot be changed. But the algorithm by which the simulator constructs occurrence sets can be anything that is useful, as we have seen. Since different algorithms are better with different nets and for different purposes, Design/CPN allows you to specify various features of the algorithm that the simulator uses. This is done by setting parameters called *occurrence set parameters*.

The occurrence set parameters specified in the diagram NewFirstModel tell Design/CPN to create occurrence sets that are as large as possible. That is why so much happens at once when

FirstModel executes. Let's take a look at these parameters, and see what each of them means.

- Choose **Occurrence Set Options** from the **Set** menu.

The **Occurrence Set Options** dialog appears:



The **Occ Set Options** dialog box is shown. It features a title bar with the text "Occ Set Options". Inside, there are five rows of settings, each with a label, a slider bar, and a percentage value. The labels are **Pages**, **Page Instances**, **Transitions**, **Different Bindings**, and **Identical Bindings**. The slider bars are all set to 100%. Above the first slider, there are labels "One" and "Max" with values "0%" and "100%" respectively. Below the sliders, there are three sections: "Method" with two checkboxes, "Random Draws" with a text box, and "Seed" with two radio buttons. The "Method" section has checkboxes for "Fair Bindings" and "Fair Time Stamps". The "Random Draws" section has a text box containing the value "1". The "Seed" section has radio buttons for "513" (selected) and "Clock". At the bottom, there are five buttons: "Save...", "Load", "Reset", "Cancel", and "OK".

	One	Max
<b>Pages</b>	0%	100%
<b>Page Instances</b>		
<b>Transitions</b>		
<b>Different Bindings</b>		
<b>Identical Bindings</b>		

**Method**

- ☐ Fair Bindings
- ☐ Fair Time Stamps

**Random Draws**

1

**Seed**

- ☒ 513
- ☐ Clock

**Buttons:** Save... Load Reset Cancel OK

Under X-Windows, the slider bars shown in the above figure are replaced by edit boxes, but the dialog is otherwise similar.

The settings for **Pages** and **Page Instances** apply only when there are enabled transitions on more than one page. The model you are now working with has only one page with enabled transitions, so we can ignore these parameters for the present and consider only the settings for **Transitions**, **Different Bindings**, and **Identical Bindings**.

### Transitions

The setting for **Transitions** determines how the simulator will construct occurrence sets when more than one transition is simultaneously enabled on a page.

When the setting is 100%, all enabled transitions will be represented in the set, subject to the restriction that no occurrence set will be constructed that contains conflicting binding elements.

Settings between 1% and 99% define the probability that a particular enabled transition will be represented in the occurrence set. For example, at a setting of 50%, each candidate transition has a 50% chance of being represented. Note that setting a value of 50% does *not* mean that 50% of the candidates will be represented: more or less might be, depending on chance and the requirement to avoid conflicting binding elements in a set.

There would be no purpose in constructing an empty occurrence set. Therefore at least one transition will be represented in the set no matter what the outcome of any random choices. When the setting is 0%, exactly one enabled transition will be represented in the set.

### Different Bindings

The setting for **Different Bindings** determines how the simulator will construct occurrence sets when a given transition is enabled with two or more nonidentical bindings.

When the setting is 100%, all enabling bindings will be represented in the set, subject to the restriction that no occurrence set will be constructed that contains conflicting binding elements.

Settings between 1% and 99% define the probability that a particular binding will be represented in the occurrence set.

Subject to the restriction on conflicting binding elements, at least one binding will be represented in the set no matter what the outcome of any random choices. When the setting is 0%, exactly one of the bindings will be represented.

### Identical Bindings

The setting for **Identical Bindings** determines how the simulator will construct occurrence sets when a transition is enabled with two or more bindings that are identical, i.e. consist of the exactly the same values.

Subject to the restriction on conflicting binding elements: when the setting is 100%, all the bindings will be represented in the occur-

rence set; when it is 0%, exactly one will be; and intermediate settings give proportionate intermediate results.

### Scope of Occurrence Set Parameters

Each of the five parameter settings works within the bounds established by those higher in the list. Thus **Transitions** determines how many enabled transitions (if there is more than one) will be represented in an occurrence set; **Different Bindings** determines how many different bindings (if there is more than one) will be included for each represented transition; and **Identical Bindings** determines how many copies (if there is more than one) will be included for each represented binding.

### Setting Occurrence Set Parameters

To set any of the occurrence set parameters, edit the number in the box to the right of the name of the particular parameter. The value must be between 0 and 100.

- Modify some parameters by editing their values.
- When you are done, restore all of the settings to 100%

## Experimenting With Net Execution

You now have the information you need in order to execute FirstModel with different settings for the occurrence set parameters. Let's start with an extreme case:

- **Set Transitions, Different Bindings and Identical Bindings** to 0%
- Execute the net.

The simulator now fires just one transition with one binding in each step, because the current occurrence set parameters specify one-element occurrence sets. This does NOT mean that things that previously happened concurrently now happen sequentially. The situation with respect to concurrency, as defined above, has not changed at all. We have enforced an order on net execution by specifying very small occurrence sets, but since order is irrelevant to concurrent activities, the imposition changes nothing fundamental: it is only a change of appearance.

- Perform additional experiments, using various settings for the occurrence set parameters, until you feel thoroughly at home with setting and using them.

Nothing you can do with occurrence set parameters makes any fundamental difference. But an appropriate choice of parameters can be very useful in studying and debugging a net. Deciding what parameter choices will probably produce an interesting and useful view of net execution is part of the art of modeling.

- Change `FirstModel`'s initial markings to something you think will give very different results.
- Execute the net with the parameters set to 100%, and then at other settings that you think will help to reveal the details of what it is doing.
- Leave the parameters set at 100%

## Faster Model Execution

The emphasis in this tutorial so far has been on creating nets and watching every detail of their execution. There have been two reasons for this emphasis:

1. It is appropriate for beginning the process of learning how to model and simulate.
2. A model is not useful until it has been developed and observed enough to justify some confidence that it correctly reflects the relevant parts of the modeled system.

Once a model has been judged to accurately reflect its system, emphasis generally shifts from watching the details of its execution to evaluating the overall performance of the system. Such evaluation typically requires a model to process a great deal of data. This requirement makes it desirable for the model to run as fast as possible.

Of course, such performance evaluation often reveals information not otherwise derivable that motivates additional changes to the model; modeling is at every stage an iterative process, and one can never be sure that the last iteration has been accomplished.

In order to facilitate model development and observation while providing a capability for very fast execution, the simulator provides two execution environments: interactive mode and automatic mode.

## Interactive Mode

The execution mode you have been using so far is called *Interactive mode*. The purpose of this mode is to facilitate interactive observation and control of an executing net.

In interactive mode the simulator maintains, manipulates, and displays much information that is not necessary for net execution, and exists only to provide an interface to humans. Interactive mode is designed for use with techniques like setting breakpoints and tuning occurrence set parameters, which can provide a very detailed view of individual events.

### Automatic Mode

The overhead of providing an interactive interface to net execution results in an execution speed that is far too slow to allow efficient evaluation of a model's performance during prolonged execution. Therefore the simulator provides a very fast execution mode called *automatic mode*. This mode is optimized for speed rather than for the observation of individual execution events. During execution in automatic mode, changes to a net's state are not displayed on the screen.

Automatic mode is intended for use with techniques such as graphical animation or the statistical variables and charts facility, which provide a high-level view of model performance but ignore individual details.

### Fair and Fast Execution

There are actually two kinds of automatic mode: *fair automatic* and *fast automatic*. In fair automatic mode, all possible enabling bindings are equally likely to be used. In fast automatic mode, possible enabling bindings are used in the order in which the simulator discovers them.

The tradeoff between fair and fast exists because insuring the fair selection of bindings requires additional processing, which slows down execution, while fast selection can result in very skewed net executions: some bindings may be used over and over, while others are never used. Such execution is not formally incorrect, but it may not represent what would happen in the system that the net represents.

Interactive execution is always fair. A fast interactive mode could exist, but it would serve no purpose: interactive mode slows execution so much that the additional overhead of insuring fair behavior is imperceptible.

## Selecting the Execution Mode

Some nets will be executed only interactively, and others only automatically; some will be executed both ways at different stages of

their development. Of those that are executed automatically, some will be executed only in fair mode, others only in fast mode, and some will be executed both ways as they develop.

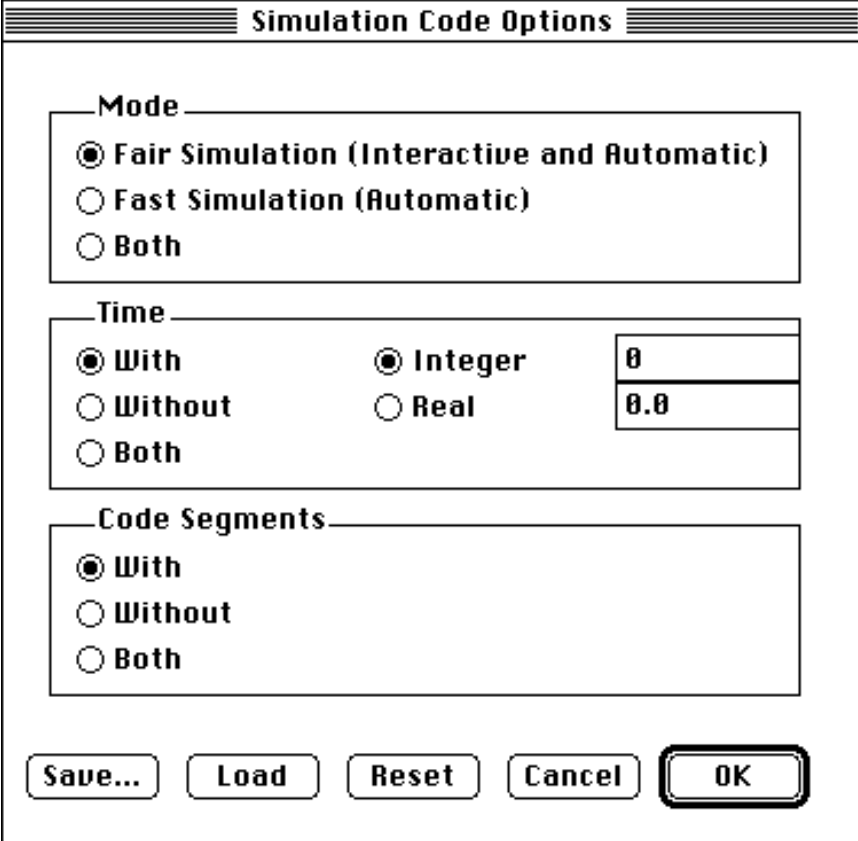
Design/CPN could compile every net to permit every type of execution, but this would result in the generation of much unnecessary code for a net that will be executed in only one way. The generation of this code would needlessly slow down the process of switching the net into the simulator.

Therefore, Design/CPN allows you to specify a net's intended execution mode(s) before you enter the simulator. This specification can be changed in the simulator, but the net will then have to be reswitched. When more than one execution mode will be needed, Design/CPN allows you to specify the desired possibilities, then choose among them on a per-run basis.

### Specifying Possible Execution Modes

To specify possible execution modes, use the **Simulation Code Options** command.

- Choose **Simulation Code Options** from the **Set** menu:



The **Simulation Code Options** dialog box is used to configure simulation parameters. It contains three main sections: **Mode**, **Time**, and **Code Segments**. The **Mode** section has three radio buttons: **Fair Simulation (Interactive and Automatic)** (selected), **Fast Simulation (Automatic)**, and **Both**. The **Time** section has two columns of radio buttons. The first column has **With** (selected), **Without**, and **Both**. The second column has **Integer** (selected) and **Real**. To the right of these are two input fields: the top one contains **0** and the bottom one contains **0.0**. The **Code Segments** section has three radio buttons: **With** (selected), **Without**, and **Both**. At the bottom of the dialog are five buttons: **Save...**, **Load**, **Reset**, **Cancel**, and **OK** (which is highlighted with a double border).



The options in the **Mode** section control the possible execution modes. **Fair Simulation** provides for either interactive simulation (which is always fair), or fair automatic simulation. **Fast Simulation** provides for fast automatic simulation only. **Both** makes all three simulation modes available.

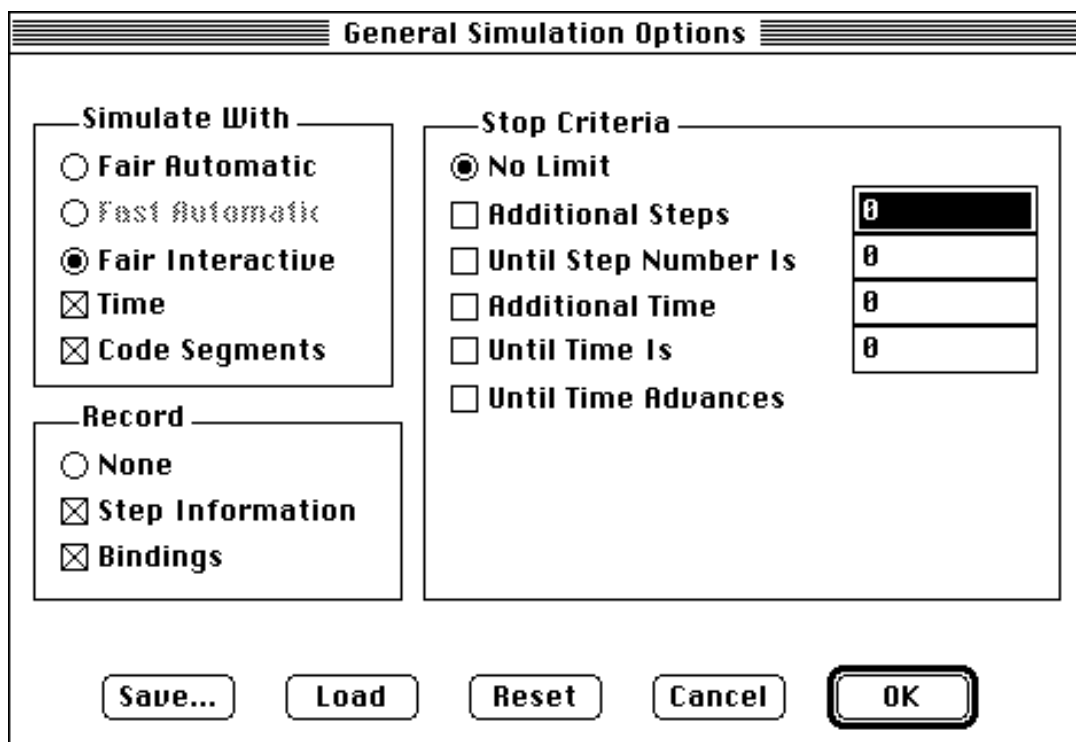
As the **Mode** section of the dialog shows, the net was compiled to provide a choice of interactive or fast automatic simulation when you took it into the simulator. There is no need to change this option now, so:

- Click **Cancel**.

### Specifying the Actual Execution Mode

To specify the execution mode that will actually be used, use the **General Simulation Options** command.

- Choose **General Simulation Options** from the **Set** menu:



The dialog box titled "General Simulation Options" contains several sections for configuring simulation parameters. The "Simulate With" section on the left has three radio buttons: "Fair Automatic" (disabled), "Fast Automatic" (disabled), and "Fair Interactive" (selected). Below these are two checked checkboxes: "Time" and "Code Segments". The "Record" section below it has three options: "None" (disabled), "Step Information" (checked), and "Bindings" (checked). The "Stop Criteria" section on the right has a "No Limit" radio button selected, followed by four unchecked checkboxes: "Additional Steps", "Until Step Number Is", "Additional Time", and "Until Time Is". To the right of these checkboxes are four input fields, each containing the value "0". At the bottom of the dialog are five buttons: "Save...", "Load", "Reset", "Cancel", and "OK".

The **Simulate With** section determines which of the possible execution modes will actually be used when the net executes. Modes that are not possible, because the **Simulation Code Options** settings do not allow for them, are grayed out, as **Fast Automatic** currently is.

The currently selected option is **Fair Interactive**, which is why all net execution so far has been interactive. Let's change this to specify fair automatic execution.

- Select **Fair Automatic** in the **Simulate With** section of the dialog.

Leave the dialog open for use in the next section.

## Specifying Stop Criteria

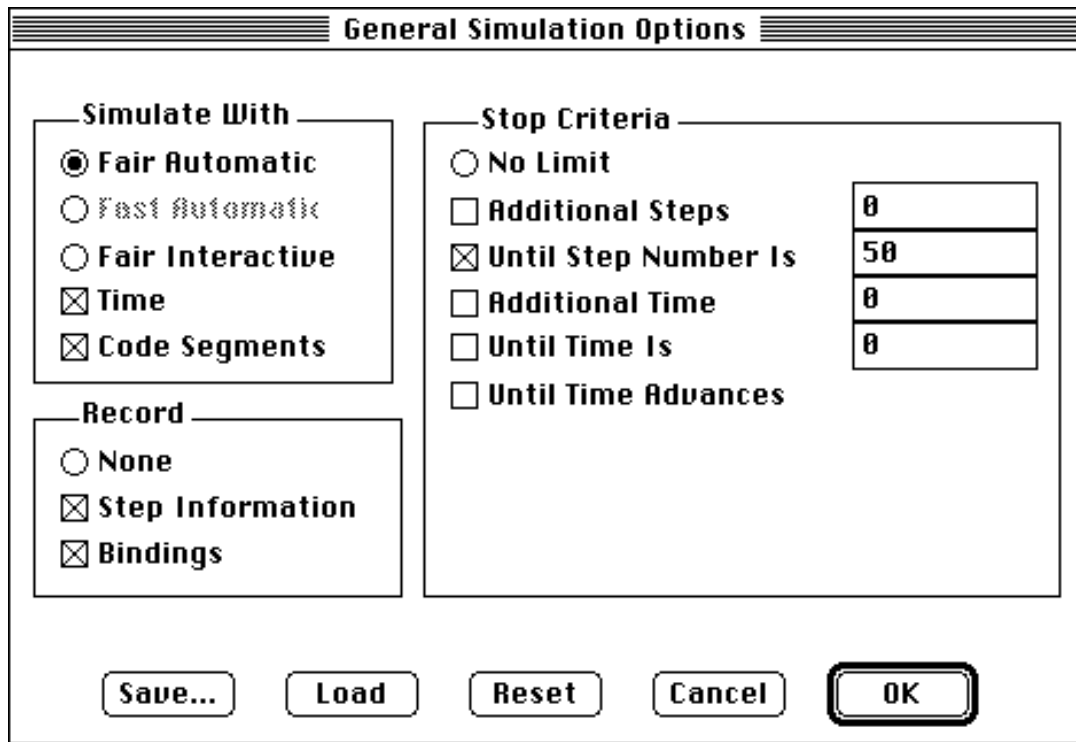
When you use interactive mode, you can set breakpoints to stop net execution. If no breakpoints are set, a net will run until there are no more enabled transitions. Some nets never run out of enabled transitions. Unless something stops them, they will run indefinitely.

Breakpoints provide only the simplest control over interactive execution, and have no effect during automatic execution. A more general method is therefore needed for stopping net execution. The **Stop Criteria** section of the **General Simulation Options** dialog provides this capability. **Stop Criteria** options apply during both interactive and automatic execution. They may be changed whenever a net is not executing.

The currently selected option is **No Limit**, meaning that a net will execute until it reaches a breakpoint in interactive mode, or runs out of enabled transitions (which may never happen) in any mode. Let's set the net to stop executing when Step 50 is complete.

- Select the **Until Step Number Is** option.
- Edit the number to the left of the option to be 50.

The dialog should now look like this:



The dialog box is titled "General Simulation Options". It contains three main sections: "Simulate With", "Record", and "Stop Criteria".

- Simulate With:**
  - ☒ Fair Automatic
  - ☐ Fast Automatic
  - ☐ Fair Interactive
  - ☒ Time
  - ☒ Code Segments
- Record:**
  - ☐ None
  - ☒ Step Information
  - ☒ Bindings
- Stop Criteria:**
  - ☐ No Limit
  - ☐ Additional Steps: 0
  - ☒ Until Step Number Is: 50
  - ☐ Additional Time: 0
  - ☐ Until Time Is: 0
  - ☐ Until Time Advances

At the bottom, there are five buttons: "Save...", "Load", "Reset", "Cancel", and "OK". The "OK" button is highlighted with a double border.

- Click **OK**.

## Automatic Net Execution

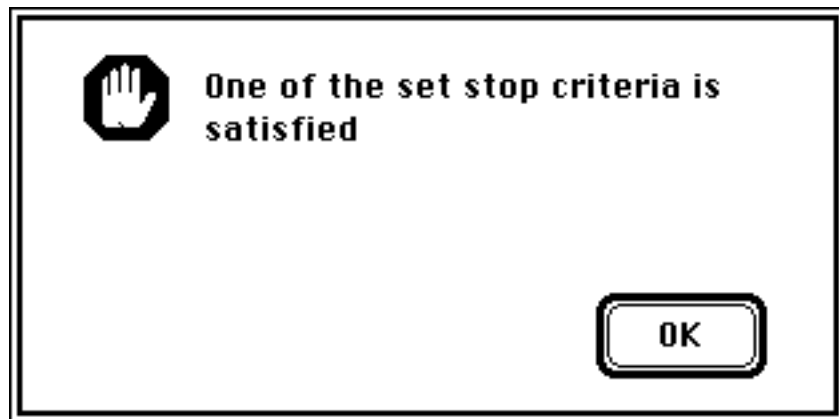
Let's give automatic net execution a try. First let's provide plenty for the net to do.

- Change Order In's initial marking to 100`Big + 100`Small.
- Change Staff Pool's marking to 10`Expert + 5`Novice.
- Change Equipment Pool's initial marking to 10`Fast + 5`Slow.
- Leave text mode; execute **Reswitch** and **Initial State**.

The net is now ready to execute:

- Choose **Automatic Run** from the **Sim** menu.

After only a second or two, a dialog appears:



- Click **OK**.

The simulator updates the appearance of the net to indicate its new state. The status bar shows that execution is stopped at Step 50. The 50 steps took much less time than even a single step in interactive mode. In fast automatic mode, execution would have been even faster.

### Alternating Execution Modes

You can alternate interactive and automatic execution freely: each will begin where the other left off.

- Choose **General Simulation Options** from the **Set** menu.
- Specify **Fair Interactive** simulation.
- Choose **Interactive Run** from the **Sim** menu.

The **Stop Criteria Satisfied** dialog appears immediately. The status bar shows that you are still at Step 50. Execution cannot proceed, because the stop criteria **Until Step Number is 50** is still in effect.

- Use the **General Simulation Options** command to specify a **Stop Criteria** of **Until Step Number is 52**.
- Choose **Interactive Run** from the **Sim** menu.

Execution proceeds interactively through the usual breakpoints. When Step 52 is complete, a breakpoint will be reached and a stop criterion satisfied at the same time. What will happen?

- Experiment with execution modes and stop criteria until you feel comfortable with them.

Be careful what stop criteria you specify for automatic execution, because there is no way to interrupt such execution once it is underway. In order to be interruptible, automatic mode would have to repeatedly check for an interrupt request. This would slow execution down, which would be inconsistent with the purpose of automatic execution.

## Saving and Loading Execution States

After a complex net has been executed for a long time, its state represents a considerable investment of time: if that state were lost, the information contained in it could be recovered only by repeating the whole execution process.

When a diagram is saved with an ordinary **Save** command, all state information is lost: if it is reopened, the net will revert to its initial state on entry to the simulator.

You can also save a net along with its current state. A diagram that contains a net saved in a particular state is called a *saved state diagram*, or for brevity, a *saved state*. When a saved state is reopened, the net is still in the state it was in when it was saved, and execution can continue from that point as if there had been no interruption.

### Saving a State

You can create a saved state whenever you are in the simulator and execution is stopped between steps.

- Choose **Save State** from the **File** menu.

The **Save As** dialog appears. If there is already a saved copy of the net, you can overwrite it, or you can create a new copy by giving a different name. In either case the net will be saved along with its current state.

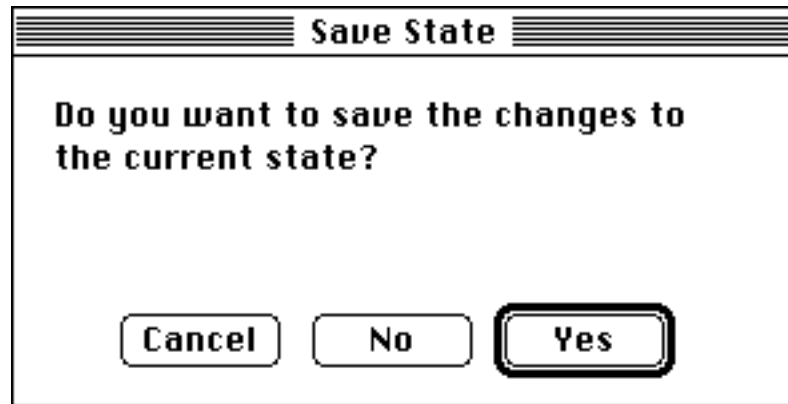
- Save the net in NewTTDiagrams under a name other than the original.

### Loading a Saved State

Loading a saved state is no different from opening a diagram generally, except that it is done from within the simulator.

- Execute the net a little more, in interactive or automatic mode.
- Choose **Load State** from the **File** menu.

The **Save State** dialog appears:



If you clicked **Yes**, the **Save As** dialog would appear; you could then save the net's current state, just as if you had explicitly chosen **Save State** from the **File** menu. There is no reason to do this now, so:

- Click **No**.

The **Save State** dialog disappears. The **Open File** dialog appears.

- Open the diagram you just created that contains the saved state.

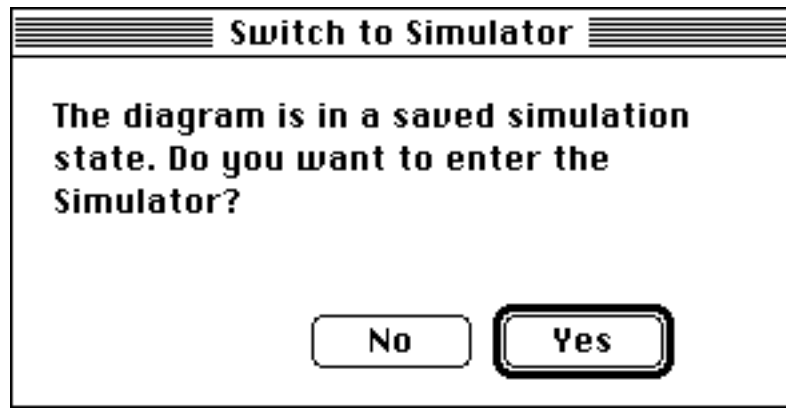
The diagram opens. It is in the same state that it was in when you saved it.

### Starting With a Saved State

You don't have to already be in the simulator and use **Load State** in order to make use of a saved state. You can use **Load State** from the editor, open a saved state from the editor, or you can start Design/CPN by opening the saved state, as you could with any ordinary diagram.

- Leave the simulator.
- Close the diagram containing the saved state.
- Open the diagram again.

If you took a saved state into the editor, and then into the simulator, the state would be destroyed and the net's initial state established. To protect against this, Design/CPN displays a dialog when you open a saved state rather than loading it:



If you clicked **No**, you would remain in the editor, and the saved state diagram would remain unopened.

- Click **Yes**.

The saved state diagram opens in the simulator. Everything is just as it would have been if you had already been in the simulator and had loaded the saved state with **Load State**.

## Saving the Net

Now let's save **FirstModel** to have the appearance, properties, and initial markings we'll need in the next chapter.

- Enter the editor.

All the simulation regions are still visible, for the reasons mentioned in Chapter 8. It will be convenient not to have to look at them while editing the net in the next chapter, but if you destroyed them with **Remove Sim Regions** (in the **CPN** menu) you would have to reposition them again next time you executed the net. The answer is to hide them.

- Select a group that consists of all visible marking key regions (the numbers in little circles).
- Choose **Hide Regions** from the **Makeup** menu.

The key regions and associated popups disappear. They still exist internally, and will reappear in the positions you moved them to, next time you execute the net.

You can't select a transition feedback region, so you can't hide such regions this way, but they represent no investment anyway, so if any are visible:

- Choose **Remove Sim Regions** from the **CPN** menu.
- Click options so that only **Feedback Regions** is selected:
- Click **OK**.

All transition feedback regions disappear. They will be back when they are needed.

- Set the occurrence set options so that **Transitions**, **Different Bindings**, and **Identical Bindings** are all at 100%.
- Set all breakpoints and token displays.
- Set `Order In`'s initial marking region to 10`Big + 10`Small.
- Set `Staff Pool`'s initial marking region to 4`Expert + 2`Novice.
- Set `Equip Pool`'s initial marking region to 2`Fast + 4`Slow

This net will be the basis for all future work you do in this tutorial.

- Save the net in NewTTDiagrams under the name NewFmodTimed.





# Chapter 16

## Simulated Time

In the last three chapter, you studied, built and executed a model called FirstModel. This model consists of a superpage containing an overview of a sales order system, a subpage containing a decomposition of that overview, and (as always) a global declaration node.

FirstModel gives a fairly realistic, though still quite high-level, view of a simple Sales Order System, except for one thing: it contains no representation of time. This results in many anomalies, some of which you may have noticed already.

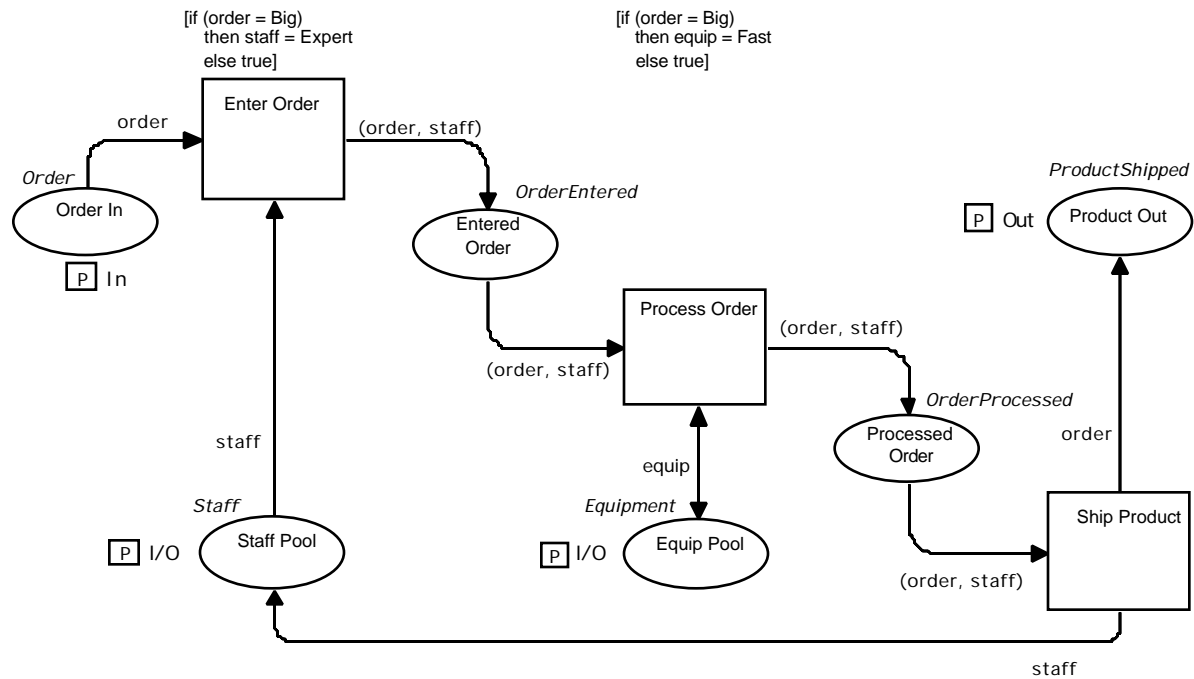
This chapter shows you how to represent and manage time in a CP net, and how to use this understanding to create a version of FirstModel that takes account of the fact that no activity happens instantaneously.

### The Untimed Version of FirstModel

- Open the diagram NewFmodTimed that you saved in the NewTTDiagrams directory while going through Chapter 15 (if it is not already open).

Be sure the net has the appearance, properties, and initial markings specified for it at the end of Chapter 15, or the instructions in this chapter may not have the effects described for them.

We don't need to be concerned with the hierarchy page or the superpage in this chapter. The subpage and global declaration node look like this:



```

color Order = with Big | Small;           (* Orders for products *)
color ProductShipped = Order;             (* Orders shipped *)
color Staff = with Expert | Novice;       (* Staff members *)
color Equipment = with Fast | Slow;       (* Pieces of equipment *)

color OrderEntered = product Order * Staff; (* Orders entered but unprocessed *)
color OrderProcessed = OrderEntered;      (* Orders processed but unshipped *)

var order: Order;                        (* An order *)
var staff: Staff;                       (* A staff member *)
var equip: Equipment;                   (* A piece of equipment *)
    
```

## The Nature of Simulated Time

In order to understand how time is represented in a CP net, we must first distinguish clearly between real time and simulated time. *Real time* is just that: time in the real physical world in which the simulator executes a model and we can watch what happens. *Simulated time* is just a symbolic representation of time that we may optionally build into a model. It has no reality outside the symbolic world of the model that contains it.

Real time and simulated time have no intrinsic relationship whatsoever. We may or may not build a symbolic representation of time into a model. If we do not, the sequence of states that an executing model passes through has no temporal interpretation: it is just a sequence of states.

This may seem strange. The state of an executing model evolves over time; we can watch it do so. How could a model's state change without a flow of time? The answer is that it cannot change without real time in which to execute, but it can change without reference to simulated time, because simulated time is just a symbol. It is nothing but what we define it to be, and if we do not define it at all, it does not exist.

It is easy but incorrect to equate the occurrence of a sequence of events within a model with the passage of time in the system that the model represents. The illusion arises because model execution occurs sequentially over real time in the real world. But you must resist this illusion in order to understand simulated time. It is nothing but a symbol that means what we define it to be

### Non-Representation of Time in FirstModel

Our approach to FirstModel up until now has intentionally glossed over any consideration of time. We have spoken of orders “waiting” in `Order In` for a staff member to become available in `Staff Pool`, and in `Entered Order` for equipment to become available in `Equip Pool`, but the relationship (if any) of such waiting to the passage of time has not been addressed.

The waiting of orders for staff members has a reasonable appearance to it. An order may exist in `Order In`, but if all appropriate staff members are busy with other orders, the order waits in `Order In` until `Ship Product` returns an appropriate staff member to `Staff Pool`.

But the waiting of orders for equipment is obviously problematical. Chapter 13 claimed that when an order exists in `Entered Order`, but all appropriate equipment pieces are busy with other orders, the order waits in `Entered Order` until `Process Order` returns an appropriate equipment piece to `Equip Pool`. You may have wondered about this statement, because equipment pieces are never “busy with other orders.” They are returned to `Equip Pool` in the same step that removes them.

Since no events intervene between the allocation of equipment and its return, equipment is being used instantaneously: it is used, but it is never in use. Equipment that exists but is never in use is, by definition, always available. So what are orders that remain in `Entered Order` waiting for? They are waiting for the net reach a state such that there is an enabling binding for `Process Order` so that the transition can fire and process one or more orders.

There cannot be more such bindings in any one step than there are tokens in `Equip Pool`, so when there are more orders than pieces of equipment, the additional orders must wait for subsequent steps.

But this stretching-out of order processing into a sequence of execution steps has no temporal implication within FirstModel, because we have not included in the model any representation of time.

Thus the unrealistic way in which orders and equipment interact is actually an artifact of the way we have constructed the model. We have represented activities that make sense only when there is a dimension of time, but have failed to provide the dimension. It is not even correct to say that equipment usage is “instantaneous,” because there is no simulated time and hence there are no simulated instants. There is only a sequence of states.

Actually the temporal unrealism of FirstModel is worse than it seems. The “reasonable appearance” of the waiting of orders in Order In for staff members is an illusion, caused by the fact that events do occur between the removal of a staff member from the staff pool and the subsequent return of that member. But these events have no temporal significance, so they provide no more temporal realism to the waiting of orders for staff than exists for the waiting of orders for equipment.

### Duration and Causality

The preceding arguments make it clear that the sequence of states that FirstModel goes through as it executes does not represent a time dimension, or reflect any structuring of events due to considerations of time. But the sequence of states must represent something. What does it represent?

The answer is: it represents the causal relationships that are defined by the structure of the model. These exist independently of the presence or absence of simulated time. The structure of FirstModel defines exactly what must be true in order for each possible event to occur. We can see these causal relationships by examining the net's structure, and observe their effects by watching the net execute.

Without an underlying structure of cause and effect there could be no events that might take time. Given such a structure, time can be expressed in terms of it. Let's see how that is done in a CP net.

### Representing Time in a CP Net

In order to take time into account, we need a way to represent and manipulate time within a model. A surprisingly simple methodology provides everything we need in order to represent time in a CP net:

1. A token may have an associated number, called a *time stamp*. Such a token is called a *timed token*, and its colorset is a *timed colorset*.

2. The simulator contains a counter called the *clock*. The clock is just a number (integer or real) whose current value is the current time.
3. A timed token is not available for any purpose whatever unless the clock time is greater than or equal to the token's time stamp.
4. When there are no enabled transitions, but there would be if the clock had a greater value, the simulator increments the clock by the minimum amount necessary to enable at least one transition.

That's all we need to create a dimension of simulated time that has exactly the properties that we need. Let's look more exactly at how these rules work to provide simulated time.

## How Simulated Time Works

Simulated time has nothing to do with the external time during which the simulator steps through net execution and observers possibly watch the execution process, or with the numbered sequence of steps by which the simulator executes a net. Simulated time is just an incrementable number that is globally available within an executing model. The value of this number can be thought of as the time indicated by a simulated clock. When the number is incremented, the clock moves forward to a later time.

The units of simulated time do not inherently represent any particular absolute time unit. We may interpret simulated time units as microseconds or millennia, depending on what we are modeling, but syntactically time is just a number. For brevity, simulated time is sometimes referred to as *model time*.

## Simulated Time and Transition Enablement

The state of a net changes only when enabled transitions fire. In order for simulated time to affect on net execution, it must therefore affect transition enablement and firing.

This effect is produced by the rule that a timed token is unavailable for any purpose unless the clock is greater than or equal to the token's time stamp. Such a token is ignored when transitions are checked for enablement: it might as well not be there at all.

In effect, a timed token does not exist until the clock reaches a certain time, given by the token's time stamp. When the clock reaches that time, the token suddenly springs into existence, becomes a fac-

tor in determining enablement, and can be subtracted by transition firing.

### The Simulated Clock

Simulated time could tick forward continually, as real time does, but that would be a very inefficient way to do things: the clock would frequently waste real time counting off intervals of simulated time during which the model remains unchanged. Such counting would accomplish nothing. It is more efficient in such a case to jump the simulated clock immediately to the next time when some change is possible, and proceed with executing the net.

Therefore the simulated clock does not move at a steady rate. Instead, it remains at its current value as long as there are any enabled transitions. When there are no enabled transitions left the clock jumps forward by the smallest amount necessary for at least one transition to become enabled. This implies that time will never move forward in a net that always has enabled transitions independently of time.

Simulated time passes when and only when the clock is incremented. Everything that happens while the clock remains at a particular setting is both simultaneous and instantaneous in simulated time.

This is convenient when a system contains an event that happens at a particular moment, but that is sufficiently complex that modeling it by firing a sequence of simple transitions, rather than one complex transitions, is most convenient. Leaving the clock unincremented during such a sequence lets us model the event as a manageable series of small changes, while preserving the instantaneity of the event's effect on the state of the model.

### Other Uses for Simulated Time

Simulated time is just a mechanism that follows certain rules. Nothing requires this mechanism to be used only for the purpose of simulating the passage of time. It can be used in any way that is useful.

A net that will use the charting facility to display information about net execution will probably need to initialize one or more statistical variables at the start of execution. You can accomplish this by using timing, but it is much more convenient to use the `init` section of a chart's code segment, as described in *The Design/CPN Reference Manual*.

## Specifying Timed Simulation

Not all nets need to use timing. To avoid adding overhead to such nets, a net that will use timing must declare the fact explicitly specifying the generation of timed code in **Simulation Code Options** dialog in the **Set** menu, and by specifying simulation with time in the **Generation Simulation Options** dialog in the **Set** menu.

## Declaring a Timed Colorset

To declare a timed colorset, declare it as you ordinarily would, and append the keyword `timed` to the declaration.

When a colorset is timed, duplicate colorsets and composite colorsets that include it are timed also, and therefore do not need be explicitly declared `timed`.

- Edit FirstModel's global declaration node to declare the colorsets `Order`, `Staff`, and `Equipment` to be timed:

```
color Order = with Big | Small timed;      (* Orders for products *)
color ProductShipped = Order;              (* Orders shipped *)
color Staff = with Expert | Novice timed;  (* Staff members *)
color Equipment = with Fast | Slow timed;  (* Pieces of equipment *)

color OrderEntered = product Order * Staff; (* Orders entered but unprocessed *)
color OrderProcessed = OrderEntered;       (* Orders processed but unshipped *)

var order: Order;                          (* An order *)
var staff: Staff;                          (* A staff member *)
var equip: Equipment;                      (* A piece of equipment *)
```

`ProductShipped`, `OrderEntered`, and `OrderProcessed` inherit the timed attribute, so all colorsets are now timed.

## Giving a Token a Time Stamp

Tokens get time stamps via expressions called *delay expressions*. A delay expression has the form:

`@+ expression`

where “@+” appears literally, and `expression` is an arithmetic expression.

A delay expression defines a time equal to the current simulated time (symbolized by the @ sign) plus (+) the value of the expression.



## Design/CPN Tutorial

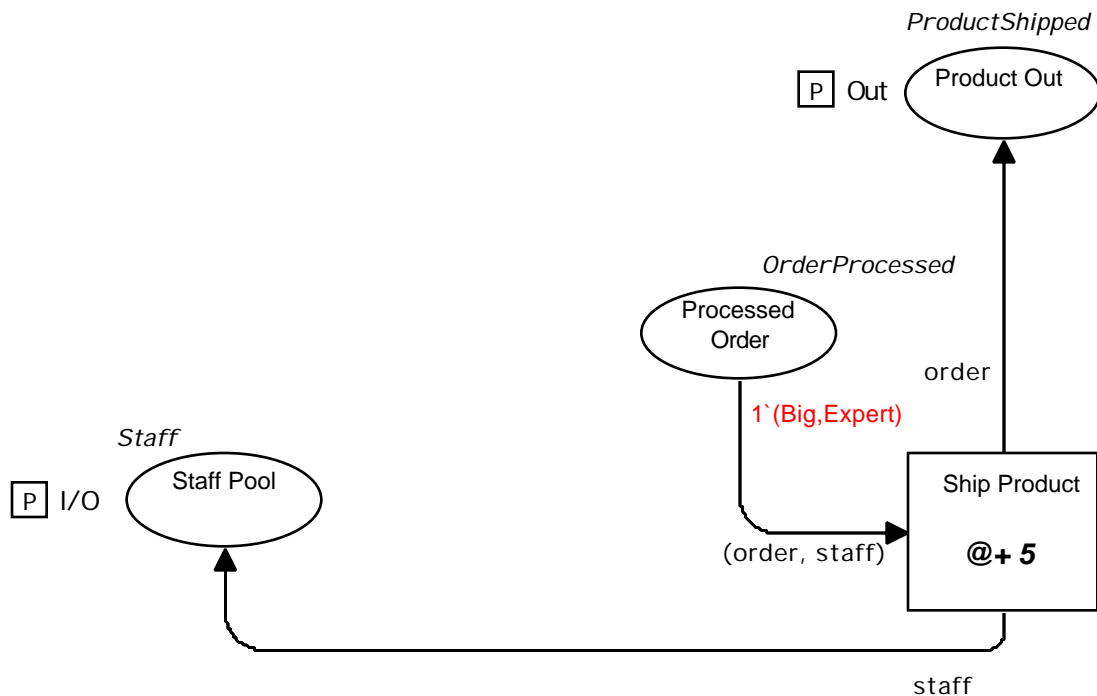
---

This value becomes the time stamp of any tokens created under the aegis of the delay expression.

There are two ways to use a delay expression to provide time stamps for tokens: by putting it in a time region, and by appending it to an output arc inscription.

### Delay Expressions in Time Regions

A *time region* is a region associated with a transition. The region contains a delay expression. Every output token of a timed colorset that is created by the transition will have a time stamp as designated by the delay expression. Output tokens of non-timed colorsets will be handled just as if there was no time region. For example:

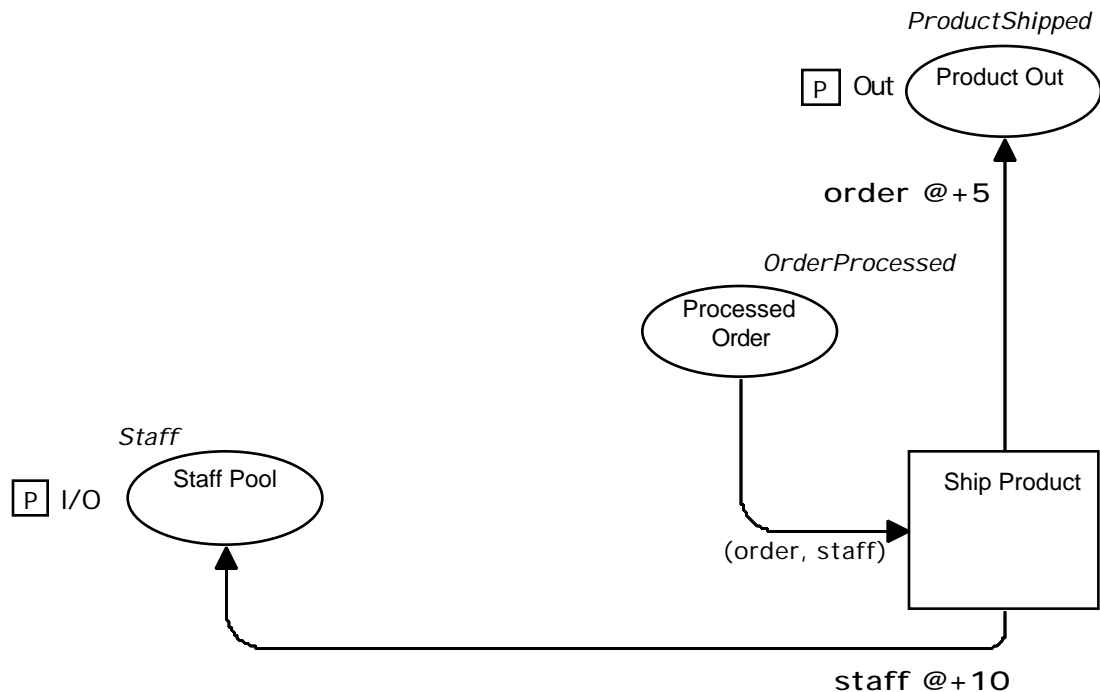


All tokens that Ship Product adds to Product Out or Staff Pool will have a time stamp equal to the model time when the tokens were created plus 5. They will therefore be effectively nonexistent until the clock has been incremented by at least 5. This might represent a situation in which shipping a product takes 5 minutes, after which the staff member is again available to process orders.

### Delay Expressions on Output Arc Inscriptions

When a time region is used, all output tokens of timed colorsets necessarily get the same time stamp. It is often convenient to give different time stamps to the tokens in different output places. This can

be accomplished by appending delay expressions to individual output arc inscriptions. For example:



All tokens that *Ship Product* adds to *Product Out* will have a time stamp equal to the model time when the tokens were created plus 5, and all tokens that it adds to *Staff Pool* will have a time stamp equal to the model time plus 10.

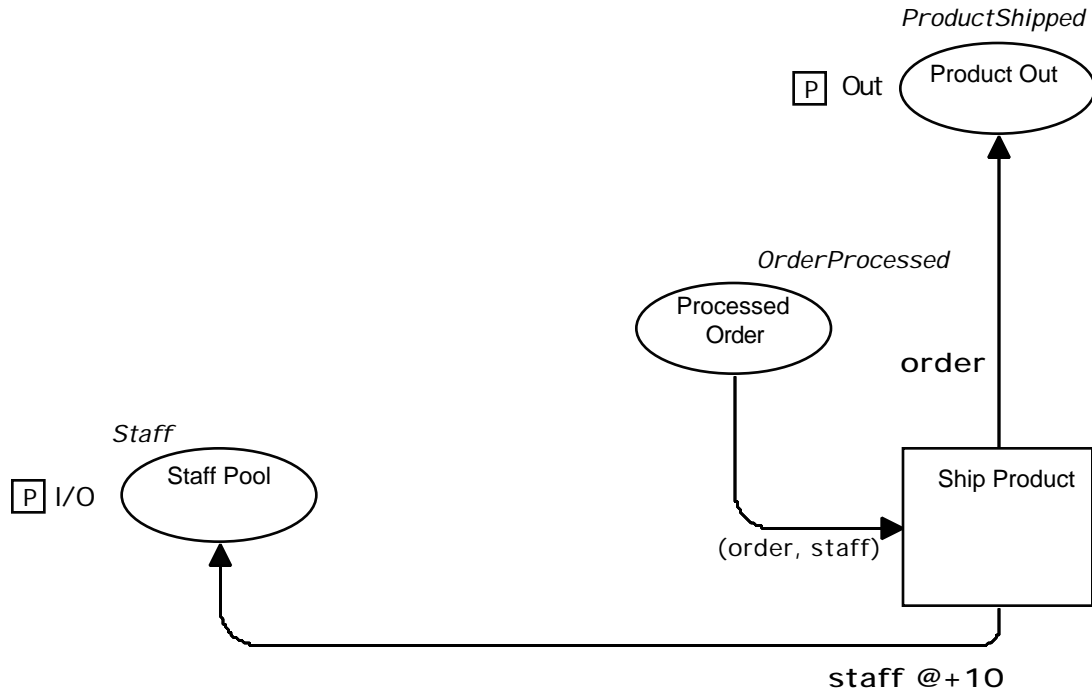
This might represent a situation in which shipping a product takes 5 minutes, after which the staff member takes a 5 minute break. The product is thus available for further processing after 5 minutes, but the staff member is not available to process another order for 10 minutes. (Since no logic is shown that uses the tokens in *Product Out*, their time stamp would make no actual difference, but such logic could easily exist.)

### Omitting a Time Stamp

The fact that a colorset is timed does not mean that time is necessarily of interest in everything a token of that colorset does. Therefore a timed token does not have to have a time stamp; it only has the option to have one. If no time stamp is given to a timed token, the default time stamp is the current time. Since the clock never has a negative value, such a token is guaranteed to be immediately available, just as if it came from a non-timed colorset but was otherwise the same.

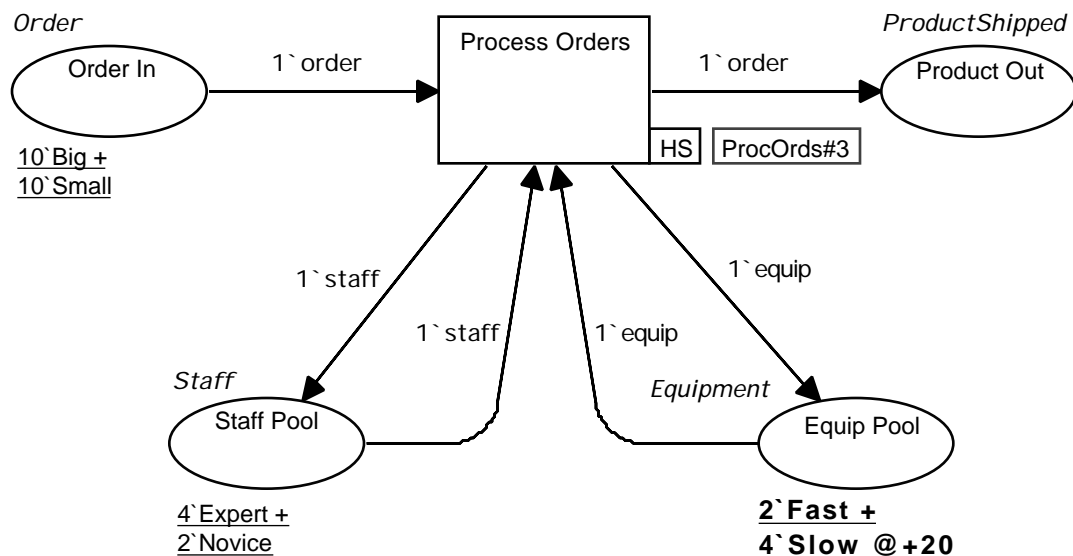
## Design/CPN Tutorial

For example, since time delays on tokens in `Product Out` accomplish nothing because the tokens will never be used again, the preceding example would be more economically expressed as:



### Time Stamps and Initial Markings

It is often useful to give a time stamp to the tokens in a place's initial marking. This is accomplished by appending a delay expression to the initial marking region. For example:

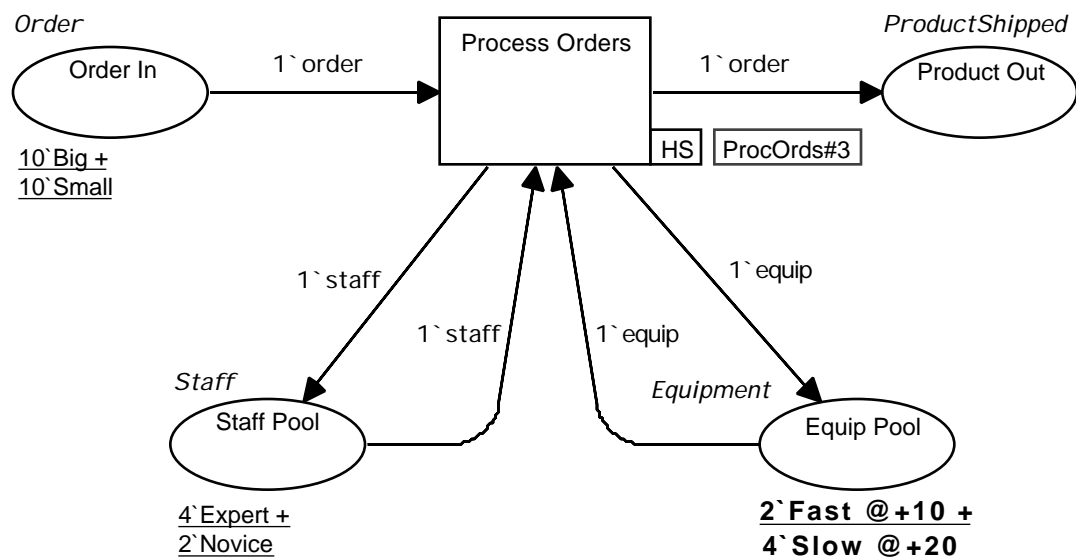


This could represent a situation in which equipment must warm up for 20 minutes at the beginning of a work day, and is not available until the warmup time has elapsed.

When no initial time stamp is specified for an initial token of a timed colorset, the default initial time stamp is the time specified in the **Time** section of the **Simulation Code Options** dialog (**Set** menu).

### Time Stamps and Multisets

When a multiset that defines more than one token is created, by an initial marking region or any other net component, all of the tokens in the multiset get the same time stamp. Thus:



This is illegal, and would cause a syntax error.

### Changing FirstModel to Assign Time Stamps

With these principles in mind, let's put some time delays into FirstModel. We'll stick with time regions rather than putting delay expressions on output arc inscriptions, and avoid putting time stamps into initial markings. Examples of these techniques would not demonstrate anything fundamental that examples of time regions do not.

For starters, suppose that every order takes 5 minutes to enter, 10 minutes to process, and 5 minutes to ship. To represent this, we will give `Process Order` the time region `@+10`, and the other two transitions a time region of `@+5`.

- Select the transition `Process Order`.
- Choose **CPN Region** from the **CPN** menu.

The **CPN Region** dialog for transitions appears:

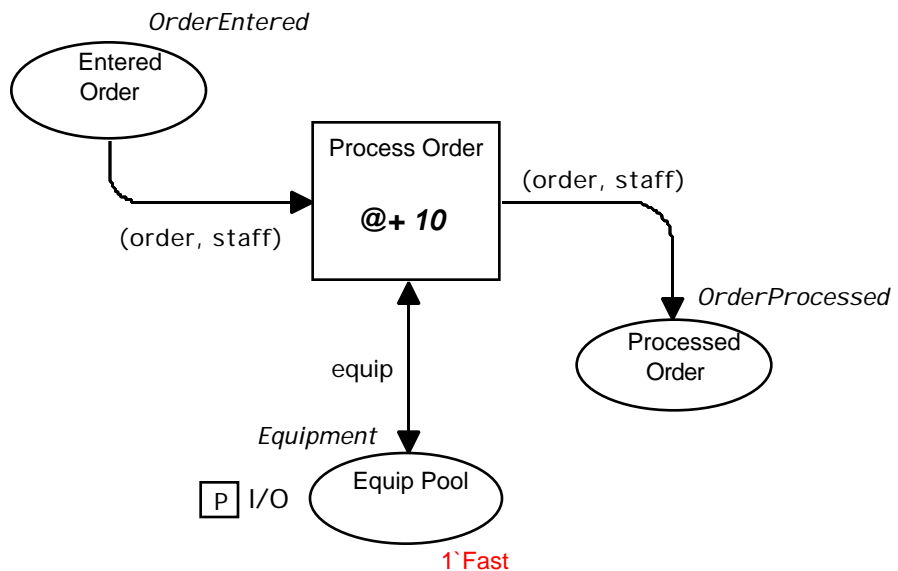


- Click **Time**.
- Click **OK**.

Creating a time region is no different graphically from creating a guard or any other text region.

- Click the mouse inside the transition.
- Type "@+10"

The transition should now look this (bolding excepted):



All the usual tricks work with time regions:

- Click on `Enter Order`.
- Give it the time region `@+5`.
- Press `ESC` to leave the creation mode.
- Execute **C**opy (via the **F**ile menu or a keystroke shortcut).
- Execute **P**aste

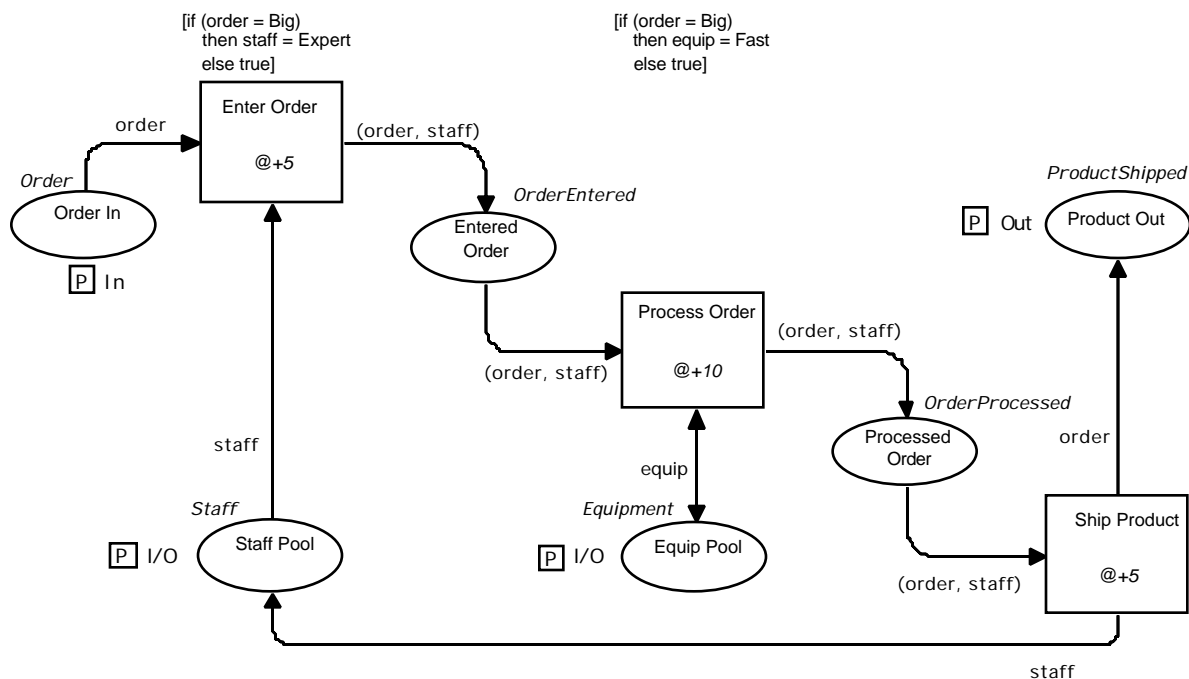
The editor pastes copy of the time region next to the original. This location is only temporary.

- Click on `Ship Product`.

The pasted time region moves to `Ship Product`.

- Adjust the time regions to center each in its transition.

The net should look like this:



## Compiling a Timed Net

When a net that uses time is compiled, code must be generated that is not necessary for an untimed net. Generating this code slows down the process of switching to the simulator. To allow you to

avoid this overhead, Design/CPN provides an option whereby you can specify whether a net is to be executed with or without time.

- Choose **Simulation Code Options** from the **Set** menu:

The screenshot shows a dialog box titled "Simulation Code Options". It contains three main sections: "Mode", "Time", and "Code Segments".

- Mode:** Three radio buttons are present: "Fair Simulation (Interactive and Automatic)" (selected), "Fast Simulation (Automatic)", and "Both".
- Time:** This section has two columns of radio buttons. The left column has "With" (selected), "Without", and "Both". The right column has "Integer" (selected) and "Real". To the right of these is a numeric input field with "0" in the top line and "0.0" in the bottom line.
- Code Segments:** Three radio buttons are present: "With" (selected), "Without", and "Both".

At the bottom of the dialog are five buttons: "Save...", "Load", "Reset", "Cancel", and "OK" (which is highlighted with a double border).

Note that the **With** option in the **Time** section is selected. This specifies that the compiler is to generate code for timed simulation. The **Integer** option specifies that time will be kept as an integer, and the number by the option specifies that the clock will read 0 when execution begins.

If you selected **Without**, then switched to the simulator, no code for managing time would be generated. If you tried this with the **FirstModel**, syntax errors would result, since it now contains instructions that make sense only in the context of timed simulation.

If you selected **Both**, code for timed simulated would be generated, but you could use the **General Simulation Options** dialog to specify that the net is to execute just as it would if there were no time constraints specified in it. We will do just that later in this chapter, so:

- Select **Both** in the **Time** section of the dialog.
- Click **OK**.

- Enter the simulator.

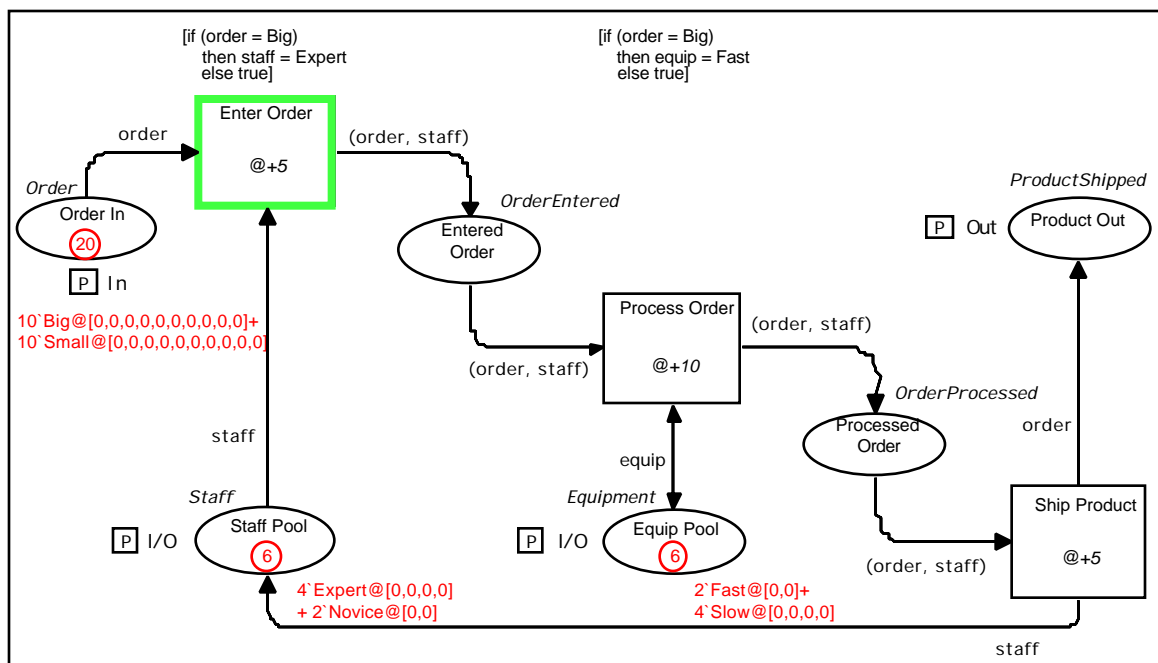
The switching process includes the generation of code for timed simulation.

## Executing a Timed Net

A timed net is executed just as any net is. The only difference is in what the net does as it executes.

- Reshape/reposition marking regions as needed.

The net should now look something like:



Note the way the time stamps of the individual tokens are indicated. The times indicated are not delays, but the actual times that the clock must reach before the various tokens become available. The time stamps all happen to be the same now, but this will not generally be the case as the net executes.

Take a look at the status bar. It displays: Time: 0 Step: 1.

- Set options to specify interactive execution with all break-points on and a stop criterion of **No Limit**.
- Execute three steps. Study the net carefully during and after each one.

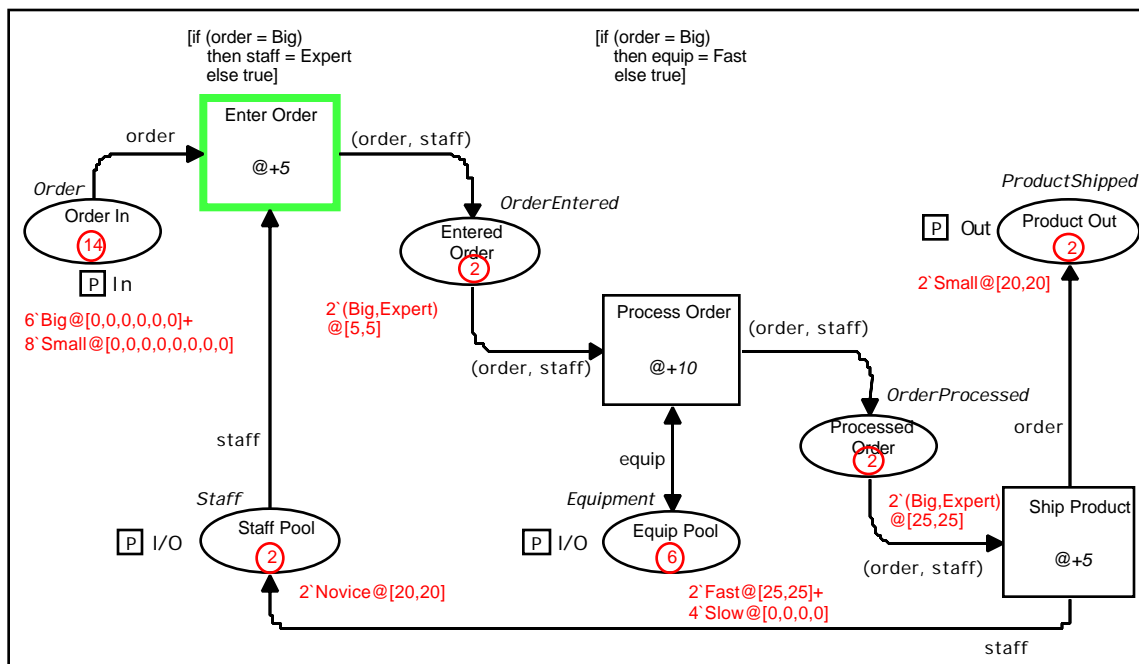


- Stop execution when the simulator finishes Step 3:

As you can see, timing hasn't made any great difference in the sorts of things you see as a net executes. All it has added is more realistic behavior with respect to time.

Unfortunately, it is not possible to depict with certainty how your net will look at this point, because the simulator may have made different random choices on your machine than it did when the figures in this chapter were made. If what you see differs from the figures so much that it fails to exemplify the points the tutorial is making, just study the tutorial figures instead.

The following shows one appearance that the net may have at this point:



Carefully examine Process Order and its input places. There are two Big orders in Entered Order, each with an associated Expert, and there are two Fast equipment pieces in Equip Pool, but Process Order is not enabled. Why not?

Look at upper left side of the status bar. The time is 20. Now look at the time stamps in the two Fast equipment pieces. Both stamps are 25. Therefore the tokens are not available; that is why Process Order is not enabled.

The pieces of equipment that the two Fast tokens represent are actually still in use to process the two orders whose tokens are now in Process Order, also with time stamps of 25. Note that Ship Order is also not enabled: the jobs will not be ready to be shipped until the model time reaches 25.

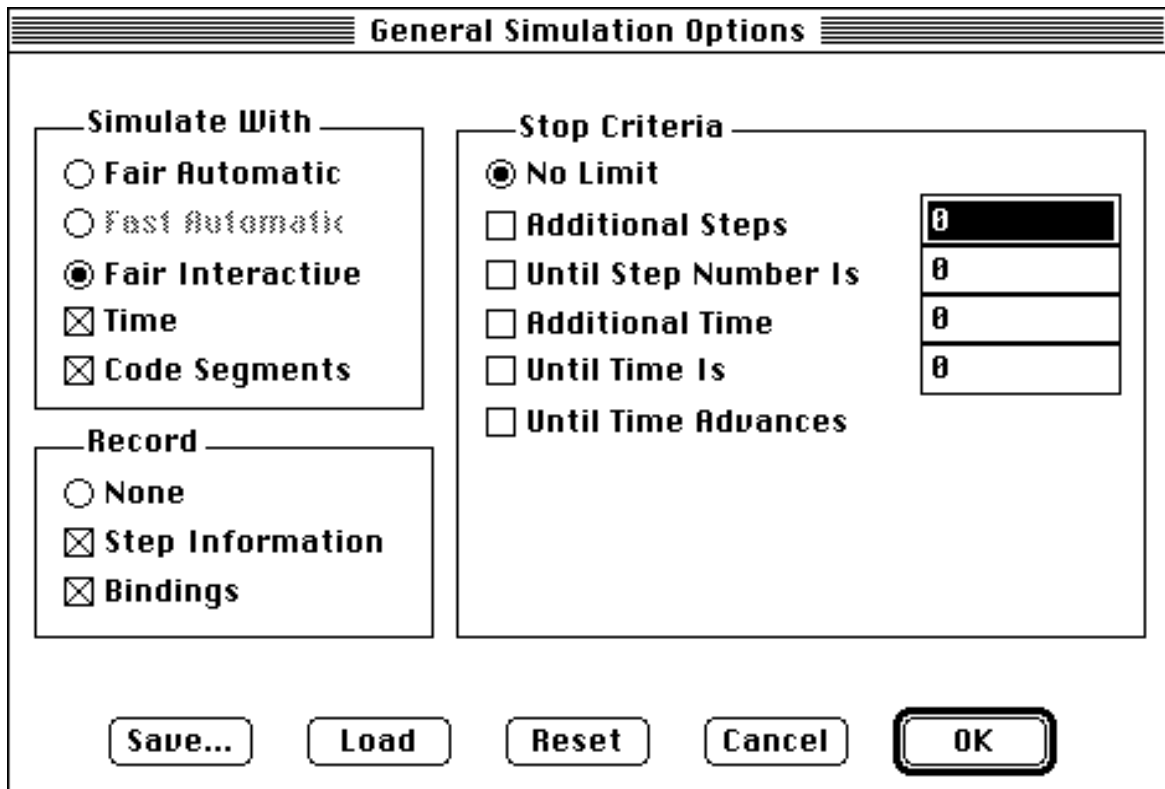
Tokens that represent unavailable entities are visible in their places, but functionally they might as well not be there at all. The simulator could have been designed to make such tokens actually invisible, which would better reflect their functional status, but doing so would make a net harder to understand by looking at it, since information would be missing.

## Simulation With and Without Time

It is sometimes useful to execute a timed net without taking account of time. Such execution can make it easier to examine the causal structure of a model, which sometimes becomes obscured when timing complicates the model's behavior.

Timed simulation can be turned on and off only if **Both** was selected in the **Time** section of the **Simulation Code Options** dialog when the net was compiled. The option was selected, so you can toggle timed simulation.

- Choose **General Simulation Options** from the **Set** menu:



The dialog box is titled "General Simulation Options". It contains three main sections: "Simulate With", "Record", and "Stop Criteria".

**Simulate With:**

- ☐ Fair Automatic
- ☐ Fast Automatic
- ☒ Fair Interactive
- ☒ Time
- ☒ Code Segments

**Record:**

- ☐ None
- ☒ Step Information
- ☒ Bindings

**Stop Criteria:**

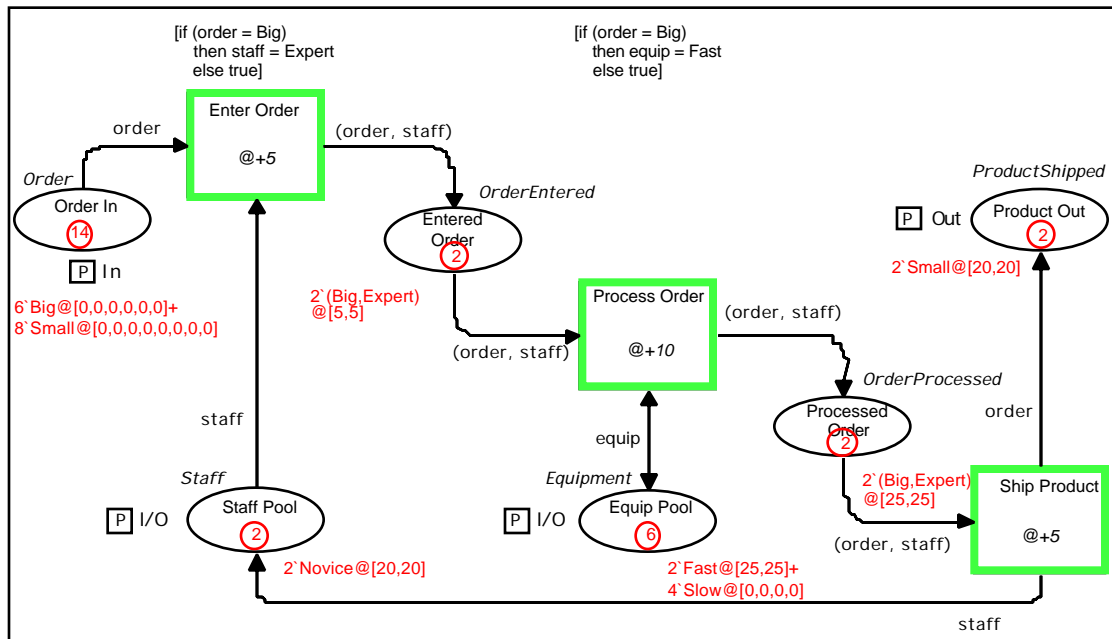
- ☒ No Limit
- ☐ Additional Steps: 0
- ☐ Until Step Number Is: 0
- ☐ Additional Time: 0
- ☐ Until Time Is: 0
- ☐ Until Time Advances

Buttons at the bottom: Save..., Load, Reset, Cancel, OK.

- Under **Simulate With**, click **Time**, so that it becomes de-selected.

- Click **OK**.

The net will now ignore all time stamps when it executes: tokens are available if they exist, no matter how they are stamped. Therefore (assuming the net state shown above) both `Process Order` and `Enter Order` are now enabled:



- Choose **General Simulation Options** from the **Set** menu.
- Under **Simulate With**, select **Time**.
- Click **OK**.

Time stamps are again operative. `Process Order` and `Enter Order` are no longer enabled.

## More Realistic Timed Behavior

Let's increase the realism of the model by changing it so that orders of different types are handled differently. To accomplish the change, all we need to do is have one or more time regions assign a delay that is conditional on the type of the order.

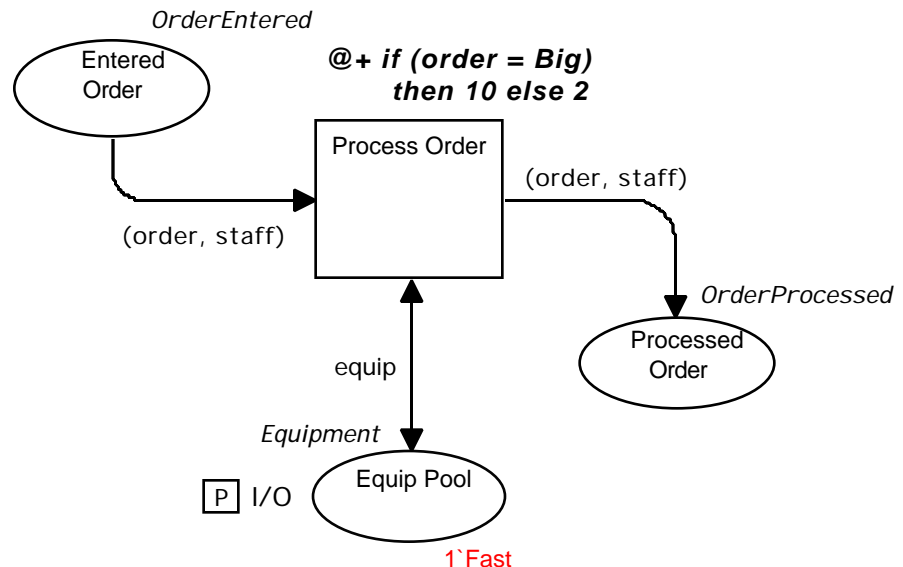
All bound arc variables are available to a time region, so everything we need to make this change is already available.

- Move `Process Order`'s time region to the clear area above the transition.

- Edit `Process Order`'s time region to be:

```
@+ if (order = Big)
  then 10 else 2
```

The transition should look like this:



- Execute **Reswitch** and **Initial State**, then execute the net.
- Execute the net for awhile in single steps.

Notice the more complex, and realistic behavior of the net.

- Remove all breakpoints, and just watch the net execute for awhile.

Can you identify any general patterns seem to recur as execution proceeds? Does the system appear to be operating efficiently?

- Make some changes to initial markings that you think will improve efficiency. Experiment to discover the effects of these changes.
- Change the times that orders spend in the various transitions different values, and study the results. Try again to produce efficiency.

You will soon notice that it is quite difficult to predict the results of a change by just thinking about it. You have to execute the net and find out what happens experimentally.

If even this tiny net cannot be modified without unexpected consequences, consider how impossible it would be to predict the results

of modifying a large net that models a real system. The system itself would be equally unpredictable in its response to change, but far more expensive than a model to repair after a change has not worked out as planned.

### Observing Simulation Results

It should be obvious by now that it is not easy to get a good picture of how efficiently (or inefficiently) a model is executing by watching the way tokens move. Gross efficiencies are generally obvious, but quantitative comparisons of different levels of efficiency resulting from different conditions cannot usually be made by just watching a net.

To make exact measurements of system performance both possible and accessible, Design/CPN includes numerous capabilities for gathering and displaying data generated during simulation. These capabilities are provided by the Statistical Variables and Charts Facility. For information on how to use this facility, see *The Design/CPN Reference Manual*.

# **APPENDIX A**

## **CPN Hierarchy Techniques**

# Chapter A1

## Introduction to Hierarchy

### Files for Use With This Appendix

The following files are used in this appendix:

1. ResourceModel (and ResourceModel.DB). These files contain a model called the Resource Use Model. This model is used to demonstrate CPN hierarchy.
2. ResmodSubtrans (and ResmodSubtrans.DB). These files contain the Resource Use Model in hierarchical form.

These files are kept in the TutorialDiagrams directory that is supplied with this tutorial. You may want to make copies of the files to experiment with. Such copies can be kept anywhere under any names (be sure to preserve the name match between unsuffixed and .DB files), and used in place of the files mentioned in this and the next two chapters. If you cannot find the files, or if the originals have been modified, (re)install them as directed in Chapter 0 of this tutorial.

### CPN Hierarchy

Models whose primary purpose is educational, or that represent very small systems, can often be drawn on a single page. However this practice would not suffice for making a realistic model of a large and/or complex system. Trying to model such a system on a single page would be like trying to write a complex program without using subroutines. Such a model, or such a program, would be far too complex, poorly structured, and redundant to be useful, or in many cases even constructible.

The answer is to allow a CP net to be kept on multiple pages that can be organized into a functioning whole, much as an ordinary program can be written as multiple modules that can be linked into an executable file. The system CP nets use to provide such modularization is known as *hierarchy*.

CPN hierarchy consists of two capabilities: fusion places and substitution transitions. These capabilities allow a net to be divided into modules, and provide facilities for linking the modules in various ways. Let's take a brief look at each one of them.

### Fusion Places

The fusion place capability allows CP net places that exist in different locations in a net to act functionally as if they were the same place. Such places are called *fusion places*, and a group of such places is called a *fusion set*.

Fusion places are similar to global variables in a conventional program. Just as references to a global variable by different parts of a program refer to the same variable and yield the same value, so uses of “different” places in a fusion set by different parts of a net are actually uses of the same place, and will find that place to have the same marking.

Fusion places can be used in many ways to simplify and generalize a net. For example, a net might model many different activities that all make use of the same pool of resources. Representing the pool as a fusion place would allow the various activities to be drawn on different pages and yet all have access to a single shared resource pool.

Fusion places are also useful in the context of a single page. When many arcs connect to a place, and these arcs come from physically distant locations on the page, it is often clearer to represent the place more than once on the page, and equate the various representations by including them all in a fusion set.

### Substitution Transitions

The substitution transition capability allows a CP net transition to represent an entire page of CP net structure. The effect is the same as if the page that the transition represents appeared physically at the site of the transition. Such a transition is called a *substitution transition*, and the page of net structure that it represents is called a *subnet* or a *submodel*.

Substitution transitions are similar to subroutines in a conventional program. The effect is not identical, because the substitution occurs physically, as with a macro, rather than by invocation, but the result is essentially the same:

1. A net can be implemented as multiple modules that can be modified independently of each other.



2. The various modules can be linked together as needed to create a net.
3. The same module can be used repeatedly at different places in a net, so that redundant logic need not be created to handle the same situation in different contexts.

For example, a net might model a computer installation with many identical workstations working in parallel on different tasks. Creating a submodel that represents the details of a workstation, and using that submodel as the value of many different substitution transitions, would allow just one submodel to represent all of the workstations.



# Chapter A2

## Fusion Places

In the nets we have worked with so far, each place has been an independent entity: there was no relationship between places except that provided by arcs and transitions.

Another form of relationship is possible in a CP net. We can establish a method for defining sets of places so that anything that happens to each place in a set also happens to all the other places in the set. The places are then functionally identical. Such places are called *fusion places*, and a set of fusion places is a *fusion set*.

Fusion adds nothing fundamentally new. If all the members of a fusion set are on the same page, we could replace the set with a single place and connect to it all the arcs that connected to any member of the set. If the members are on different pages, we could copy everything on the several pages to a single page, and again collapse the set.

Conversely, if a net contains a place that has many arcs connecting to it, or requires very long arcs to reach it, we could unfold it into several places, on the same or different pages, and so simplify the net's graphical structure without changing its meaning. Such unfolding is a common event during the process of net development. Frequently the need for it can be anticipated, and fusion places used from the beginning.

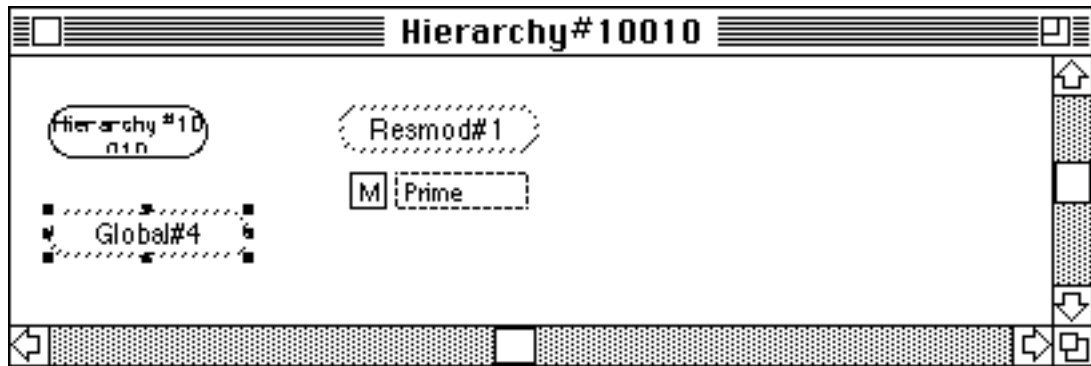
We really need only one type of fusion place to derive all the benefits fusion can provide. But as we shall see, it is useful to have different types that have different scopes. The rest of this chapter shows how to use fusion places.

## The Resource Use Model

This chapter demonstrates fusion place techniques using a model called the Resource Use Model. Let's take a quick look at this model before we begin working with it.

- Open the diagram ResourceModel, in the TutorialDiagrams directory.

The diagram's hierarchy page appears:



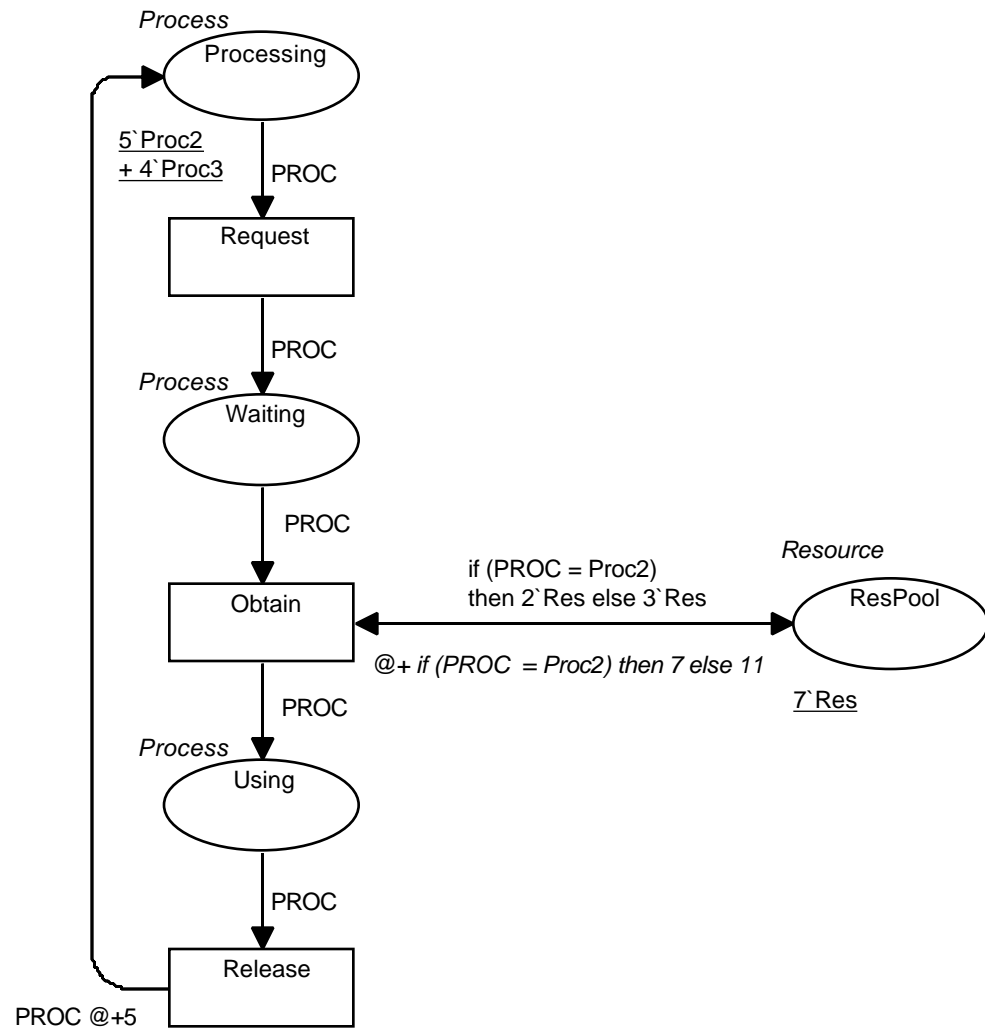
- Open the page Global#4.

This page contains the global declaration node:

```
color Process = with Proc2 | Proc3 timed;  
var PROC:Process;  
color Resource = with Res timed;
```

- Return to the hierarchy page.
- Open the page Resmod#1.

This page contains the executable part of the diagram:



## Description of the Model

The Resource Use Model represents a simple computer system in which processes vie for resources. Each process goes through a cycle in which it develops a need for resources, requests them, possibly waits for them, then obtains, uses, and releases them.

There are two kinds of processes, *Proc2* and *Proc3*, as defined by the colorset *Process*. There is only one type of resource, *Res*, defined by the colorset *Resource*. A *Proc2* process uses two *Res* resources at a time, and a *Proc3* process uses three *Res* resources at a time, as defined by the inscription on the input/output arc between *Obtain* and *ResPool*. A *Proc2* process uses resources for 7 time units, and a *Proc3* process uses them for 11 time units, as defined by the time region on the transition *Obtain*.

### Executing the Model

Before we start working with this model, let's see how it executes.

Since the diagram containing the net was not created on your system, it does not contain the information necessary to allow Design/CPN to communicate with the ML process. Therefore you must load that information into it before you can execute it.

- Choose **ML Configuration Options** from the **Set** menu.

A dialog appears.

- Click **Load**.
- Click **OK**.

The necessary information about the ML process is copied to the diagram.

- Enter the simulator.
- Execute the model.
- Do a few of the usual experiments with net execution.

You should soon feel at home with what the Resource Use Model does. It does not, of course, do very much, but that is not the point. We will soon be using it to create a variety of sometimes complex structures, and you will find it helpful to first become clear on what it does in its simplest form.

### Fusion on a Single Page

To study fusion places, let's implement the example mentioned in Chapter A1, in which different activities all share a common pool of resources. Let's look first at the simplest case, in which all the activities are represented on the same page, then proceed to the slightly more complex case in which they are on different pages.

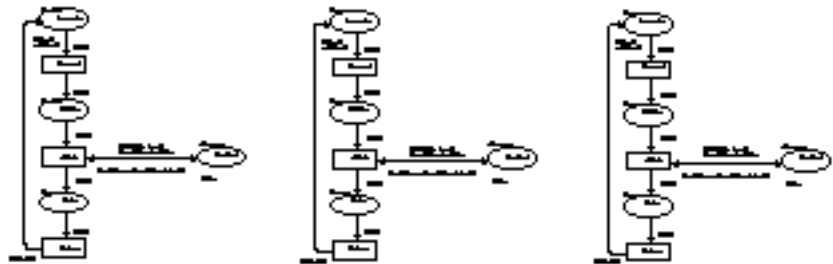
To implement fusion on a single page, we'll make three copies of the net on page Resmod#1 (shown above), then create a single resource pool that is shared by all three copies.

- Make Resmod#1 the current page.
- Select and then **Copy** all nodes on the page.
- Execute **Paste**.

The copied nodes will be pasted over the originals, a little lower and a little to the right.

- Use the mouse to adjust the group of pasted nodes so that it is about an inch to the right of the originals.
- Scroll the window if necessary to provide room for a third copy to the right of the one you just created.
- Execute **Paste**.
- Use the mouse to adjust the group of pasted nodes so that it is about an inch to the right of the first copy, with no overlap.

There will probably now be more graphics on the page than you can see at once. If you could see the whole page, it would look about like this:



The rest of this chapter refers to the net you just copied as Resnet, and the three copies as Resnet1 (on the left), Resnet2 (in the middle), and Resnet3 (on the right).

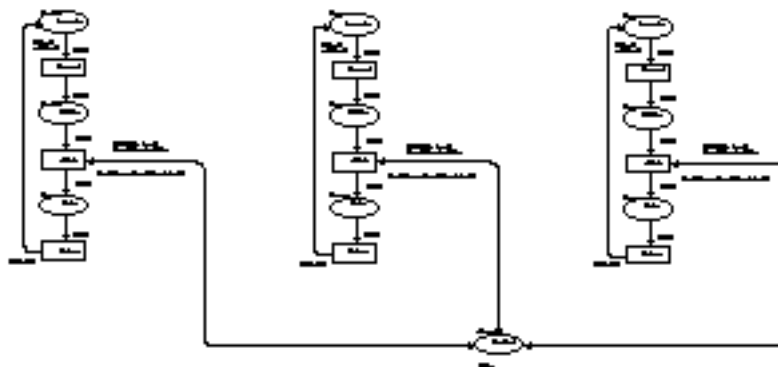
### Results of Executing the Diagram

If you executed the diagram as it is now, what would happen? Since there is no functional connection between the three copies, they would execute side by side just as if each existed alone. They would have no effect on each other at all. We could consider the copies to be three independent nets or a single net with three disjoint components. Nothing requires a CP net to be connected, so the two interpretations are equally valid.

However, it would obviously be useless to have three identical independent nets executing simultaneously. None of them would do anything that the others would not do; the only effect would be to slow execution down.

### Combining the Resource Pools

Let's make the situation more interesting by making all three copies share a single pool of resources. We could of course do this by deleting two of the resource pools and connecting the third to all three copies. The result could be something like:



This technique would be enough for the simple case we are working with, but if the resource pool were needed in many more locations that were much farther apart on the page, a lot of very long arcs would be needed. These would clutter up the page. It would be better to leave the resource pools where they are, and combine them functionally into a single pool by putting them in a fusion set.

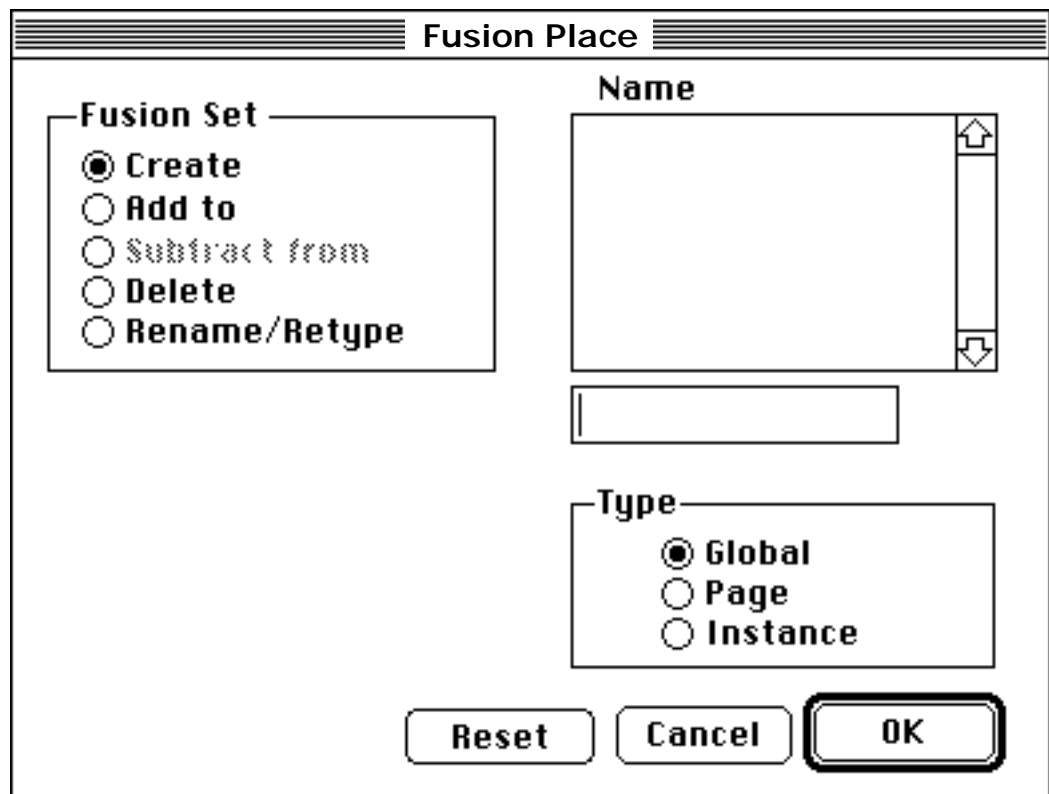
### Creating a Fusion Set

Let's create a fusion set that contains the place `ResPool` in Resnet1, and then add the `ResPools` in Resnet2 and Resnet3 to the set.

- Select `ResPool` in Resnet1.
- Choose **Fusion Place** from the **CPN** menu.

The **Fusion Place** dialog appears:





The image shows a dialog box titled "Fusion Place". It has a title bar with the text "Fusion Place". Inside the dialog, there are two main sections. The first section is labeled "Fusion Set" and contains five radio buttons: "Create" (which is selected), "Add to", "Subtract from", "Delete", and "Rename/Retype". The second section is labeled "Name" and contains a large text entry field with up and down arrow buttons on its right side. Below the "Name" section is a smaller, empty text entry field. At the bottom of the dialog, there are three buttons: "Reset", "Cancel", and "OK" (which is highlighted with a double border). Below the "Name" section, there is a section labeled "Type" with three radio buttons: "Global" (selected), "Page", and "Instance".

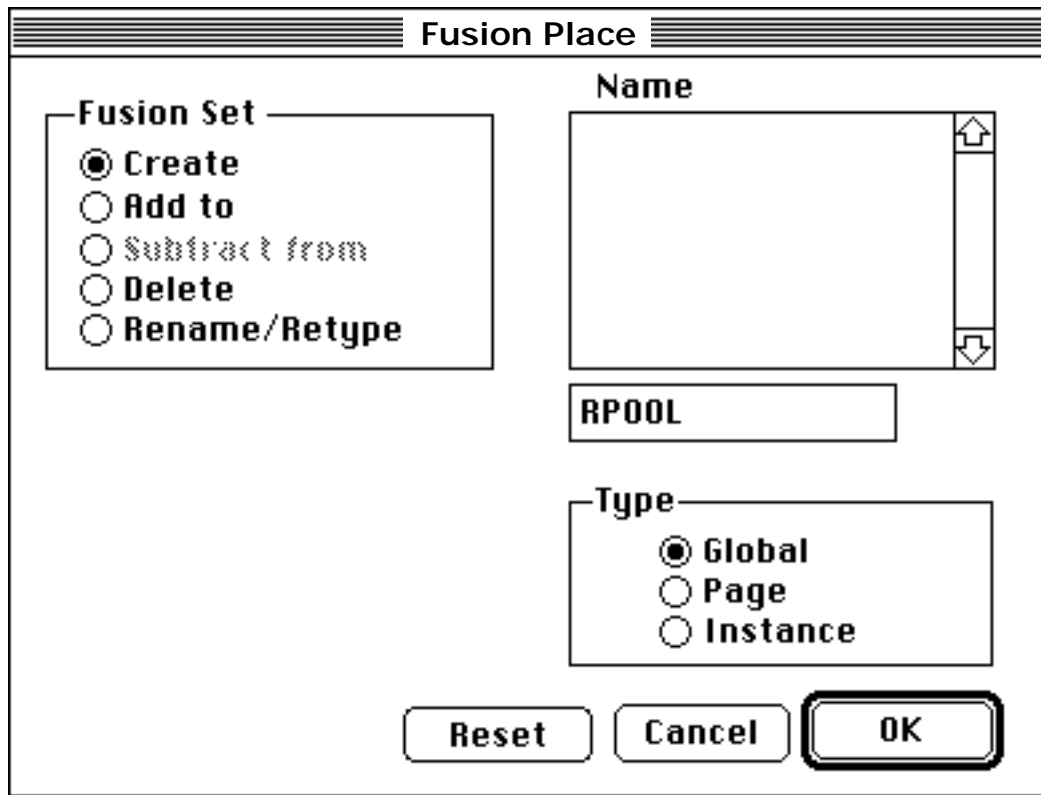
This dialog allows you to create and edit fusion sets. **Create**, the operation we want, is already selected as the default.

When you create a new fusion set, you must give it a name and indicate its type. Let's name the new fusion set "RPOOL".

- Type "RPOOL" into the edit box immediately below the **Name** section.

We want a fusion set that simply equates all constituent places wherever they occur. Such a set is called a *global fusion set*. Since **Global** is the default type in the dialog, we don't need to do anything explicit to indicate the new set's type. (**Page** and **Instance** fusion sets will be explained later in this chapter.)

The dialog should now look as follows:



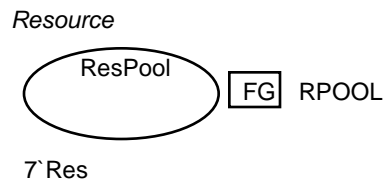
- Click **OK**.

The dialog disappears.

You have now created a global fusion set named **RPOOL**. The set contains one place, **ResPool**. A place that is a member of a global fusion set is called a *global fusion place*.

### Physical Appearance of a Global Fusion Place

The status bar now lists the type of **ResPool** as Place, Global-Fusion, and **ResPool**'s appearance has changed:



The **FG** marker is a *Fusion Key Region*. It indicates that the place is a global fusion place. The name of the fusion set to which the place belongs, **RPOOL**, is indicated next to the Fusion Key Region, in another region called the *Fusion Region*. The Fusion Region is a

popup, so you can hide and redisplay it by double-clicking on the associated key region.

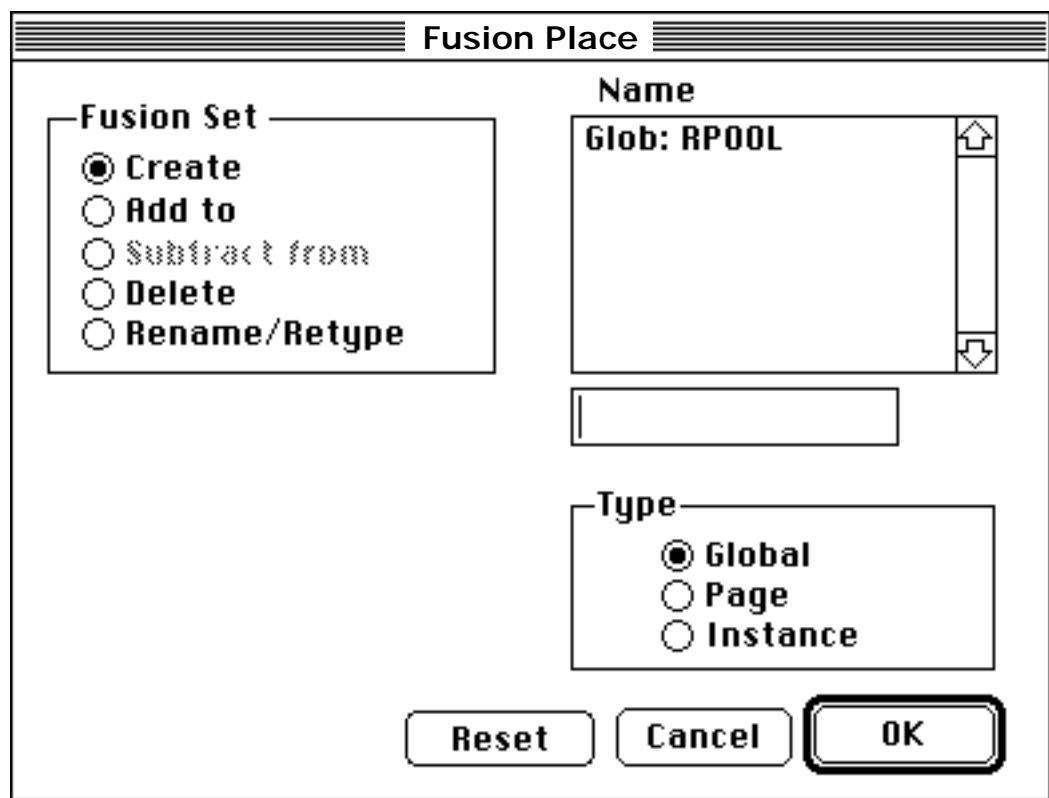
- Hide and redisplay “RPOOL” by double-clicking on .

### Adding Places to a Fusion Set

A fusion set with one place obviously accomplishes little. Let's add the other two `ResPool` places to the set. We don't have to add them one at a time: we can add them simultaneously.

- Select both `ResPool` in `Resnet2` and `ResPool` in `Resnet3`. Be sure that nothing else is selected: the status bar should display: Group of 2 Nodes.
- Choose **Fusion Place** from the **CPN** menu.

The **Fusion Place** dialog reappears:



The **Fusion Place** dialog box is shown. It has a title bar with the text "Fusion Place". Inside, there are several sections:

- Fusion Set**: A group box containing five radio buttons: **Create** (selected), **Add to**, **Subtract from**, **Delete**, and **Rename/Retype**.
- Name**: A text box containing "Glob: RPOOL". To the right of the text box is a vertical scrollbar. Below the text box is an empty text input field.
- Type**: A group box containing three radio buttons: **Global** (selected), **Page**, and **Instance**.
- At the bottom, there are three buttons: **Reset**, **Cancel**, and **OK** (which is highlighted with a double border).

Note the listing of **Glob: RPOOL** in the box under **Name**. This box lists all existing fusion sets, both to help you keep track of what they are, and to let you select an existing set without retyping its name.

We want to add to an existing set, so:

- Click **Add To** under **Fusion Set**.

The edit box and the **Type** section disappear, since we can add only to a fusion set that already exists.

- Click on **glob: RPOOL** in the box under **Name**.
- Click **OK**.

The dialog disappears. All three `ResPool` places are now part of the fusion set **RPOOL**: they are functionally not three places, but one place that is represented three times in three different locations.

As you may have guessed, you could have selected all three `ResPool` places before creating **RPOOL**. They would all have then been members of **RPOOL** from the start, and there would have been no need to add any afterwards. In general, any fusion-related operation that is meaningful for a group of places can be executed on a group.

### Initial Markings and Fusion Sets

All three places in **RPOOL** have an initial marking region, and each region specifies the same marking. In the editor, they could just as well have had different markings: they are just text regions, so it does not matter if they agree or not.

The simulator is another matter. When you enter the simulator, any initial marking regions are evaluated, and the tokens they specify are put into the corresponding places: marking regions are converted into actual markings. But the places in a fusion place, being functionally one place, intrinsically can have only one marking. Therefore all places in a fusion set must have the same initial marking region (if any) before entry to the simulator. If they do not, a syntax error will occur, and you will remain in the editor.

### Removing Places From a Fusion Set

Removing places from a fusion set is just the inverse of adding them.

- Select `ResPool` in Resnet3. Be sure no other places are selected.
- Choose **Fusion Place** from the **CPN** menu.
- **Subtract From** is already selected under **Fusion Set**. That is the operation we want, so:
- Click **OK**.

ResPool in Resnet3 is now an ordinary place. The other two ResPool's are still equated in the fusion set.

As a shortcut, you can remove a place from a fusion set by deleting its Fusion Key Region.

- Select the Fusion Key Region of ResPool in Resnet2.
- Press DELETE.

The Fusion Key Region and the Fusion Region disappear. ResPool in Resnet2 is now an ordinary place. The remaining ResPool, in Resnet1, is still a member of the set.

- Put the ResPools in Resnet2 and Resnet3 back into the RPOOL set.

### Deleting a Fusion Set

It is sometimes useful to delete a fusion set. All of its constituent places then revert to independent status.

- Choose **Fusion Place** from the CPN menu.
- Click **Delete** under **Fusion Set**.
- Select Glob: RPOOL under Name.
- Click **OK**.

The dialog disappears. RPOOL is gone; all ResPool places are now freestanding, as if RPOOL had never existed.

### Fusion Across More Than One Page

In the context of a single page, fusion does not add any fundamental power, since anything fusion can do on a single page could be done by drawing arcs. Fusion on a page can be very convenient, but the real purpose of fusion is to make it possible to establish connections between net structures that exist on different pages. Without fusion there would be no way to do this, because there is no way to draw an arc that runs between one page and another. With fusion we can equate places on one page with places on another, and so connect the pages.

We need an additional page to practice on, and we don't need all three copies of Resmod any longer; two will be enough.

- Select all seven nodes in Resmod3. Be sure that nothing else is selected.
- Execute **Cut** (via the **File** menu or a keystroke shortcut).

Resmod3 disappears. (You could also have used DELETE, but it is a good practice to use **Cut** for large deletions: with **Cut** you can undo the deletion via **Paste**, but if you DELETE the deleted structures are gone irretrievably.)

Now let's make a copy of Resmod#1. We could create a new page, then cut and paste from Resmod#1 to create the copy page, but let's try something different: using the file system to copy Resmod#1 in one operation.

### Saving and Loading a Subdiagram

Design/CPN allows you to save a page to a disk file, then load that page into a diagram, resulting in a new page that is a copy of the saved page. This facility can be used to duplicate pages within a diagram, or to copy pages from one diagram to another.

- Make Resmod#1 the current page.
- Choose **Save Subdiagram** from the **File** menu.

The **Save As** dialog appears.

- Save Resmod#1 under the name ResPage1.
- Choose **Load Subdiagram** from the **File** menu.

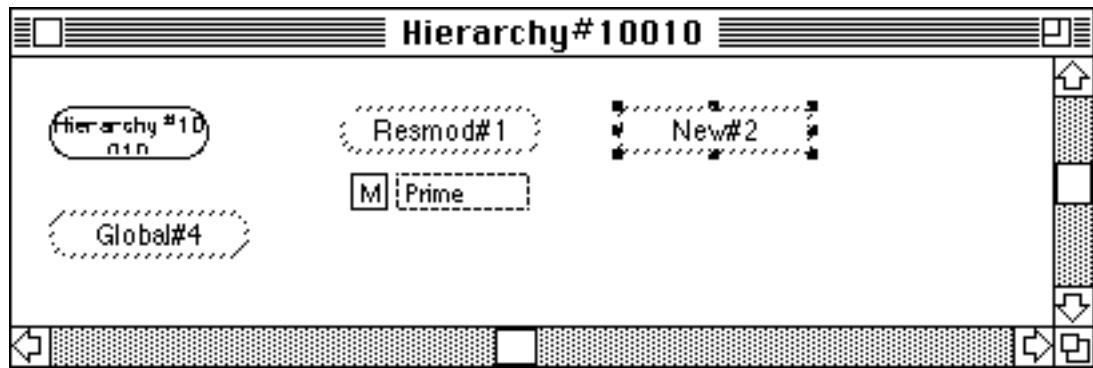
The **Open File** dialog appears.

- Open the saved page ResPage1.

A new page is created, named New#2. It contains a copy of the saved page Resmod#1.

- Open the hierarchy page.

The page looks approximately as follows:



New#2 is shown as a box rather than an oval to indicate that it has been loaded into the diagram rather than created as part of it.

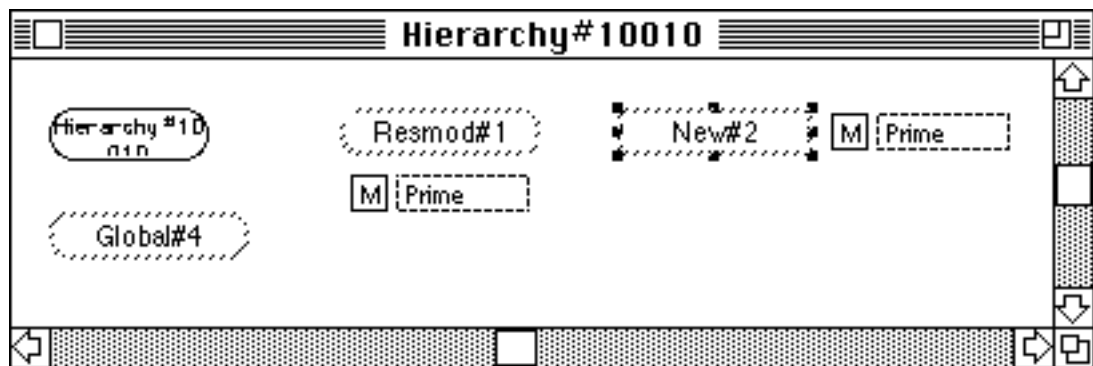
Note that **Load Subdiagram** doesn't care where the page it loads came from: it just loads it. If you had closed ResourceModel, opened some other diagram, and loaded ResPage1 into it, the effect would have been to copy Resmod#1 between diagrams.

### Make the New Page Prime

New#2 must be a prime page, or the simulator will ignore it.

- Make New#2 a prime page (**Mode Attributes** in the **Set** menu)

The hierarchy page shows that New#2 is now prime:



## Working With Fusion Sets That Span Pages

Working with a fusion set that equates places on more than one page is essentially the same as working with a single-page set. The only differences result from the fact that it is impossible to form a group of places (or anything else) that extends across pages. For example, we could not create a fusion set containing all four ResPools in one operation. We need at least two: one for the ResPools on Resmod#1, and a second for those on New#2.

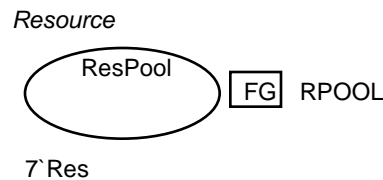
Let's create a fusion set that equates all four `ResPools`:

- Create a group that contains both `ResPools` on Resmod#1.
- Use **Fusion Place** in the **CPN** menu to create a global fusion set named **RPOOL**. Use the same techniques that you used to create **RPOOL** before.

**RPOOL** contains both `ResPools` on Resmod#1, because they were selected at the time it was created. Now let's add the other two `ResPools` to the set:

- Make New#2 the current page.
- Create a group that contains both `ResPools` on New#2.
- Use **Fusion Place** in the **CPN** menu to add the `ResPools` to **RPOOL**. Use the same techniques you used to add places to **RPOOL** before.

Now look at each of the four `ResPools`. All have the same appearance:



This is just how the two `ResPools` on Resmod#1 looked when **RPOOL** equated places on only one page. Its extension to equate places on more than one page adds nothing new.

There is no need at this point to practice deleting places from **RPOOL**, or deleting **RPOOL** as a whole. As you probably expect, these operations are no different when there are multiple pages than when there is just one.

## Working With More Than One Fusion Set

The existence of more than one fusion set at a time adds nothing fundamentally new. The only requirement is that each fusion set must have a unique name.

The current net models a system in which there are four computers that share a resource pool. Each computer handles for itself all the details of allocating, using, and releasing resources. This is a common configuration where many workstations are networked together.

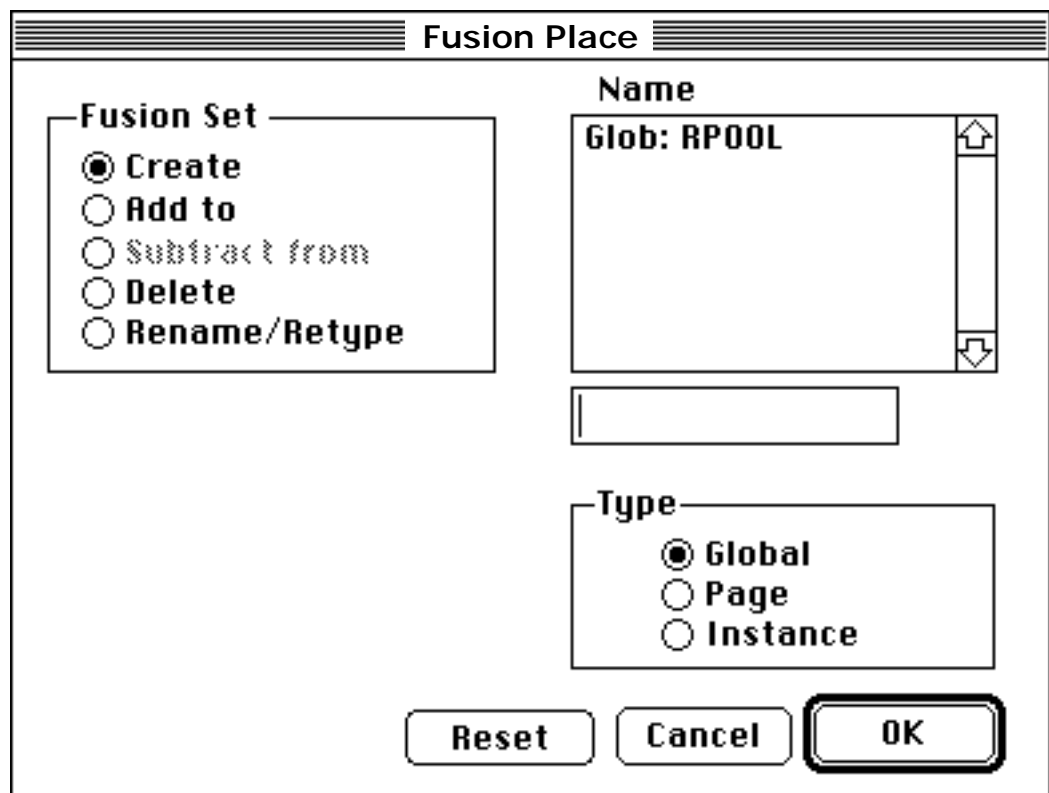


But consider a different system, in which there is a single mainframe computer that handles all processes. Mainframe computer time tends to be expensive, and resource usage can be very time consuming. Therefore it is a common practice to connect a mainframe computer to several less expensive subsidiary computers, and to let the subsidiary computers handle the details of resource allocation, use, and deallocation. The resources themselves remain in a common pool, available to any subsidiary.

Lets modify our net to model such a system. The necessary change is minor: all we need to do is to create a second fusion set that equates the four `Processing` places. That set will represent the mainframe, **RPOOL** will continue as the shared resource pool, and unshared parts of the four `Resnets` will represent four subsidiary computers. Let's call the new fusion set **MFRAME**.

- Select both `Processing` places on `Resmod#1`.
- Choose **Fusion Place** from the **CPN** menu.

The **Fusion Place** dialog appears:



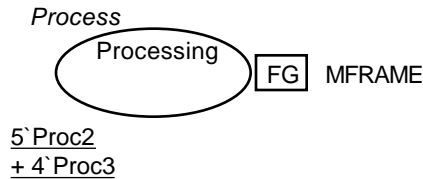
The **Fusion Place** dialog box is shown. It has a title bar with the text "Fusion Place". Inside, there are two main sections. The left section is titled "Fusion Set" and contains five radio buttons: "Create" (selected), "Add to", "Subtract from", "Delete", and "Rename/Retype". The right section is titled "Name" and contains a text box with the text "Glob: RPOOL" and a vertical scrollbar. Below the text box is an empty text input field. Below the input field is a section titled "Type" with three radio buttons: "Global" (selected), "Page", and "Instance". At the bottom of the dialog are three buttons: "Reset", "Cancel", and "OK" (which is highlighted with a double border).

The fact that one fusion set already exists makes no difference to the creation of another set. **Create** and **Global** are already the defaults, so just:

- Type "MFRAME" in the edit box below the **Name** section.

- Click **OK**.

Both `Processing` places on `Resmod#1` now look like this.



Now to complete the new fusion set:

- Select both `Processing` places on `New#2`.
- Choose **Fusion Place** from the **CPN** menu.
- Click **Add To** under **Fusion Set**.
- Click on **Glob: MFRAME** in the box under **Name**.
- Click **OK**.

The net now models a mainframe computer, represented by **MFRAME**; four subsidiary computers for handling resources; and a shared resource pool, represented by **RPOOL**.

## Page Fusion Sets

A shared resource pool is often a good idea. For example, we would not want every printer user to have its own dedicated printer if one shared printer would be enough to serve them all. On the other hand, complete resource sharing is not always desirable. A computer network might be distributed through several buildings, but its users would probably want print jobs to be done only on printers in their own building. How can the net be modified to model this kind of situation.

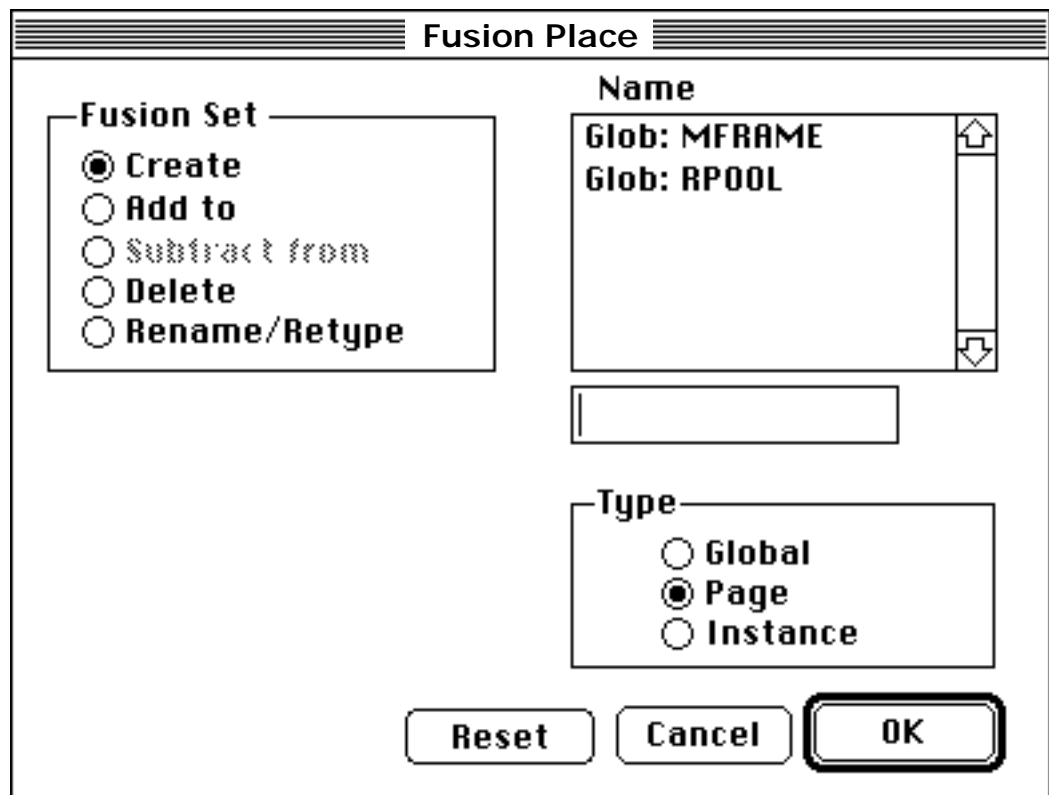
The obvious answer is just to use multiple fusion sets: we could remove some of the `ResPools` from **RPOOL** and put them into as many other fusion sets as we needed. But if a large model required many similar but separate fusion sets, this technique would result an annoying proliferation of different names for different instances of essentially the same thing.

There is a better way: *page fusion sets*. Page fusion sets are identical with global fusion sets (the kind we have been working with so far) in every way but one: while a global fusion set equates every constituent place irrespective of page, a page fusion set is divided

into subsets, each of which equates only constituent places that are on the same page.

Put another way, a page fusion set is a collection of fusion sets that have the same name but exist on different pages. The members of such a collection are called *page fusion subsets*, and each constituent place is called a *page fusion place*.

Page fusion sets are created and modified much as global sets are. The only difference occurs at the beginning: when a page fusion set is first created, select **Page** rather than **Global** in the **Type** section of the **Fusion Place** dialog:



The image shows a dialog box titled "Fusion Place". It has a title bar with the text "Fusion Place". Inside the dialog, there are several sections:

- Fusion Set**: A section with five radio buttons: **Create** (selected), **Add to**, **Subtract from**, **Delete**, and **Rename/Retype**.
- Name**: A text area containing the text "Glob: MFRAME" and "Glob: RPOOL". To the right of the text area are up and down arrow buttons. Below the text area is a single-line text input field.
- Type**: A section with three radio buttons: **Global**, **Page** (selected), and **Instance**.
- Buttons**: At the bottom right, there are three buttons: **Reset**, **Cancel**, and **OK** (which is highlighted with a double border).

Once a page fusion set exists, all the techniques you have just used for global fusion sets will work on it exactly as they would if the set were a global set.

### Creating a Page Fusion Set

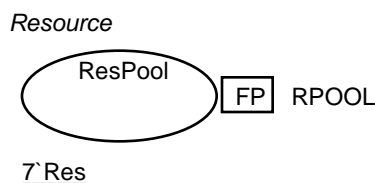
We could now delete **RPOOL** and create a page fusion set in its place. But that would illustrate nothing new. Let's do something more interesting: convert **RPOOL** directly from a global fusion set to a page fusion set:

- Choose **Fusion Place** from the **CPN** menu.

- Choose Rename/Retype under Fusion Set.
- Click on **Global: RPOOL** in the box under **Name**.
- Choose **Page** under **Type**.
- Click **OK**.

The dialog disappears. **RPOOL** is now a page fusion set: the two **ResPools** on **Resmod#1** are functionally one place, and the two on **New#2** are functionally a second place. In other words, the page fusion set **RPOOL** consists of two page fusion subsets, one for each page that contains members of the set.

All of the places in a page fusion set look the same:



Note the change from **FG** to **FP**, indicating the change from a global fusion set to a page fusion set. There is nothing in the place's appearance to indicate which page fusion subset the place belongs to: its presence on the particular page is indication enough.

As you might expect, the members of a page fusion set on any one page must agree as to initial marking, but the members on different pages can have different initial markings. Members of a page fusion set on different pages have no more relation to each other than they would if they were members of different global fusion sets.

## Watching Fusion in Action

By now you should have a fairly good idea of how fusion pages work. But there's nothing like seeing it in action to make its properties clear. Let's go into the simulator and execute the net.

- Enter the simulator.
- Save the net in the **NewTTDiagrams** directory. Then saved net can be used to make future entries to the simulator much faster, and you won't have to load the ML information again.

The diagram you have been working with already has a breakpoint between steps and occurrence set parameters set to 100%. These are the right parameters for observing this net. However you will need

to adjust the various simulation regions that show place markings so you can see clearly how the markings change during simulation.

- Adjust the simulation regions on both pages.
- Start simulation.

Continue simulation through several steps. Between steps, look at the various copies of `Resnet`, with particular attention to their `Processing` and `ResPool` places. Since resources are put back into `ResPool` as soon as they are used, with time stamps to mark them as unavailable, you will need to compare time stamps rather than token counts when comparing `ResPool` markings.

Verify by observation that:

1. All four `Processing` places always have the same marking.
2. The two `ResPool` places on `Resmod#1` always have the same marking.
3. The two `ResPool` places on `New#2` always have the same marking.
4. The markings of the `ResPool` places may differ between `Resmod#1` and `New#2`. (When they look the same it is just a coincidence.)
5. The markings of the places `Waiting` and `Using` may differ among the various copies of `Resnet`, both on the same page and between pages. (When they look the same it is just a coincidence.)

If you understand why these five properties must hold, you understand fusion places. If you do not, be sure to not to proceed until you have studied this chapter further and know how fusion places work.

## Instance Fusion Sets

In the simulation you just ran, there were two identical pages. Two copies of a page is not that bad, but suppose we want to model a situation in which there are many hundreds or thousands of instances of the same entity, and that this entity is too complex to represent with a token but must be modeled with a piece of net structure? We would not want to put hundreds or thousands of copies of the same structure on a page, or worse yet, have hundreds or thousands of copies of the same page. Either method would render the net completely unmanageable.

To deal with such situations, Design/CPN allows you to create a single page in the editor, and then use that page as many times as needed in the simulator. This allows us to have it both ways: physically there is only one page, but functionally there can be as many copies of the page as we need. This capability is called *multiplicity*, and the different functional copies of the same page are called *page instances*.

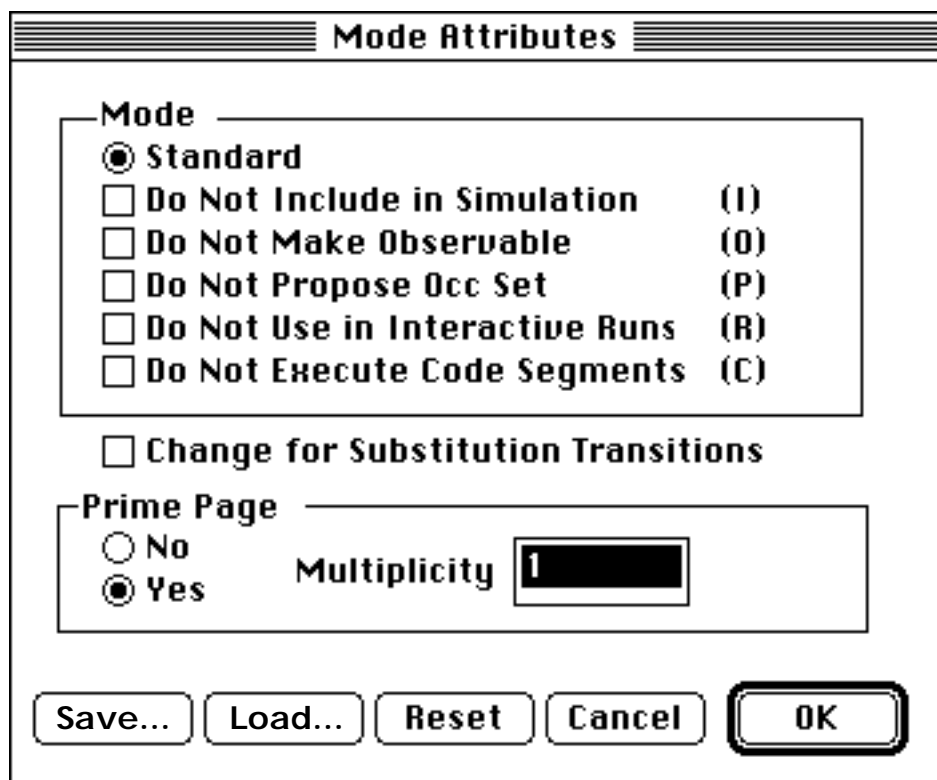
As we noted above, completely disjoint copies of the same net structure would serve no purpose, since none would do anything the others do not do. The same argument applies when the copies are multiple instances of a page: the instances must be interconnected in some way to form a larger whole, or there is no reason to have them. As with physically separate pages, the interconnection is accomplished by using fusion places.

### Creating Multiple Page Instances

Suppose that we need a third copy of Resmod#1. We could use **Load Subdiagram** again, resulting in a third identical page. But by creating a second instance of the page, rather than loading in another copy, and then using fusion appropriately, we can produce exactly the same functional effect while avoiding the overhead that a third physical copy would entail.

- Open the hierarchy page.
- Select the page node for Resmod#1.
- Choose **Mode Attributes** from the **Set** menu.

The **Mode Attributes** dialog appears:



The dialog box is titled "Mode Attributes". It contains two main sections: "Mode" and "Prime Page".

**Mode**

- ☒ Standard
- ☐ Do Not Include in Simulation (I)
- ☐ Do Not Make Observable (O)
- ☐ Do Not Propose Occ Set (P)
- ☐ Do Not Use in Interactive Runs (R)
- ☐ Do Not Execute Code Segments (C)

☐ Change for Substitution Transitions

**Prime Page**

- ☐ No
- ☒ Yes

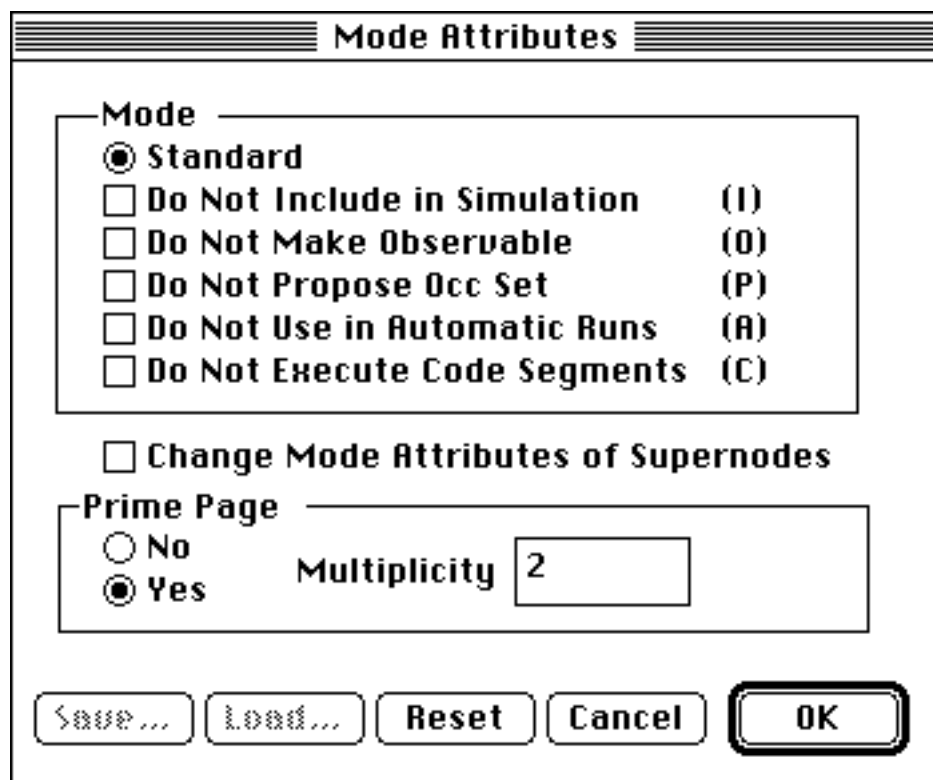
**Multiplicity**

Buttons: Save... Load... Reset Cancel OK

Note the figure for **Multiplicity**. It is 1, indicating that there is only one instance of Resmod#1. The number in the **Multiplicity** edit box is already selected, so:

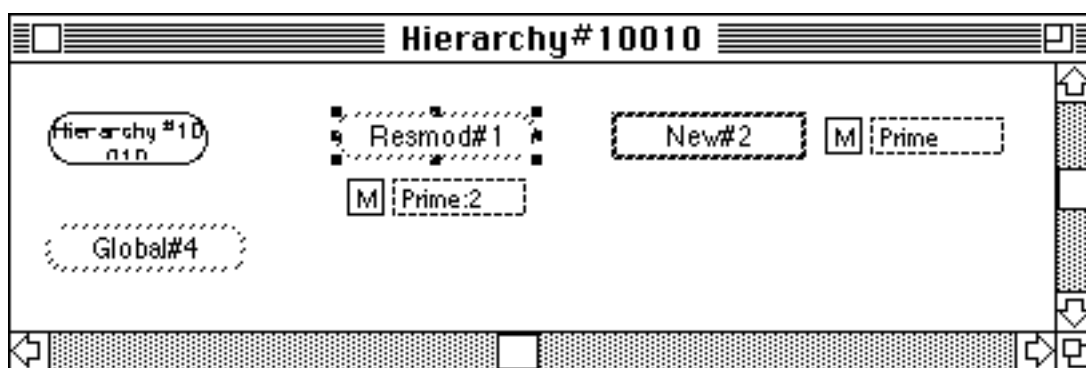
- Type "2".

The dialog should now look this:



- Click **OK**.

There are now two instances of Resmod#1. The hierarchy page now looks as follows:



Note the new designation Prime:2 for Resmod#1. The "2" indicates that there are now two instances of the page.

### Multiplicity and Fusion

Let's take a moment to look at the overall structure of the diagram, with particular attention to relationships between fusion sets and multiple page instances.



The diagram ResourceModel now has two executable pages: Resmod#1 and New#2. These two pages are identical. There are furthermore two instances of Resmod#1. These are of course identical with each other and with New#2. There are physically four copies of Resnet, two on each page; but functionally there are six copies, because each copy on Resmod#1 exists twice, once on each instance of the page.

There are two fusion sets: **MFRAME** and **RPOOL**. **MFRAME** is a global fusion set, so it equates all six copies of the place **Processing**: the two on the first Resmod#1 instance, the two on the second, and the two on New#2. **RPOOL** is a page fusion set. It equates the two copies of ResPool on New#2 into one fusion subset. But what should it do with the ResPools in the two instances of Resmod#1?

There are two possibilities, equally reasonable:

1. Form a single fusion subset that includes all instances of the page, on the grounds that there is really only one page and a page fusion set, by definition, equates all constituent places on each page.
2. Form a separate fusion subset for each instance of the page, on the grounds that the instances are so much like separate pages that they deserve separate fusion subsets.

In practice, there are situations where the first method would more useful, and situations when the second would be. Therefore Design/CPN allows both possibilities. Page fusion sets conform to the first possibility: there is one page fusion subset for all instances of a page. To create a separate fusion subset for each instance of a page, we use the third type of fusion set: the *instance fusion set*.

### Working With Instance Fusion Sets

Instance fusion sets are identical with page fusion sets in every way but one: while a page fusion set equates every constituent place irrespective of page instance, an instance fusion set is divided into subsets, each of which equates only constituent places that are on the same page instance.

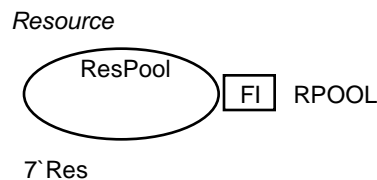
In other words, an instance fusion set is a collection of fusion sets that have the same name but exist on different page instances. The members of such a collection are called *instance fusion subsets*, and each constituent place is called an *instance fusion place*.

Instance fusion sets are created and modified much as global and page sets are. The only difference occurs at the beginning: when you first create an instance fusion set, select **Instance** in the **Type** section of the **Fusion Place** dialog.

Let's create an instance fusion set for the `ResPools` of Resmod#1, so that each instance of the page will have its own resource pool. If there were currently no fusion set for the Resmod#1 `ResPools`, you could select both of them and create a instance fusion set just as if you were creating a page or global fusion set. Since we already have a page fusion set, **RPOOL**, let's just change its type, as we did when we converted it from a global to a page fusion set.

- Make Resmod#1 the current page.
- Choose **Fusion Place** from the **CPN** menu.
- Choose **Rename/Retype** under **Fusion Set**.
- Click on **Page: RPOOL** in the box under **Name**.
- Choose **Instance** under **Type**.
- Click **OK**.

**RPOOL** on Resmod#1 is now an instance fusion set. Note the change in the appearance of the `ResPool` places:



Now take a look at the `ResPools` on New#2. They have not changed: **RPOOL** is still a page fusion set for New#2. It might seem more consistent if **RPOOL** changed everywhere, but in practice a per-page granularity for converting page to instance fusion sets turns out to be more convenient.

## Observing Fusion Across Multiple Instances

Let's re-enter the simulator, execute the model, and take a look at how instance fusion sets look in action.

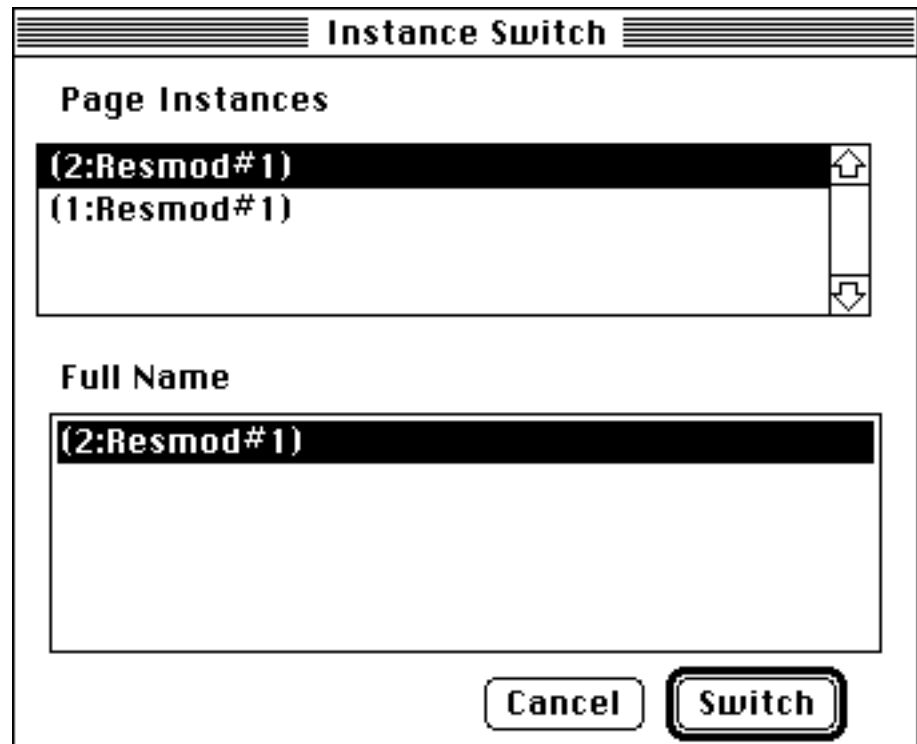
- Re-enter the simulator.

You can examine New#2 as you did before. But what about Resmod#1? There is only one page, and hence only one window, but there are two instances, and it will be useful to examine each one of them.

To switch between instances of a page, use the **Instance Switch** dialog. To activate it:

- Depress the SHIFT key.
- Click the mouse on the title bar of Resmod#1.

You can get the same effect by choosing **Select Instance** from the **Sim** menu. In either case, the **Instance Switch** dialog appears:



All the instances of Resmod#1 are listed under **Page Instances**. The instance currently on display in the Resmod#1 window is highlighted. Note that the highlighted name matches the name in the Resmod#1 window's title bar. The title bar of a window that displays a page with multiple instances always tells which instance is currently visible.

- Use the mouse to select the other instance of Resmod#1.
- Click **Switch**.

The dialog disappears. The title bar of the Resmod#1 window now indicates that the other instance is on display. You won't see any other change, because both instances of the page are in the same initial state. Once simulation begins they will diverge.

- Start simulation.

Continue simulation through several steps. Switch back and forth between the two instances of Resmod#1, and verify that both `ResPool` places on each instance always have the same marking,

but that the markings may be different on the two instances. (When they look the same it is just a coincidence.) Everything else relating to fusion is the same in the current simulation as it was in the previous one. If you have any doubts, verify by observation that this is so.

# Chapter A3

## Substitution Transitions

In the nets we have looked at so far, each transition has been a fundamental unit: there was no functional significance to a transition except that defined by any associated places, arcs, arc inscriptions, guard, code segment, and/or time region.

Another form of transition is possible in a CP net. We can establish a method by which a transition can stand for an entire piece of net structure, so that the net containing the transition executes as if the logic that the transition represents were physically present at the location of the transition. Such a transition is called a *substitution transition*.

Substitution transitions add nothing fundamentally new. Everything that can be done with them can also be done by using fusion places, which as we have seen add only convenience, not power, to a CP net. But like fusion places, substitution transitions, add so much convenience that they can make the difference between feasibility and total impossibility.

When a CP net uses a substitution transition, the logic that the transition represents must be kept somewhere. It is kept on a page called a *subpage*, and the logic on the subpage is called a *subnet*, or sometimes a *submodel*. The page that contains the substitution transition is called a *superpage*. Superpages and subpages are connected by equating places on the two pages using special-purpose fusion sets. A place that belongs to such a fusion set is called a *port* if it is on a subpage, and a *socket* if it is on a superpage.

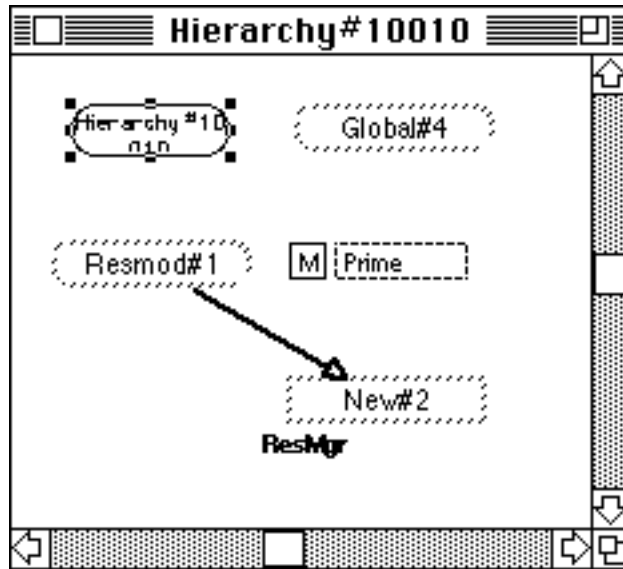
It would be inconvenient to have to create substitution transitions by manually creating the requisite fusion sets, though it could be done. Design/CPN provides extensive capabilities for facilitating the creation and use of substitution transitions. This chapter tells you what those capabilities are, and shows you how to use substitution transitions to create hierarchical CP nets.

### Structure of a Diagram With Substitution

Before we begin creating substitution transitions, let's take a look at a simple diagram that already contains one.

- Open the diagram ResmodSubtrans, in the TutorialDiagrams directory.

The diagram's hierarchy page appears:



The page Global#4 is the same page of global declarations that is used in ResourceModel, the net you worked with in the previous chapter, so we will ignore it for now and consider the other pages. Let's start with the hierarchy page itself.

### The Hierarchy Page

We can tell a lot about this (or any) diagram's structure just by looking at its hierarchy page. The ResmodSubtrans hierarchy page shows that ResmodSubtrans contains two executable pages, Resmod#1 and New#2. The arrow linking the two page nodes indicates that Resmod#1 contains a substitution transition representing net structure that is kept on New#2. When two pages are related in this way, the page that contains the substitution transition is called a *superpage*, and the page that contains the net structure that the transition represents is called a *subpage*. The net structure on a subpage is sometimes referred to as a *subnet* or a *submodel*.

Resmod#1 is a prime page, so the simulator will execute it. New#2 is not a prime page, but it does not have to be: when a superpage is prime, all subpages that it uses are automatically included in simulation.

## Substitution Transitions

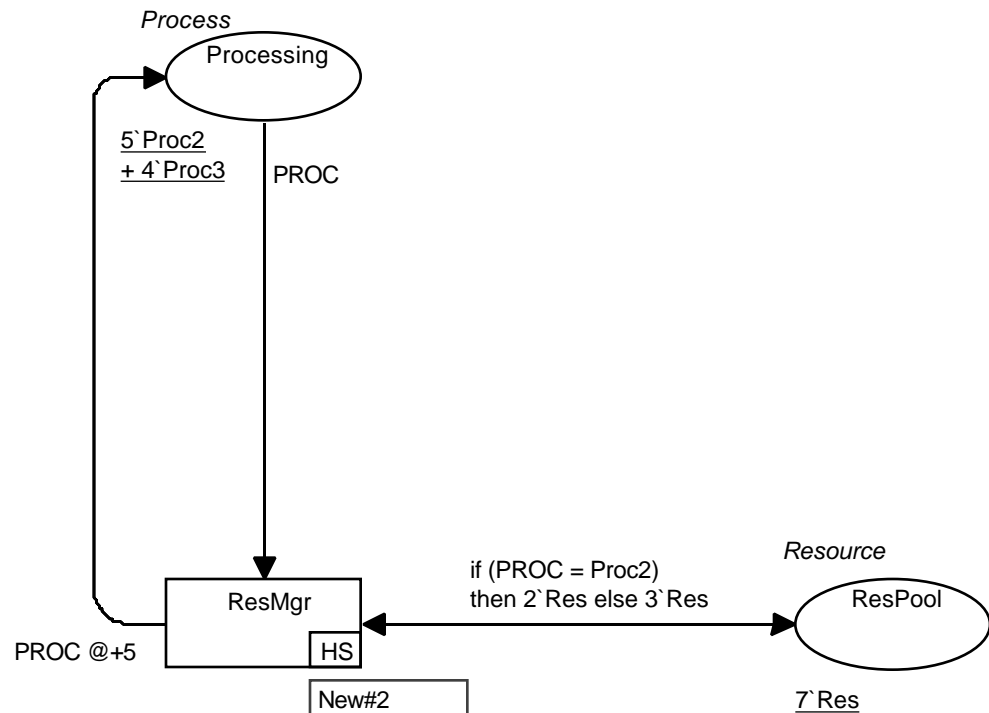
Below the node for New#2 the string “ResMgr” appears, in a region called a *substitution tag region*. This indicates that the substitution transition in Resmod#1 has a name: ResMgr.

### The Superpage Resmod#1

Now let's look at the superpage:

- Open Resmod#1.

The page looks like this:



Doubtless this has a familiar look. Processing and ResPool are the same as they were in the ResourceModel diagram with which you worked in the previous chapter. However, in place of the rest of the net there is a substitution transition named ResMgr.

Note that this page is, in and of itself, a functional net. If ResMgr were an ordinary transition, and everything else were as shown, you could take this net into the simulator and execute it. Its behavior would be a simplified version of the behavior of Resnet. Take a moment to execute the net mentally, treating ResMgr as an ordinary transition, and notice that there is nothing unusual about what the net does.

Look at the Status Bar. It describes the place ResPool, which is currently selected, as a Place, I/O-Socket. This indicates that

`ResPool` is a socket: somewhere, on some other page (by definition a subpage), there is another place, a port, associated with it.

- Select the place `Processing`, and look at the status bar.

It too is a Place, I/O-Socket; that is, a socket, so it too has a port somewhere on a subpage.

### The Subpage New#2

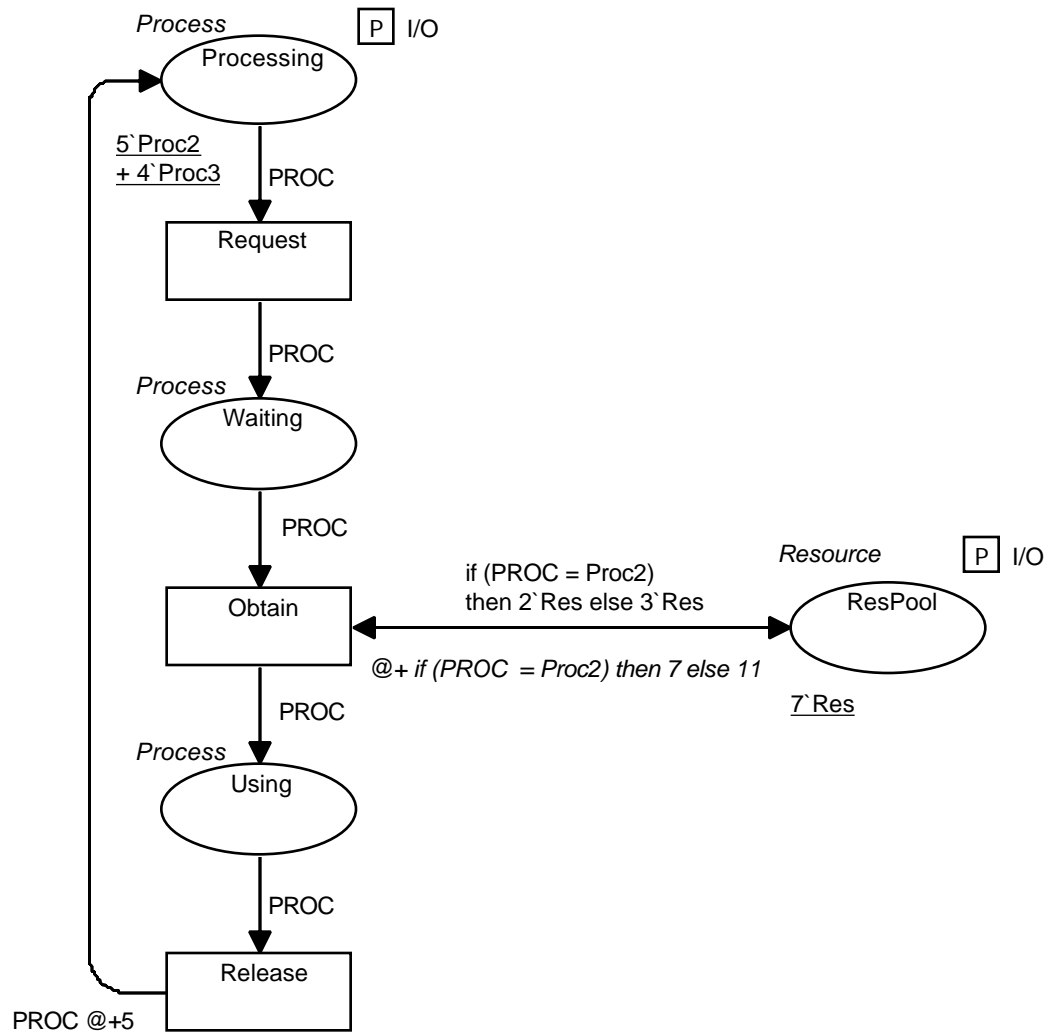
Now let's take a look at what the sockets on the superpage Resmod#1 are connected to.

New#2, as we saw on the hierarchy page, is the subpage for the substitution transition `ResMgr` on Resmod#1, so the ports must be on New#2. To get to New#2, you could reopen the hierarchy page, then open New#2 like any other page. But there is a more convenient method. By double-clicking on a substitution transition, you can jump directly to the subpage that it represents.

- Double-click on the substitution transition `ResMgr`. (Don't click directly over its name or nothing will happen, because the name region will intercept the clicks.)

The subpage New#2 opens just as if you had navigated to it via the hierarchy page. It looks like this:





This is essentially Resnet, the net that you coped with so many copies of in the previous chapter. But there is one obvious difference: the box P and the string “I/O” next to `Processing` and `ResPool`.

The P is called a *port key region*. It indicates that the associated place is a port. The “I/O” is contained in a region called a *port region*. The “I/O” means that the ports have both input arcs and output arcs connecting to them on the subpage.

Look at the Status Bar. It describes the place `ResPool`, which is currently selected, as a Place, I/O-Port. This too indicates that `ResPool` is a port: somewhere, on some other page (by definition a superpage), there is another place, a socket, associated with it.

- Select the place `Processing`, and look at the status bar.

It too is a Place, I/O-Port; that is, a port, so it too has a socket somewhere on a superpage.

## Ports and Sockets

We know now that there are two sockets on the superpage Resmod#1, and two ports on the subpage New#2. How can we tell what port is equated with what socket? In this case it is easy: they have the same name. The `Processing` places on the two pages are one port-socket pair, and the `ResPool` places are another. If the names did not match, Design/CPN would show additional information to tell you what is equated with what, as we will see later in this chapter.

A port-socket pair is nothing more than a two-member fusion set. Port-socket pairs are what link superpages and subpages so as to implement substitution transitions. Therefore substitution transitions are in practice just a specialized application of fusion sets.

## Jumping Directly to a Superpage

You can jump directly from a subpage to its superpage, bypassing the hierarchy page, by double-clicking on any port on the subpage.

- Double-click on either port on New#2 (avoiding its name region).

The superpage Resmod#1 becomes the current page.

## Overall Structure of the Diagram

We have now looked at all the parts of this diagram. How do they fit together? What do they add up to?

You may already know the answer: this diagram is functionally identical with ResourceModel, the net we worked with in the previous chapter. The change from a flat to a hierarchical implementation has made no functional difference at all.

Before you proceed, be sure that you understand why this is so. If you don't see it, imagine doing the following:

1. Replace each port-socket pair in ResmodSubtrans with a two-member fusion set.
2. Physically move the net structure on New#2 to Resmod#1, leaving New#2 empty.
3. Collapse the two fusion sets on Resmod#1 into single places, as described in the previous chapter.

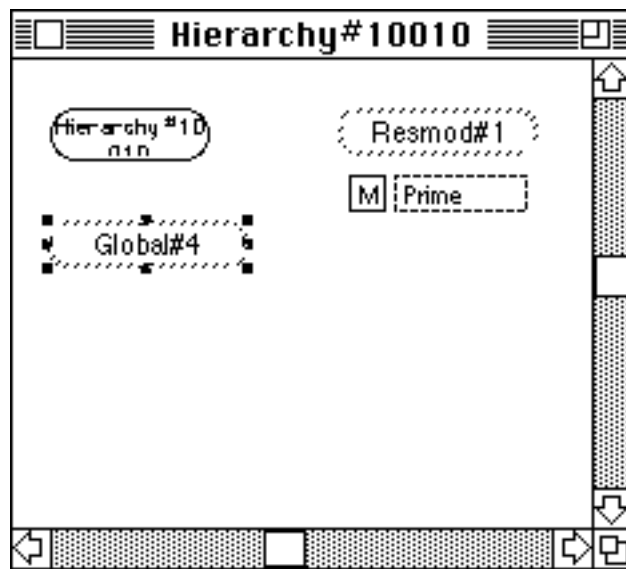
The net on Resmod#1 is now structurally identical to Resnet in the ResourceModel diagram. This proves that ResourceModel and ResmodSubtrans contain functionally identical nets.

### Creating a Substitution Transition

To study the creation of substitution transitions, with their attendant superpages and subpages, let's convert Resnet to hierarchical form and then restore it to flat form.

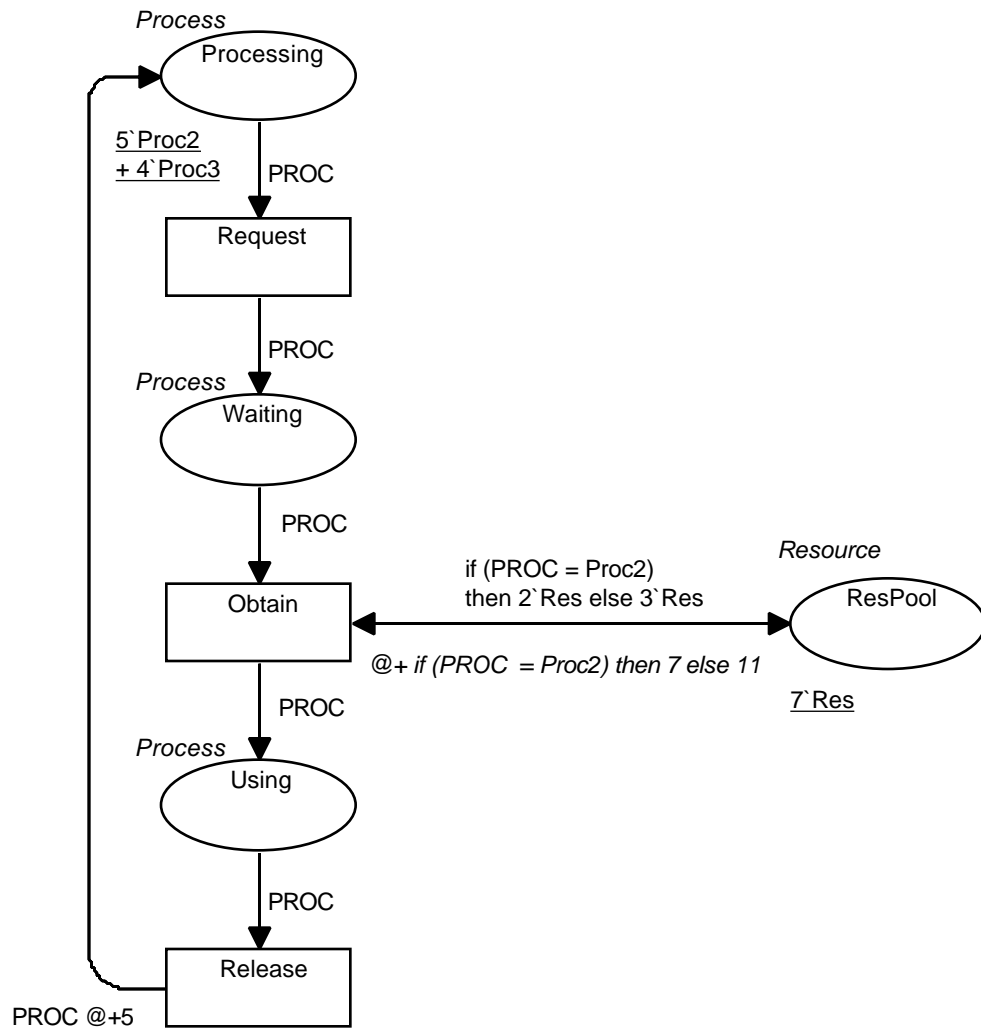
- Open the diagram ResourceModel, in the TutorialDiagrams directory.

The diagram's hierarchy page appears:



- Open the page Resmod#1.

The page looks as follows:



The simplest way to create a substitution transition is to move part of an existing net to a subpage, leaving a superpage behind. The steps in this process are:

1. Designate which net component(s) to move to the subpage.
2. Initiate the process of subpage creation.
3. Specify where on the superpage you want the substitution transition to appear.
4. Name the substitution transition (if desired).

Let's look at these steps in detail, and execute them on Resnet.

### Designate the Net Components to Move to the Subpage

This is just a matter of selecting the component(s) to move. If you want to move a single transition, click the mouse on it so that it becomes the selected object. If you want to move a larger piece of net structure, form a group that contains all of the constituent nodes.

- Select the nodes `Request`, `Waiting`, `Obtain`, `Using`, and `Release`. Be sure no other nodes are selected: the status bar should display: Group of 5 Nodes.

### Initiate Subpage Creation

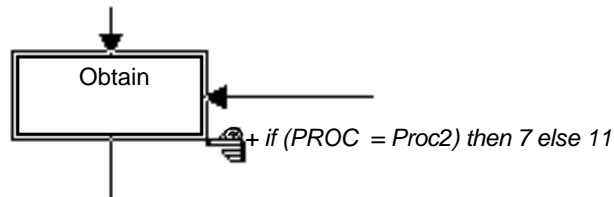
- Choose **Move to Subpage** from the **CPN** menu.

### Specify the Substitution Transition's Location

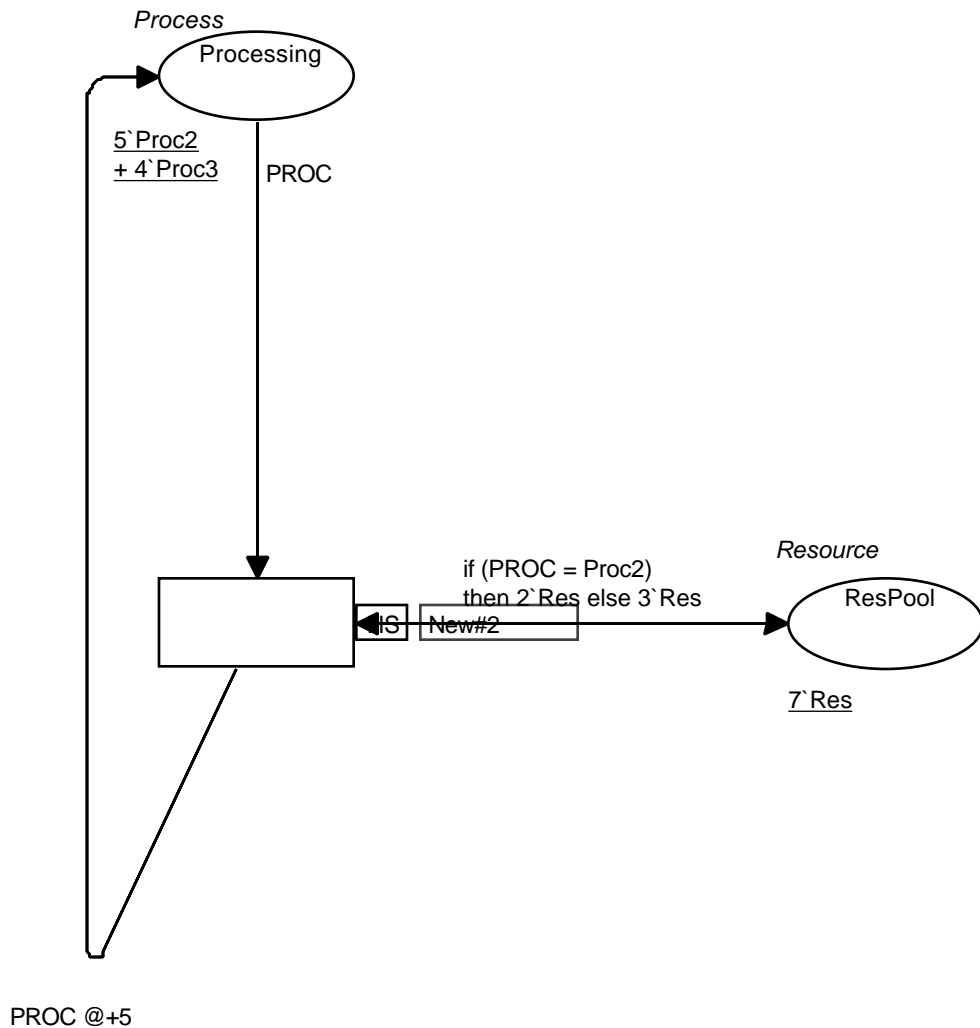
Design/CPN enters substitution transition creation mode. This mode is similar to ordinary transition creation mode, except that the result is a substitution transition.

- Use the mouse with the SHIFT key to create a transition that just surrounds the existing transition `Obtain`.

Just before you release the mouse button, `Obtain` should look like this:



When you have finished creating the transition, Design/CPN moves all the nodes you selected, and any associated regions, to a new page that it creates for this purpose. That page is a subpage, and the page you have been working on is a superpage. The superpage now looks like this:



The transition is now a substitution transition. To indicate this, a region called the *hierarchy key region* has been created. This consists of a box, **HS**. The box is currently partly obscured by the arc to `ResPool`. The next section shows you how to move this region to a better location.

Next to the hierarchy key region is another region, called the *hierarchy region*, containing the text `New#2`. This region indicates that the net structure that the substitution transition stands for is on the page `New#2`. It is a popup region, so you can make it appear and disappear by double-clicking on the key region.

### Name the Substitution Transition (If Desired)

A substitution transition does not have to have a name. If it does not, three dots will appear on the hierarchy page by the node for the transition's subpage, where the transition's name would otherwise

be displayed. But it is generally a good idea to provide a name, particularly if a diagram is complex.

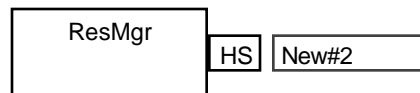
When you create a substitution transition via **Move to Subpage**, Design/CPN automatically enters text mode. You could name the substitution transition immediately by typing in text, but you would not be able to reposition the text afterwards to improve its appearance, so:

- Leave text mode.

Naming a substitution transition is exactly like naming any other transition: select the transition, then use the **CPN Region** command from the **CPN** menu.

- Name the substitution transition `ResMgr`. Position the name region in the top center of the transition.

The substitution transition should now look like this (with the hierarchy regions obscured by an arc):



### Status of the Diagram

The diagram you have been working on, ResourceModel, has now been converted to a hierarchical diagram with one subpage and one superpage. It is now structurally identical with the diagram ResmodSubtrans, which you examined earlier in this chapter. However it is not identical in appearance: its appearance leaves much to be desired. The next section shows how its appearance may be improved.

## Improving the Net's Appearance

The appearance that has resulted from creating the substitution transition is obviously not ideal. Let's improve the appearance of the superpage, and then look at and improve the subpage and hierarchy page also.

### Improving the Superpage's Appearance

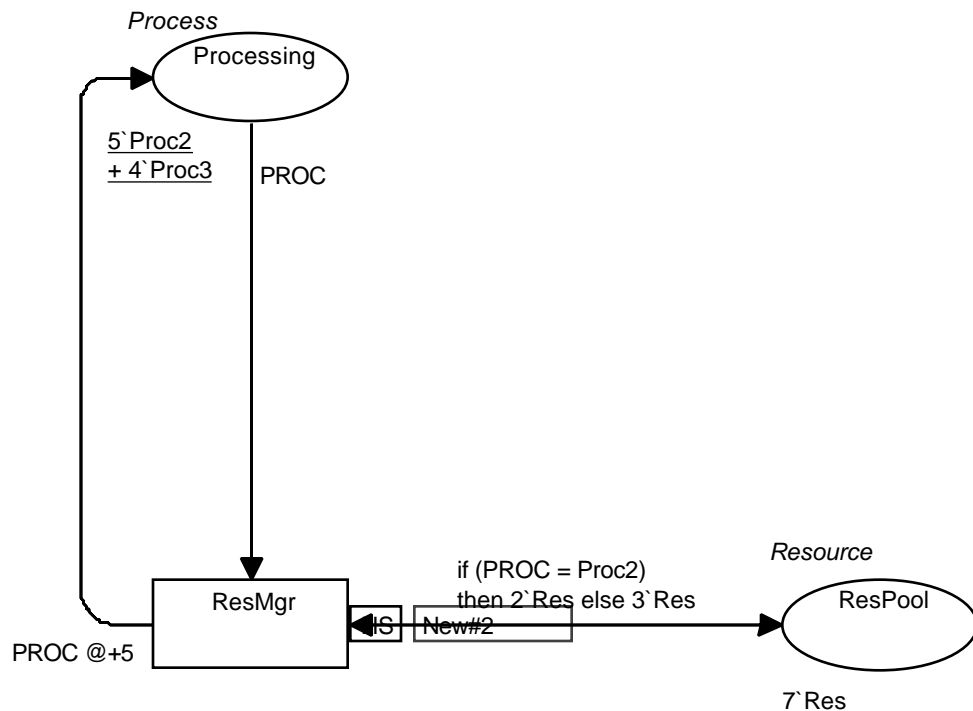
It is rare for a superpage to have an ideal appearance from the first. Usually some tuning is appropriate, as in the current case. Two changes would obviously be useful:

1. Reroute the arc that runs from the new substitution transition to *Processing*.
2. Move the hierarchy key region and hierarchy region so that the arc does not obscure them.

### Rerouting the Arc

- Position the mouse pointer over the adjustment point of the sharp corner of the arc (towards the bottom of the window).
- Depress the mouse button.
- Move the adjustment point so that the arc follows a right angle (no “jaggies” in the adjacent segments).
- Release the mouse button.
- Use the mouse to move the arc inscription until it is just below the bottom segment of the arc.

The superpage should now look like this:



### Moving the Regions

Moving the regions presents a small problem: they are so small that it is difficult to grab onto them with the mouse without getting the



arc instead. The hierarchy key region is particularly hard to get hold of.

To deal with situations like this, Design/CPN provides commands in the **Makeup** menu that allow you to select and move objects without using the mouse.

- Select the substitution transition.
- Choose **Child Object** from the **Makeup** menu.

The transition has only one child object, the hierarchy key region, so the region becomes selected. (If there were more than one child object, and **Child Object** did not select the needed object, you could use **Next Object** and **Previous Object** to select among the child objects.)

So much for selecting the region. In order to move it:

- Choose **Drag** from the **Makeup** menu.

The editor is now in drag mode. The mouse pointer becomes the drag tool.

- Position the hierarchy key region in the lower right corner of the substitution transition.

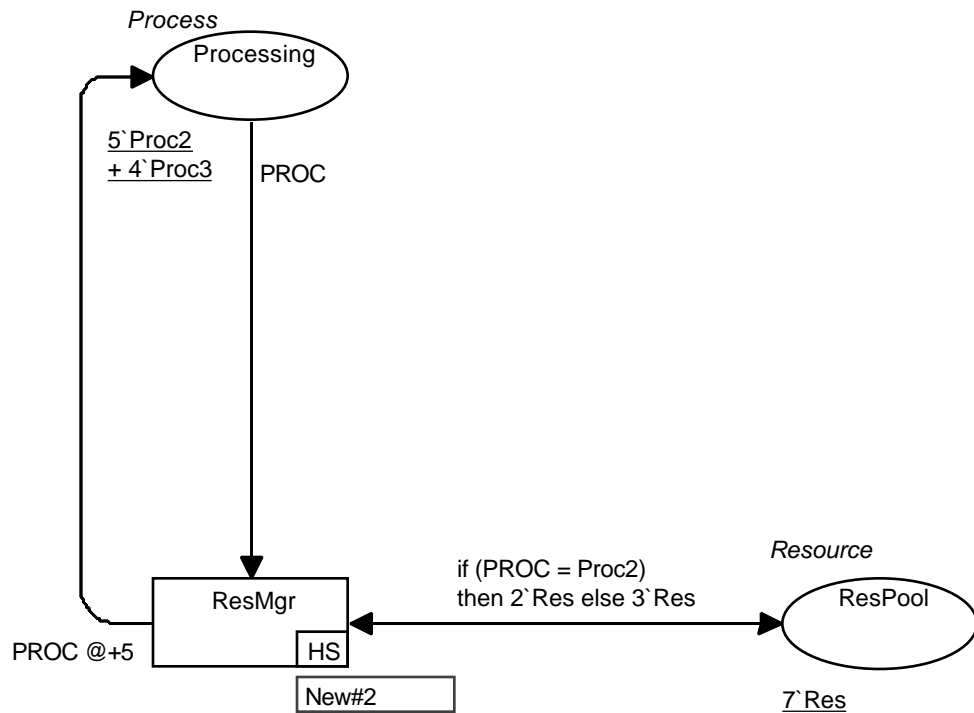
To exit drag mode:

- Press ESC.

The editor returns to graphics mode. The hierarchy region is still a bit close to the arc, but it is far enough away to position directly with the mouse:

- Position the hierarchy region so that it is directly below the key region, extending off to the right.

The superpage should now look like this:

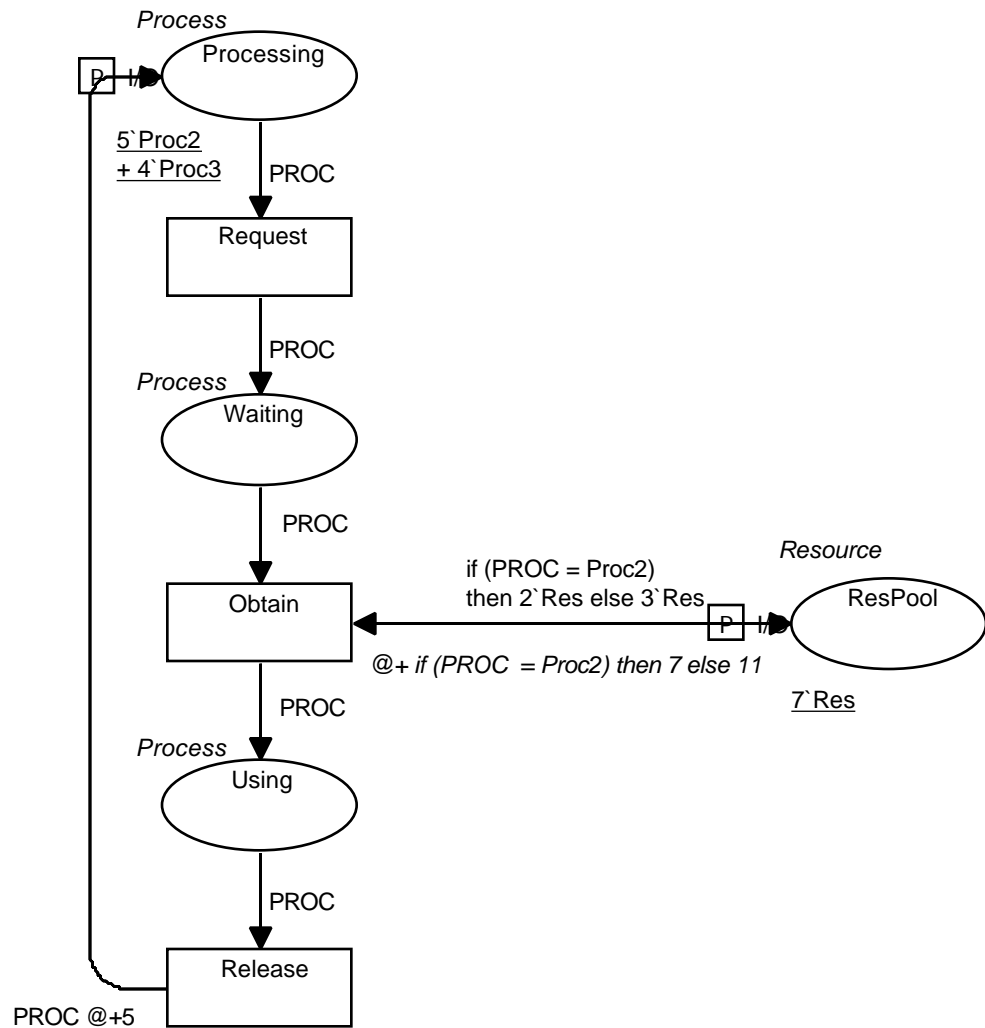


### Improving the Subpage's Appearance

A subpage created by moving part of an existing page usually needs little appearance tuning, but sometimes improvements can be made.

- Double-click on the substitution transition ResMgr.

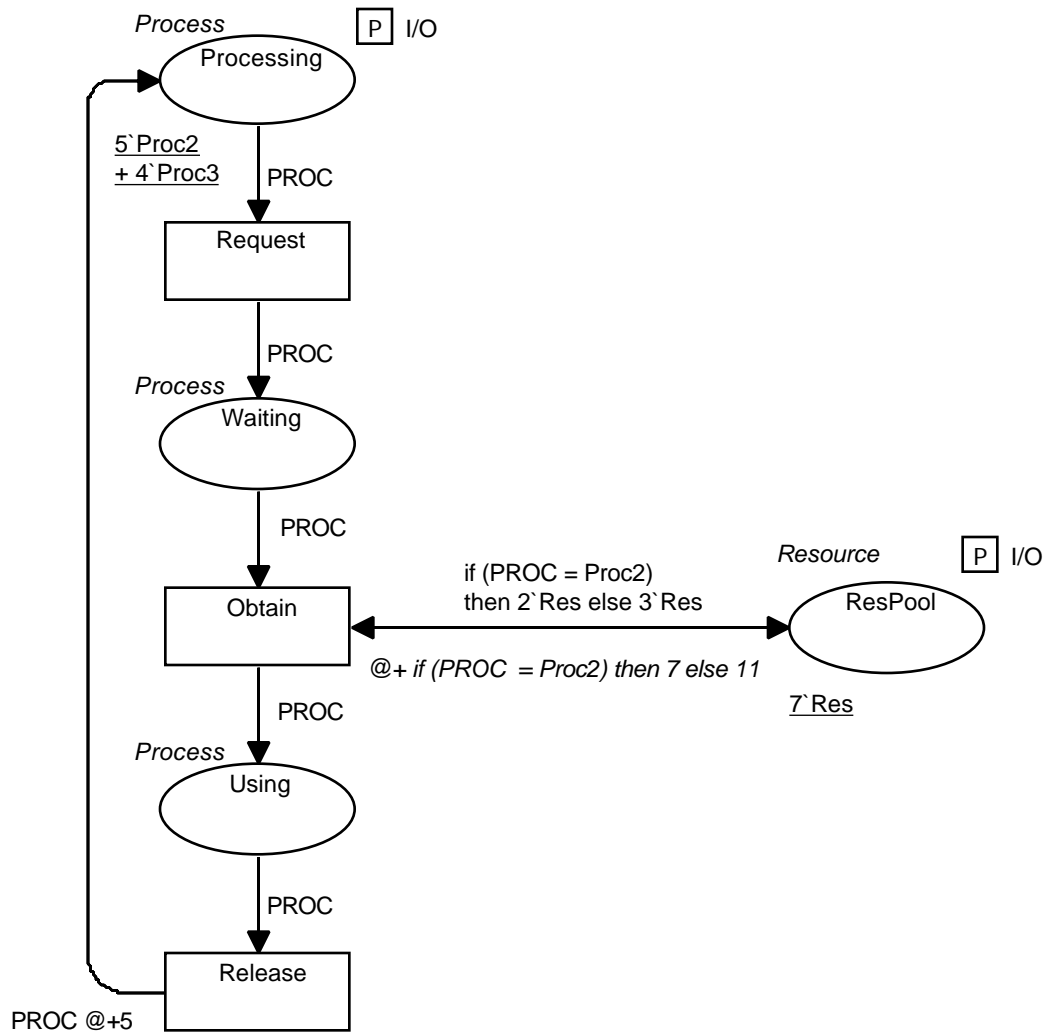
The subpage appears:



This page looks pretty good except for the placement of the port regions. Let's move them to more felicitous locations:

- Use **Child Object**, **Next Object**, **Previous Object**, and **Drag**, from the CPN menu, to position the port regions at the upper right edge of their places.

The subpage should now look like this:

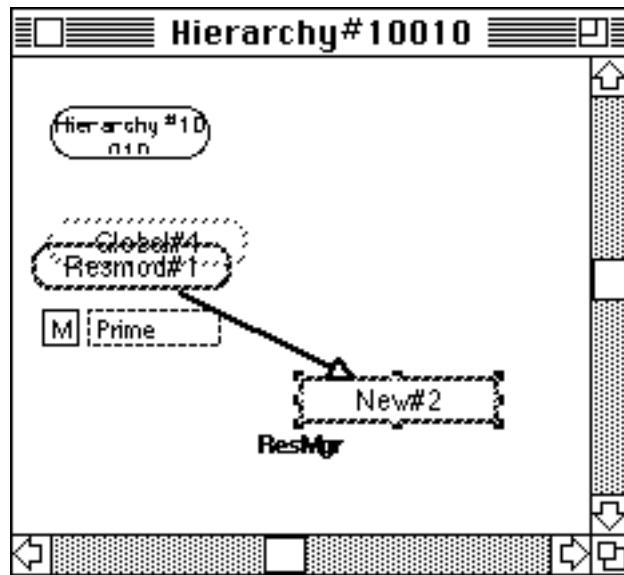


### Improving the Hierarchy Page's Appearance

When you create a substitution transition, Design/CPN automatically updates the hierarchy page. It does not attempt to do this intelligently, because such an attempt could destroy a hand-crafted hierarchy page organization that could not have been produced algorithmically.

- Open the hierarchy page.

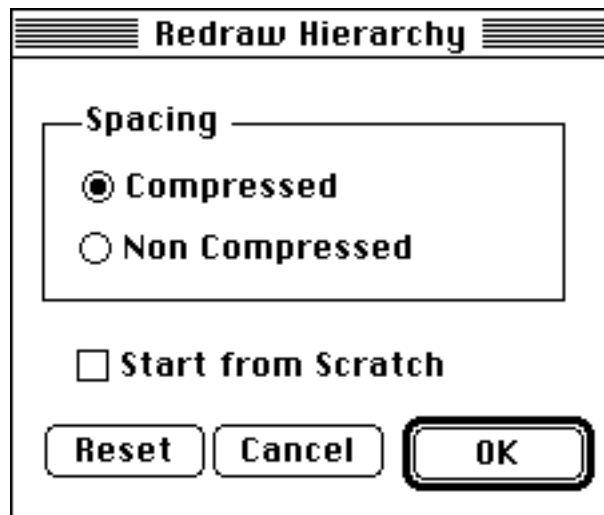
The page appears:



Obviously this is not an acceptable layout. But we don't have to improve it by hand. Design/CPN can do the job automatically.

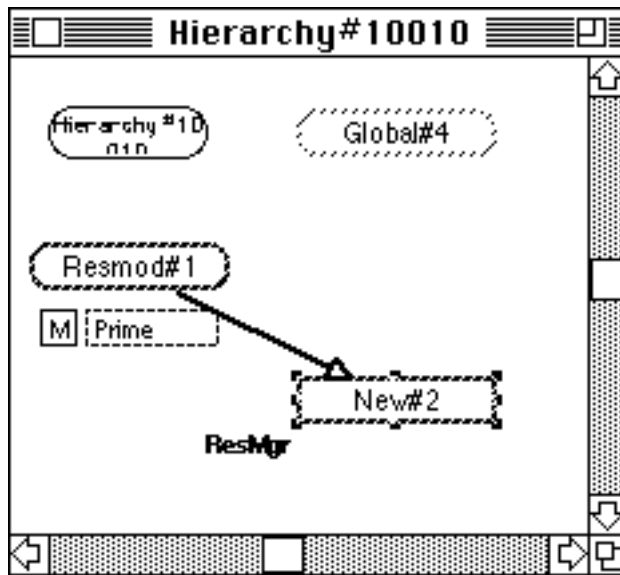
- Choose **Redraw Hierarchy** from the **Page** menu.

The **Redraw Hierarchy** dialog appears:



- Click **OK**.

The dialog disappears. Design/CPN redraws the hierarchy page:



Clearly this is an improvement.

### Status of the Diagram

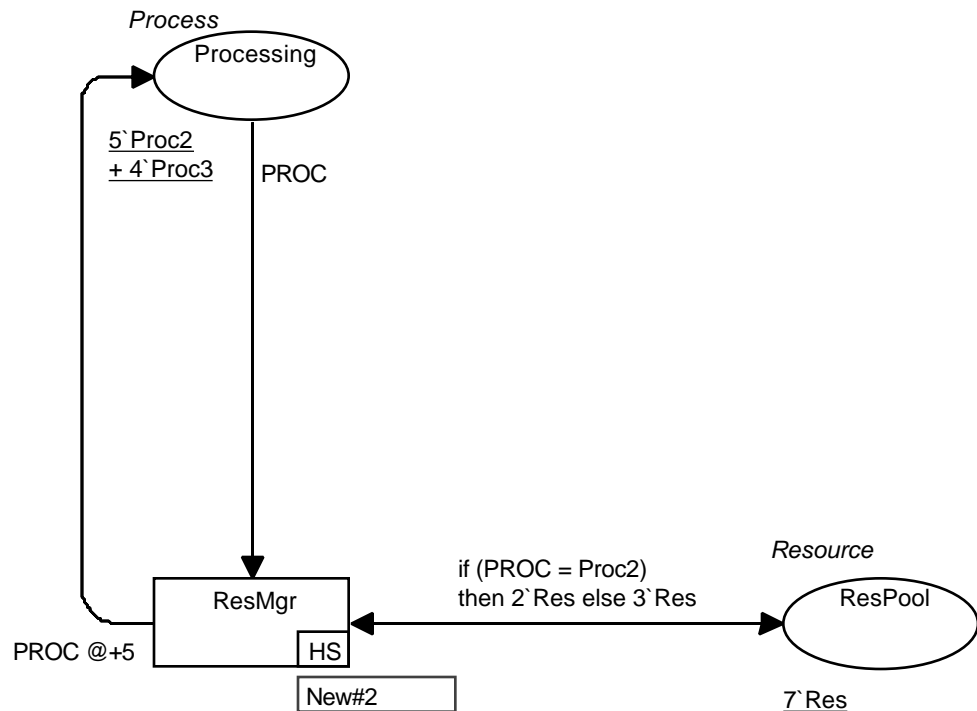
The diagram you have been working on, ResourceModel, is now not only structurally the same as ResmodSubtrans; it is, except for minor variations, identical in appearance as well. The steps you have just performed were in fact exactly the steps by which ResmodSubtrans was created.

## Reversing Substitution Transition Creation

As a model evolves, it sometimes happens that a substitution transition that was once useful ceases to be so. When this occurs, you can reverse the creation process, copying the subpage into the superpage and eliminating the substitution transition. Let's see how this works.

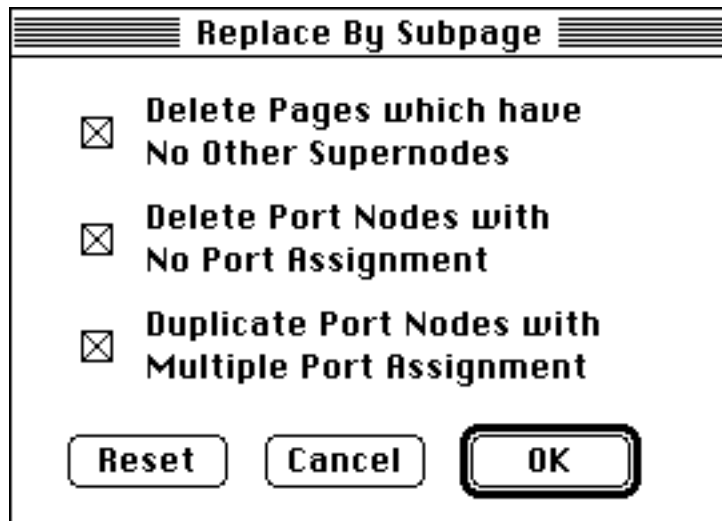
- Open the superpage Resmod#1.

The superpage appears:



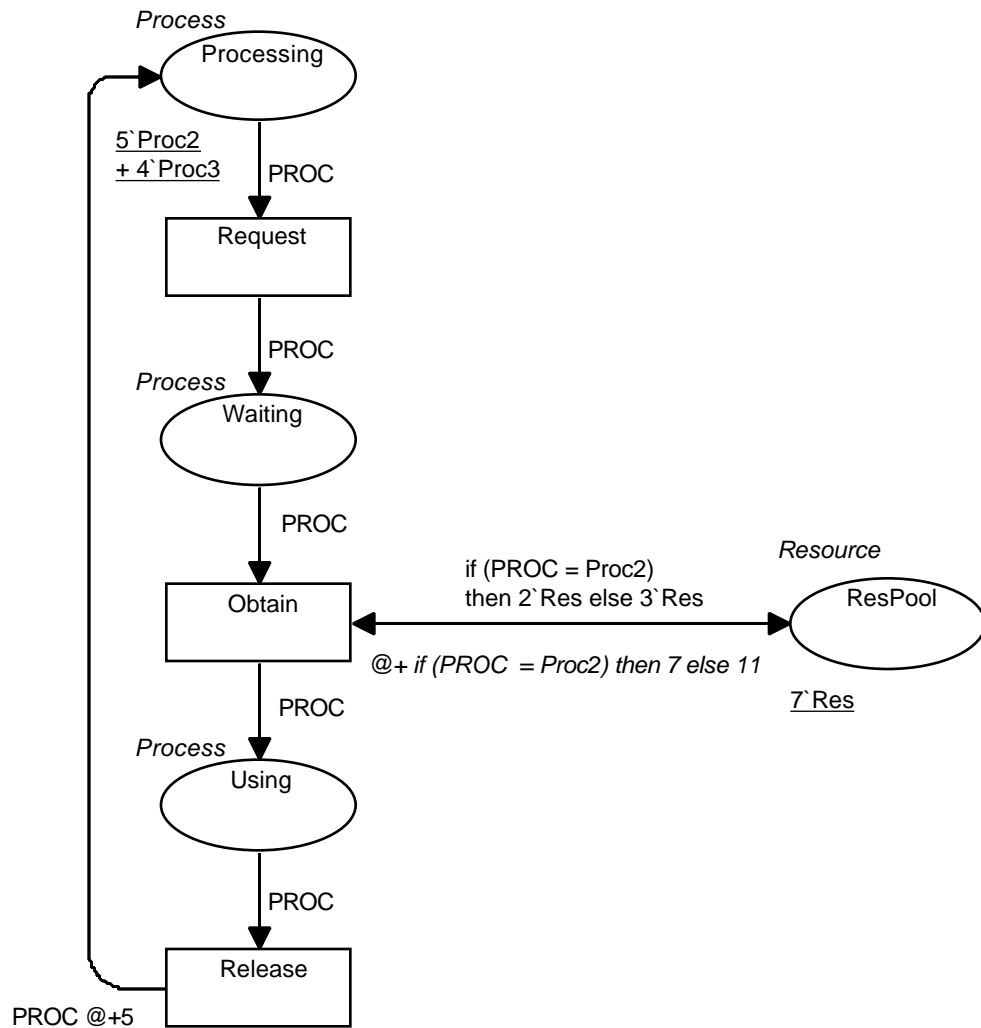
- Select the substitution transition **ResMgr**.
- Choose **Replace by Subpage** from the **CPN** menu.

The **Replace By Subpage** dialog appears:



- Click **OK**.

The dialog disappears. Design/CPN copies the subpage back into the superpage. The result is:



### Status of the Diagram

The diagram is now exactly as it was before you created the substitution transition. It is not structurally or functionally different in any way for having been split into two hierarchical pages and then re-assembled. The two operations are exactly inverse.

In this case, the original appearance of the page was restored when the subpage and superpage were reassembled, but in general this will not be the case. It happened here only because the subpage was originally extracted from the superpage, and we did very little graphical editing while they existed separately. If the subpage had not originally been extracted from the superpage but had originated in some other way (as described later in this chapter), or if it or the superpage had been modified significantly before reassembly, the graphical appearance of the resulting combined page would probably not be good. It would then have to be adjusted manually.



We need ResourceModel back in hierarchical form in order to continue developing it, but there's no need to redo the work you just undid. The results already exist in ResmodSubtrans:

- Close ResourceModel. Don't save changes.
- Open ResmodSubtrans.

You are now back where you were before executing the **Replace by Subpage** command.

### Developing on a Subpage

In the preceding example, we created a net first, then moved part of it to a subpage, leaving behind a substitution transition to represent it.

It is also possible to create a substitution transition first, open the resulting subpage, and do development work on the subpage. This method is generally preferable to creating a net and then extracting from it, because it never requires everything to be on the same page at the same time. After all, the whole purpose of hierarchy is to avoid having to put everything on the same page.

For example, suppose we want to add additional logic to Resnet to improve the way processes obtain resources. Currently the transition Obtain performs this task, and it makes no attempt to do so intelligently: this could be a source of inefficiency.

One possibility is to replace Obtain with something more complex directly on the subpage that contains it. But that would clutter up the subpage with details. These would obscure the clear view of Resnet's overall structure that the subpage currently presents.

A better method is to move Obtain to a subpage, and then modify the subpage to provide the behavior we want. Then the subpage on which Obtain now appears, would also be a superpage of a lower subpage. No special problems result from such an arrangement: substitution transitions may be nested to arbitrary depth.

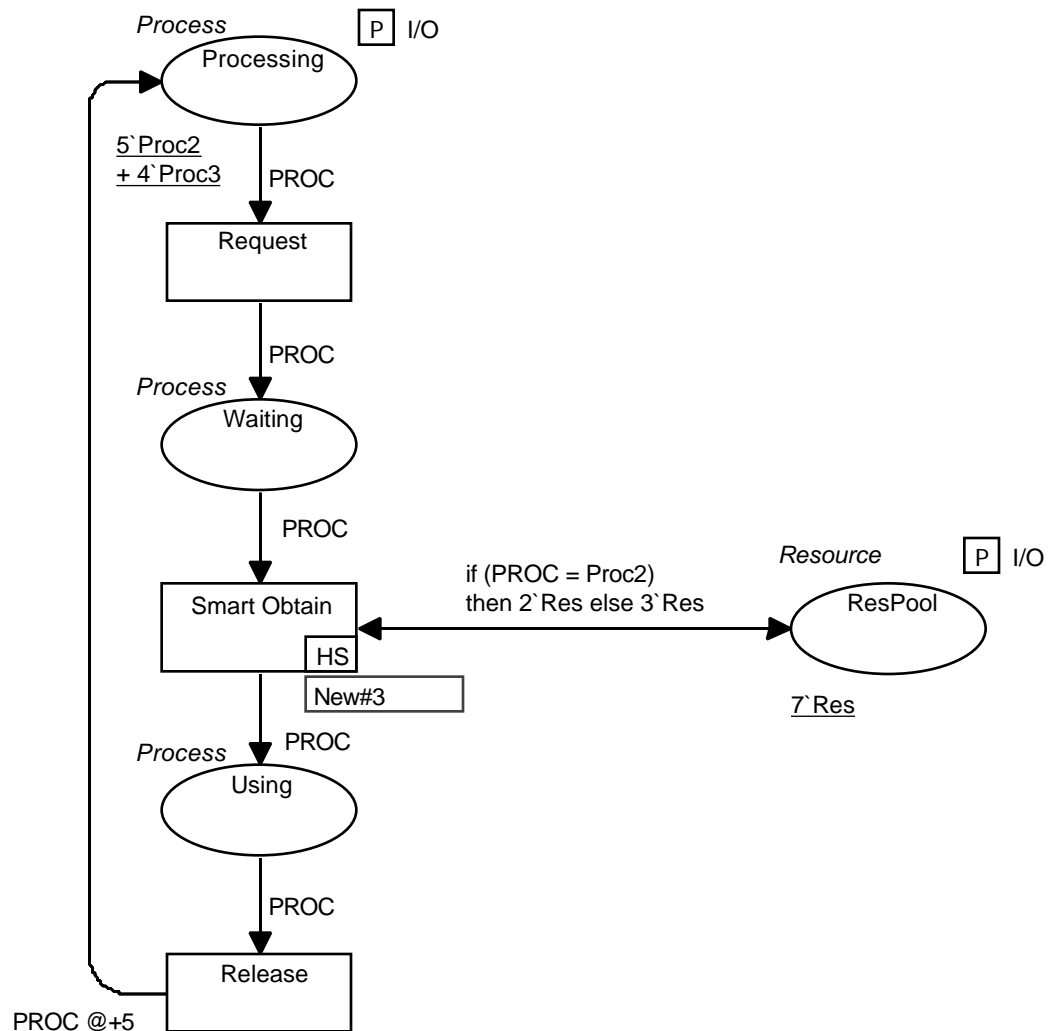
### Create the Substitution Transition and Subpage

- Make New#2 the current page.
- Select the transition Obtain. Be sure nothing else is selected.
- Choose **Move to Subpage** from the **CPN** menu.

## Design/CPN Tutorial

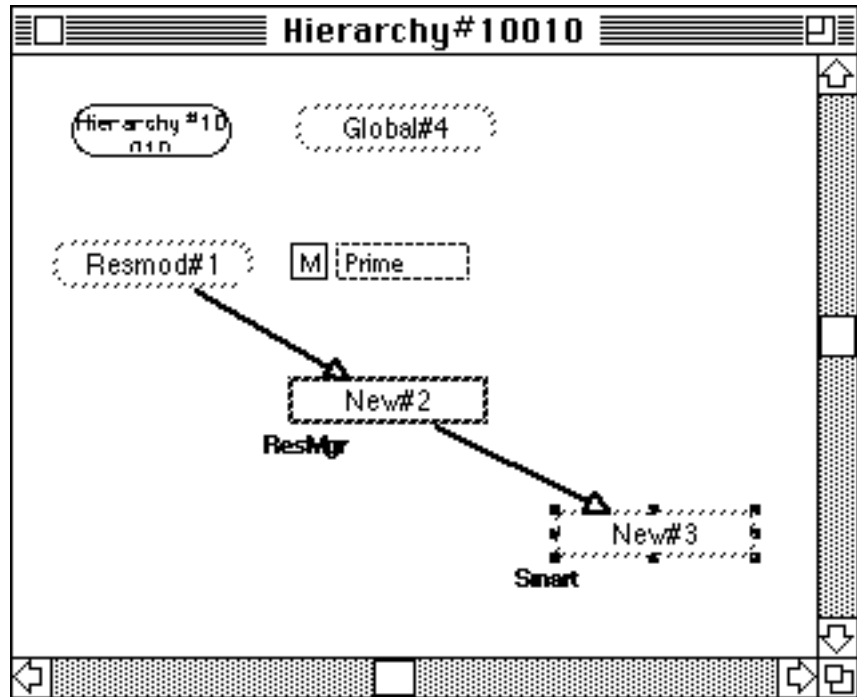
- Draw the substitution transition so that it just surrounds Obtain, just as you did before.
- Name the substitution transition “Smart Obtain”.
- Move the hierarchy regions to better positions.

New#2 should now look about like this:



### The Modified Hierarchy Page

- Open the hierarchy page:



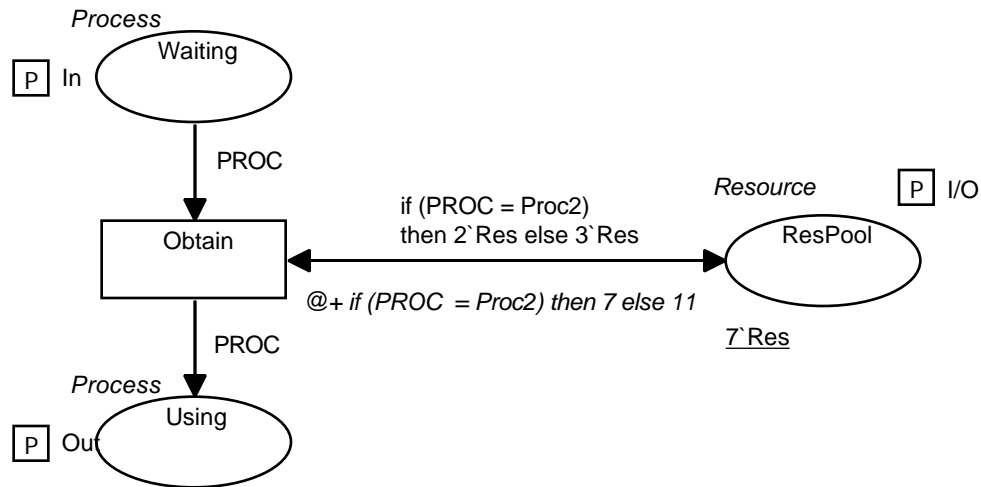
The page shows that New#2 is a subpage of Resmod#1 and contains the details of the substitution transition ResMgr, and that New#3 is a subpage of New#2 and contains the details of the substitution transition Smart Obtain, which has been truncated to Smart. Design/CPN does this to avoid cluttering up the hierarchy page with long names.

Note that this time the hierarchy page was not disordered by the creation of the subpage. The reason is that it was already in a standardized configuration established by the **Redraw Hierarchy** command after New#2 was created, so Design/CPN could determine automatically how to update it. The hierarchy page prior to the creation of New#2 had been rearranged manually.

### The New Subpage

- Double-click on the page node for New#3 to open the new subpage.

The subpage opens:



There is only one general difference between this subpage and New#2, the subpage (which is now also a superpage) that you created previously. Both ports on New#2 were listed as “I/O”. Here Waiting is listed as “In”, and Using is listed as “Out”. These designations refer to the status of the ports in the context of the subpage: Waiting is an “In” port because it is an input place of Obtain, and Using is an “Out” port because it is an output place of Obtain.

ResPool now exists on three different pages. It is a socket on Resmod#1, both a port and a socket on New#2, and a port here on New#2. All this really means is that ResPool is now part of a three-member fusion group that spans three pages. It is still the same place on all three pages: nothing new has been added.

### Relationship of Pages in a Hierarchy

The various pages in a hierarchy created with substitution transitions are connected only by their ports and sockets. They have no other functional relationship to each other. Therefore they can be modified completely independently of each other: changes made on a subpage have no effect whatever on the net structure on a superpage, and changes made on a superpage have no effect on a subpage.

Subpages and superpages interact directly only during simulation. Their interaction consists only of the consequences of the fact that a port and a socket are functionally the same place, so that any change that a subpage makes to a port's marking intrinsically changes the superpage's socket's marking, and vice versa.

Thus a subpage functions as a module in much the same way that a subroutine does. There is a clearly defined interface, implemented in this case by the ports and sockets, and the only connection between a module and anything that uses it is the interface. However a

subpage is not exactly like a subroutine: it is in some ways more like a macro, as we will see later in this chapter.

### Deleting a Subpage

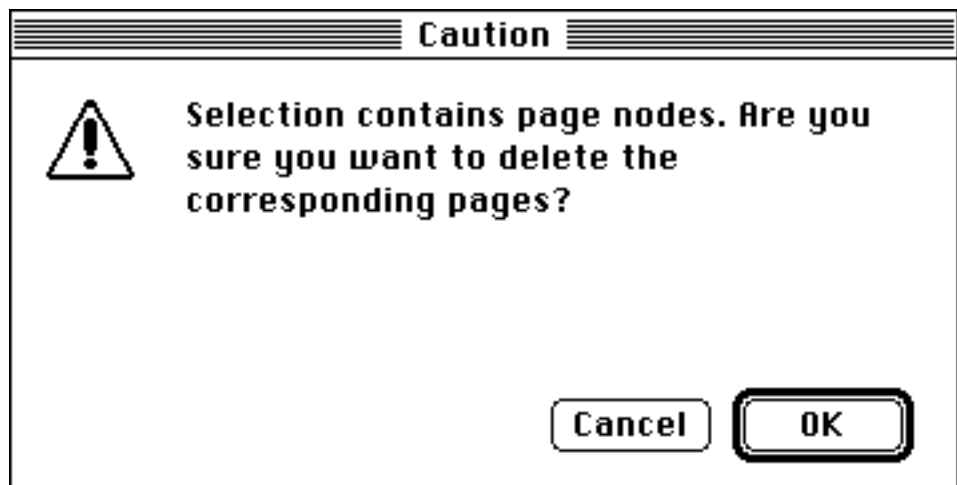
As a diagram evolves, it sometimes happens that a subpage is no longer needed for some reason. The need then is not to copy it to a superpage, but to get rid of it entirely.

Deleting such a subpage is no different from deleting any other page. Design/CPN automatically makes all adjustments necessary to reflect the fact that the subpage no longer exists.

For example, suppose we decided that implementing a more intelligent algorithm for `Obtain` is not necessary after all, so that the subpage New#3 on which we might have implemented such an algorithm is no longer necessary. To delete the subpage:

- Open the hierarchy page.
- Select the page node for New#3.
- Press DELETE. (You can't **Cut** a whole page.)

A confirmation dialog appears:



- Click **OK**.

The page node for New#3, and the connector designating it as a subpage of New#2, both disappear. New#3 is gone.

- Open New#2.

Smart Obtain is now just an ordinary transition, and the regions designating it as a substitution transition have disappeared.

Is the net functionally the same as before Smart Obtain was created? It is not! Smart Obtain does not have a time region, but Obtain did: the region was moved to a subpage along with Obtain, and disappeared from the diagram when the subpage was deleted. Creating and then deleting a subpage, even if only one transition was moved to the subpage, can permanently change a superpage.

Let's go back to the original diagram and proceed down another path.

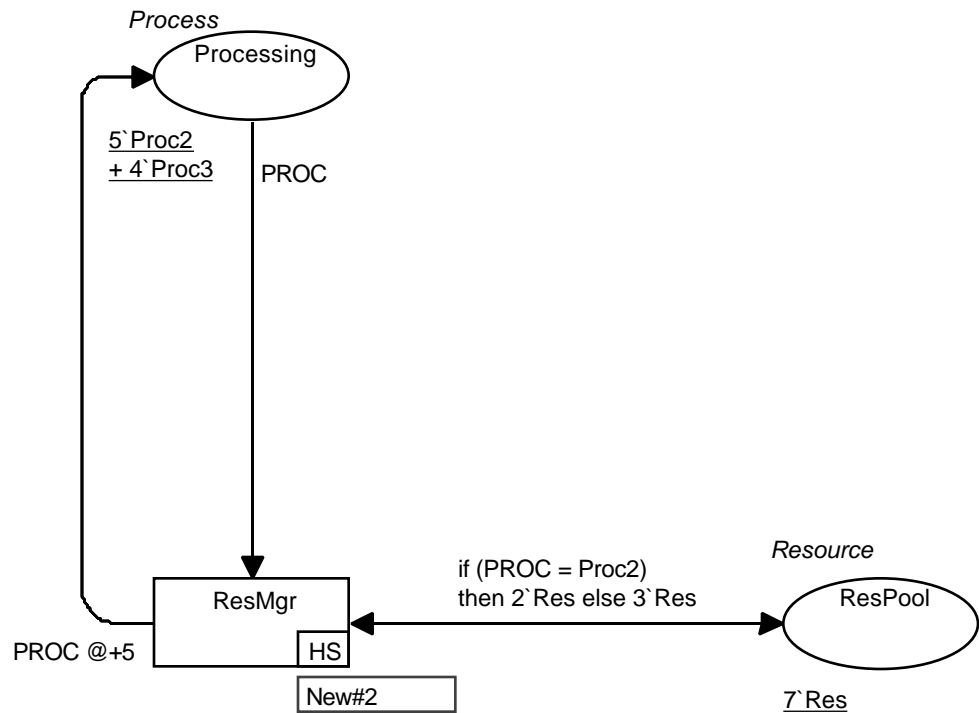
- Close ResmodSubtrans. Don't save changes.
- Re-open ResmodSubtrans.

## Using a Subpage More Than Once

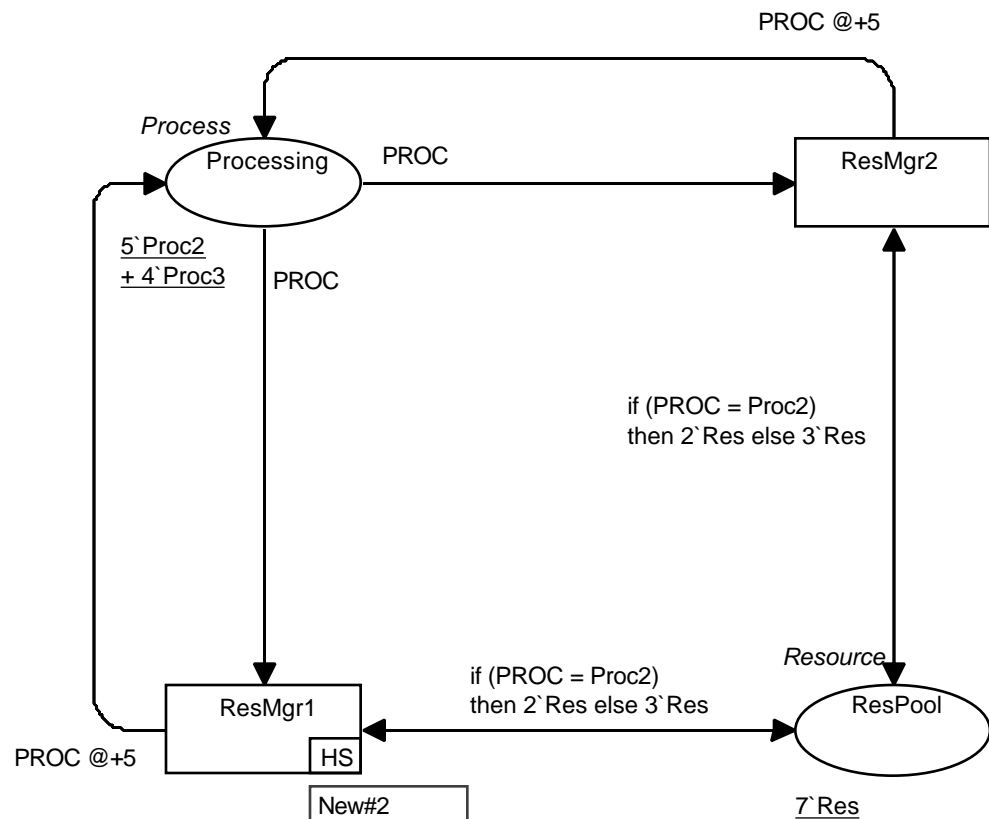
The capabilities we have looked at so far would be more than enough to make substitution transitions a useful technique. But these capabilities are only the beginning. The real power of substitution transitions lies in the fact that a subpage need not be the value of only one substitution transition: it can be used repeatedly, as the value of any number of substitution transitions on any number of superpages. Let's see how this works.

- Make Resmod#1 the current page:

## Substitution Transitions



Now modify the page so that it looks like this:



Suggestions:

1. Get ResMgr2 to be the same size as ResMgr (now ResMgr1) by using **Change Shape**. To use it:
  - Select ResMgr2.
  - Choose **Change Shape** from the **Makeup** menu.
  - Click on ResMgr1.
2. Use **Align Horizontal** and **Align Vertical** to position ResMgr2 exactly.
3. Create the new arc inscriptions by cutting and pasting the existing inscriptions.

The next step is to convert ResMgr2 to a substitution transition, and associate it with the existing subpage New#2.

- Select ResMgr2.
- Choose **Substitution Transition** from the **CPN Menu**.

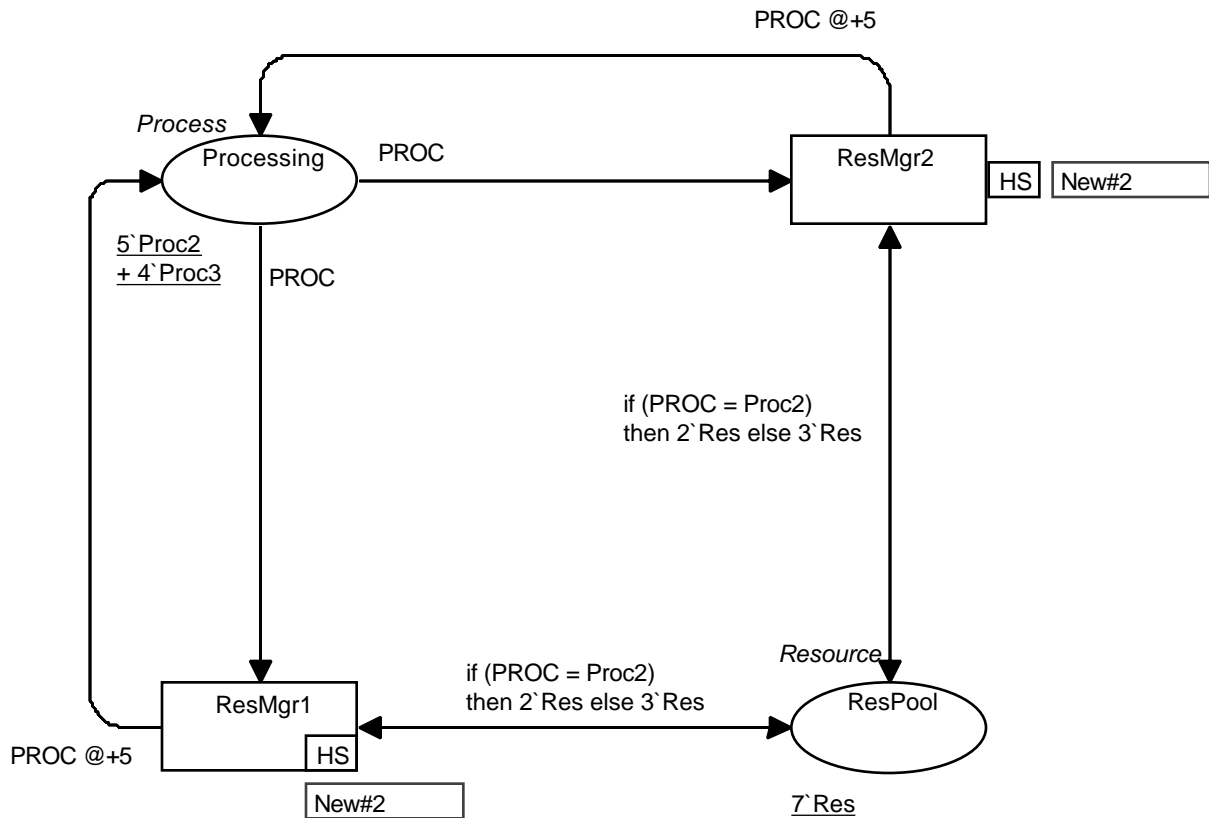
In order to convert ResMgr2 into a substitution transition, Design/CPN needs to know what subpage to associate with the transition. It therefore displays the hierarchy page, allowing you to use the mouse to indicate the subpage.

- Click the mouse on the page node for New#2.

Resmod#1 reappears. ResMgr2 is now a substitution transition whose value is the subpage New#2:

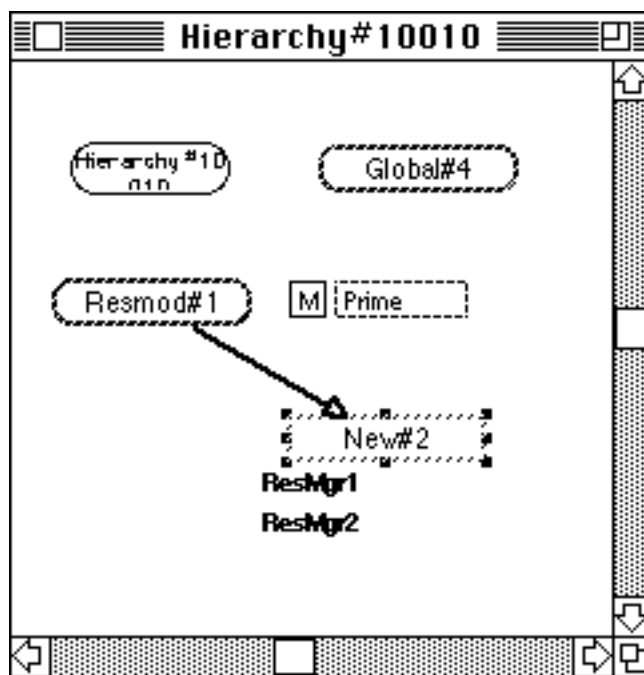


## Substitution Transitions



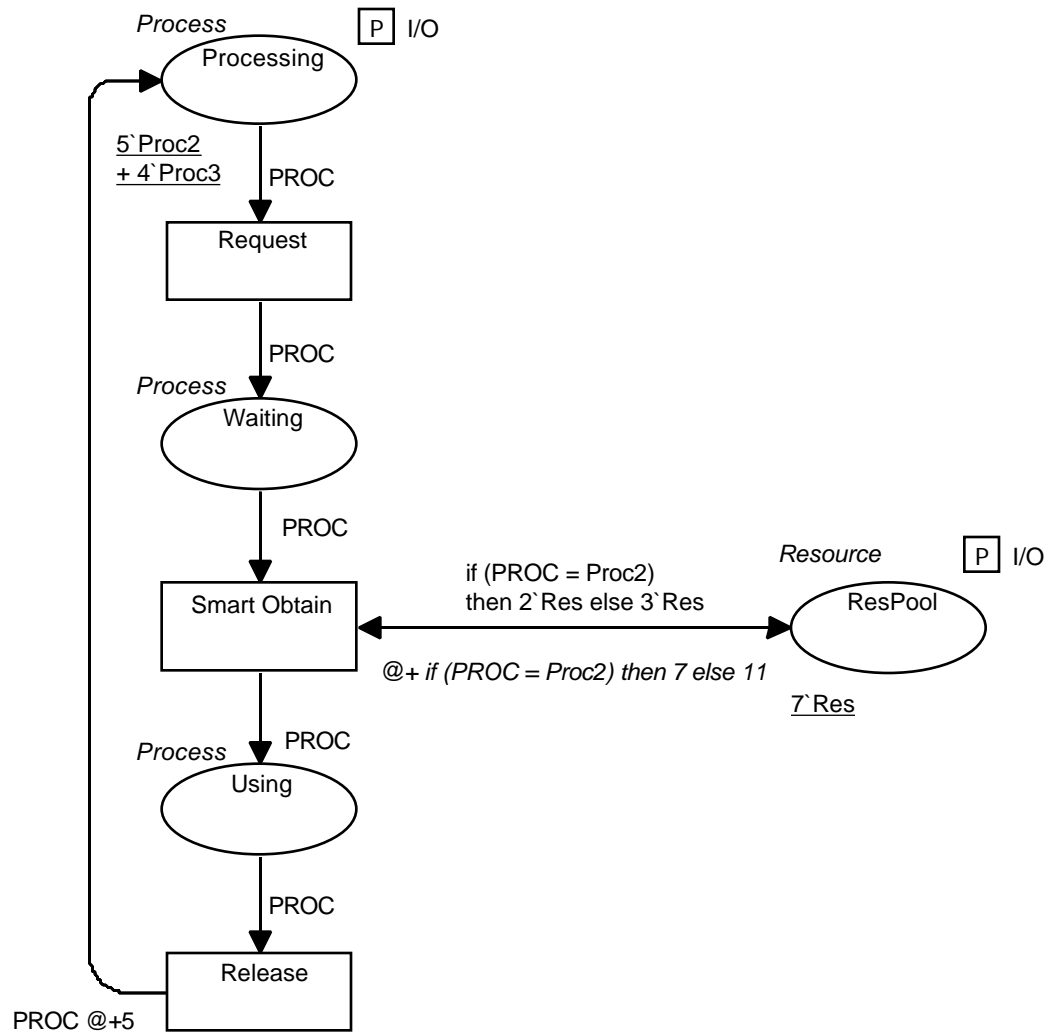
The two port places `Processing` and `ResPool` on the subpage have the same names as two socket places connected to `ResMgr2` on the superpage. Design/CPN has used this information to assign the ports to the sockets automatically. If any port had a name that did not match that of a socket, you would need to assign that port to a socket manually, as described later in this chapter.

- Open the hierarchy page:



New#2 is now the value of two substitution transitions, so both names appear below its page node in substitution tag regions.

- Open New#2:

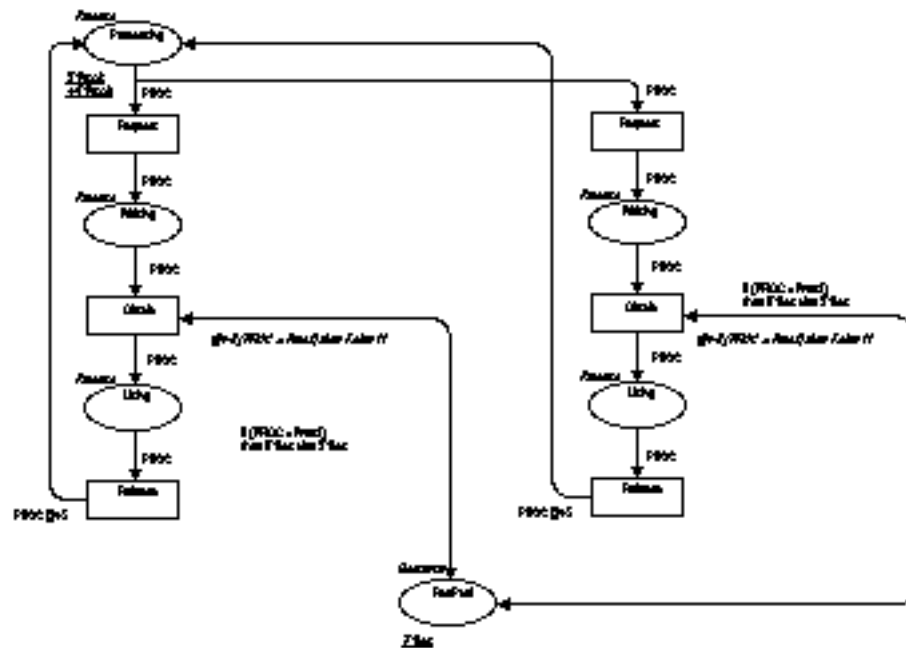


It's appearance hasn't changed at all. It is still the same as it was when it was the value of only one substitution transition. It would remain the same no matter how many substitution transitions it was the value of.

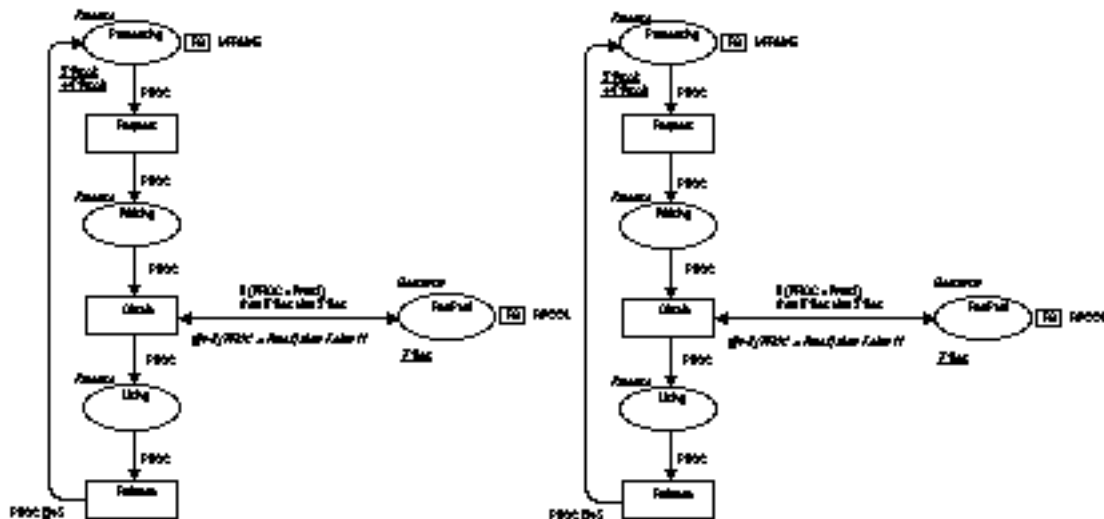
## Structure of the Diagram

The diagram now contains a superpage, Resmod#1, with two substitution transitions. Each of these transitions has as its value the subpage New#2.

The structure of the diagram would be functionally the same if the submodel on New#2 appeared physically on the superpage in place of each of the substitution transitions ResMgr 1 and ResMgr 2. If we actually made this change, and did some physical rearranging, the result could look like this:



Those long arcs are perfect candidate for elimination by using fusion sets:



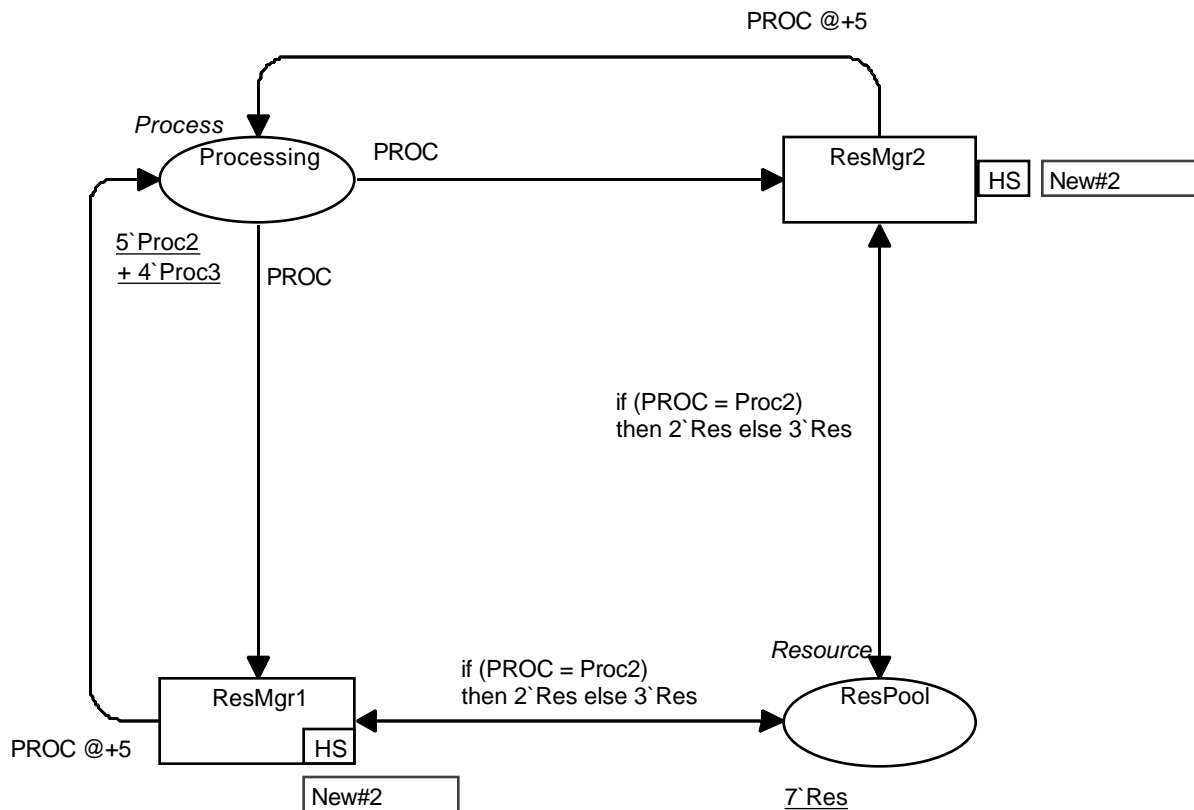
You can't tell from the small print in the figure, but the two Processing nodes are in a fusion set called **MFRAME**, and the two ResPool nodes are in a fusion set called **RPOOL**. These names are appropriate, because as the transformation we have just traced proves, Resmod#1 in the current diagram is functionally identical to the Resmod#1 with two copies of Resnet and two fusion sets that you worked with in the previous chapter. The only difference is that the previous Resmod#1 was implemented by using fusion places, while this one was implemented using substitution transitions.

Since there is only one page, any type of fusion set would produce functional identity with Resmod#1 from the previous chapter. The sets above happen to be global sets. Later in this chapter we will see how to use substitution transitions to create the same effects as global, place, and instance fusion sets.

### Substitution Transitions and Multiplicity

Let's take another look at the current Resmod#1.

- Make Resmod#1 the current page:



To show the isomorphism between this page and Resmod#1 in the previous chapter, we imagined physically replacing **ResMgr1** and **ResMgr2** the submodel on **New#2**. Of course, we would not want Design/CPN to actually perform such a copying operation when executing the diagram. If a large subpage were used many times, the result would be impossibly huge.

Design/CPN avoids such problems by using multiplicity: it creates a separate instance of a subpage for each substitution transition that uses it. This is the same technique we used at the end of the previous chapter to avoid having to physically duplicate Resmod#1 to get a copy of it. The only difference is that in this case we don't have to

explicitly indicate the number of instances needed, because Design/CPN knows how many substitution transitions use a given subpage, and generates instances accordingly.

## Subpages, Subroutines, and Macros

The similarity between a subpage and a subroutine should now be obvious. Just as you can create a subroutine independently of other code, have only one copy of it, and call it from many points in a program, so you can create a submodel independently of other net structures, have only one copy of it, and use it in many locations in a model.

However, subpages differ from subroutines in one important way. Although they are not physically copied into superpages, the effect of such copying being provided by using instances, the functional effect is the same as if they actually had been copied. Consequently substitution transitions are really more like macros than like subroutines: they are not called during execution and instantiated only when and if they are called, as subroutines are, but are replaced by their values during compilation, as macros are.

In practice, the macro-like quality of substitution transitions has only one consequence: substitution transitions cannot be used recursively, either directly or indirectly, for such usage would lead to an infinite loop of substitution. Design/CPN does not permit you to specify recursive substitution.

## Simulating With Hierarchy

Now let's switch over to the simulator and see how the diagram executes.

Since the diagram containing the net was not created on your system, it does not contain the information necessary to allow Design/CPN to communicate with the ML process. Therefore you must load that information into it before you can execute it.

- Choose **ML Configuration Options** from the **Set** menu.

A dialog appears.

- Click **Load**.
- Click **OK**.

The necessary information about the ML process is copied to the diagram.

- Enter the simulator.
- Adjust the simulation regions on Resmod#1 and New#2.
- Start simulation.

Continue simulation through several steps. Between steps, look at both Resmod#1 and the two instances of New#2. You will need to switch between the instances as you did when watching simulation with instance fusion sets.

- Depress the SHIFT key.
- Click the mouse on the title bar of Resmod#1.

The **Instance Switch** dialog appears.

- Use the mouse to select the other instance of Resmod#1.
- Click **Switch**.

The dialog disappears. The title bar of the Resmod#1 window now indicates that the other instance is on display.

Verify that all three `Processing` places (one socket and two ports) always have the same marking, that the same is true for all three `ResPool` places, and that all other markings may be different on the two instances of New#2.

When you have completed your observations:

- Leave the simulator.

There will be various simulation regions left over, and these would get in the way of future observations.

- Use **Remove Sim Regions** from the **CPN** menu to remove the simulation regions from each page.

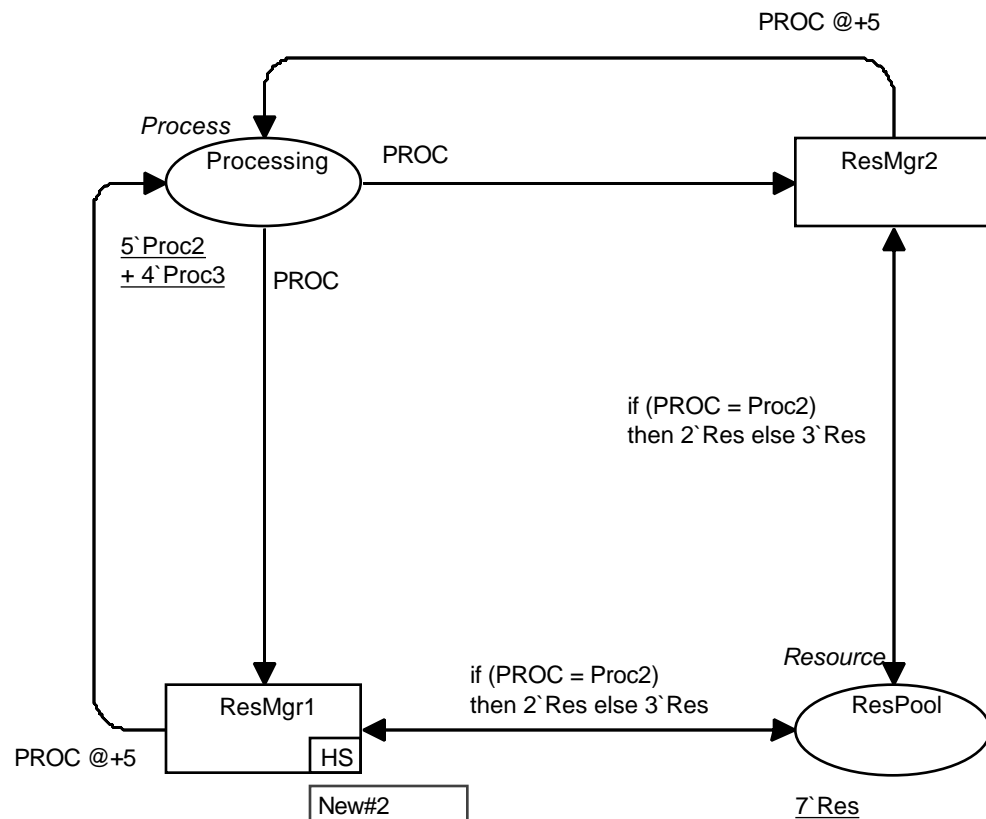
## Deleting a Reference to a Subpage

Sometimes there is a reason to break the association between a substitution transition and its subpage. When such an association is broken, the substitution transition becomes an ordinary transition. The subpage remains associated with any other substitution transitions that reference it, and continues to be available for use by additional substitution transitions.

To delete a reference to a subpage, all you need to do is delete the hierarchy key region from the substitution transition that references it.

- Open Resmod#1.
- Select the hierarchy key region **HS** next to ResMgr2.
- Press DELETE.

The hierarchy key region and associated hierarchy region disappear:



ResMgr2 is now an ordinary transition. Its status is the same as if it had never been a substitution transition.

## Manually Assigning Ports to Sockets

It was easy to convert ResMgr2 into a substitution transition, because there was no need to indicate explicitly which port should be associated with which socket: Design/CPN made the assignment automatically by matching port names with socket names.



Equating ports and sockets by matching their names can be a great convenience, but it does not always provide what is needed.

Requiring port and socket names to match would be like requiring a subroutine always to be called with the same variable names in the argument list. The disadvantages are obvious. To avoid such problems, Design/CPN allows you to manually specify what port will be matched with what socket when a substitution transition is created.

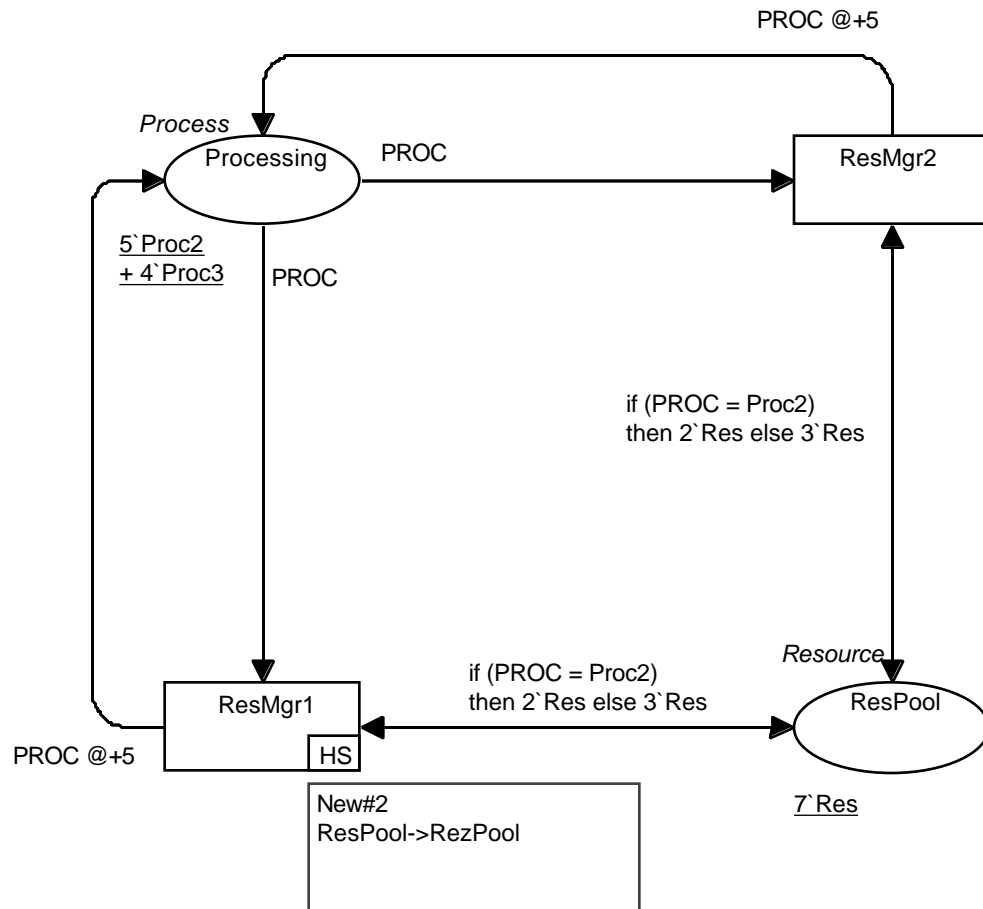
Let's convert `ResMgr2` back to a substitution transition, but this time do the port assignments by hand. This will be more interesting if all the names don't match, so:

- Make New#2 the current page.
- Change the name of `ResPool` to `RezPool`.

You have now changed the name of a port that was already equated to a socket, on `ResMgr1`. This makes no functional difference. Design/CPN matched the names when `ResMgr1` became a substitution transition, but the association of port with socket that was created then has no further connection with the name of either place. Port and socket names may therefore be changed without reference to one-another.

When names don't match, we need some other way to determine what port is associated with what socket. The necessary information is kept in the hierarchy region of the substitution transition.

- Expand the hierarchy region of `ResMgr1` so that you can see all of its contents:



The second line in the region indicates that the socket `ResPool` in `Resmod#1` is associated with a port named `RezPool`. The first line indicates that `RezPool` is on `New#2`. There is no line for `Processing`, because it has the same name as its port (at least in the context of `ResMgr1`). The region could contain a line "`Processing->Processing`", but for economy such lines are just omitted. If both `Processing` and `ResPool` matched their ports, the region would contain only the name of the relevant subpage.

Now let's make `ResMgr2` a substitution transition again, but do our own port assignments.

- Select `ResMgr2`.
- Choose **Substitution Transition** from the **CPN Menu**.

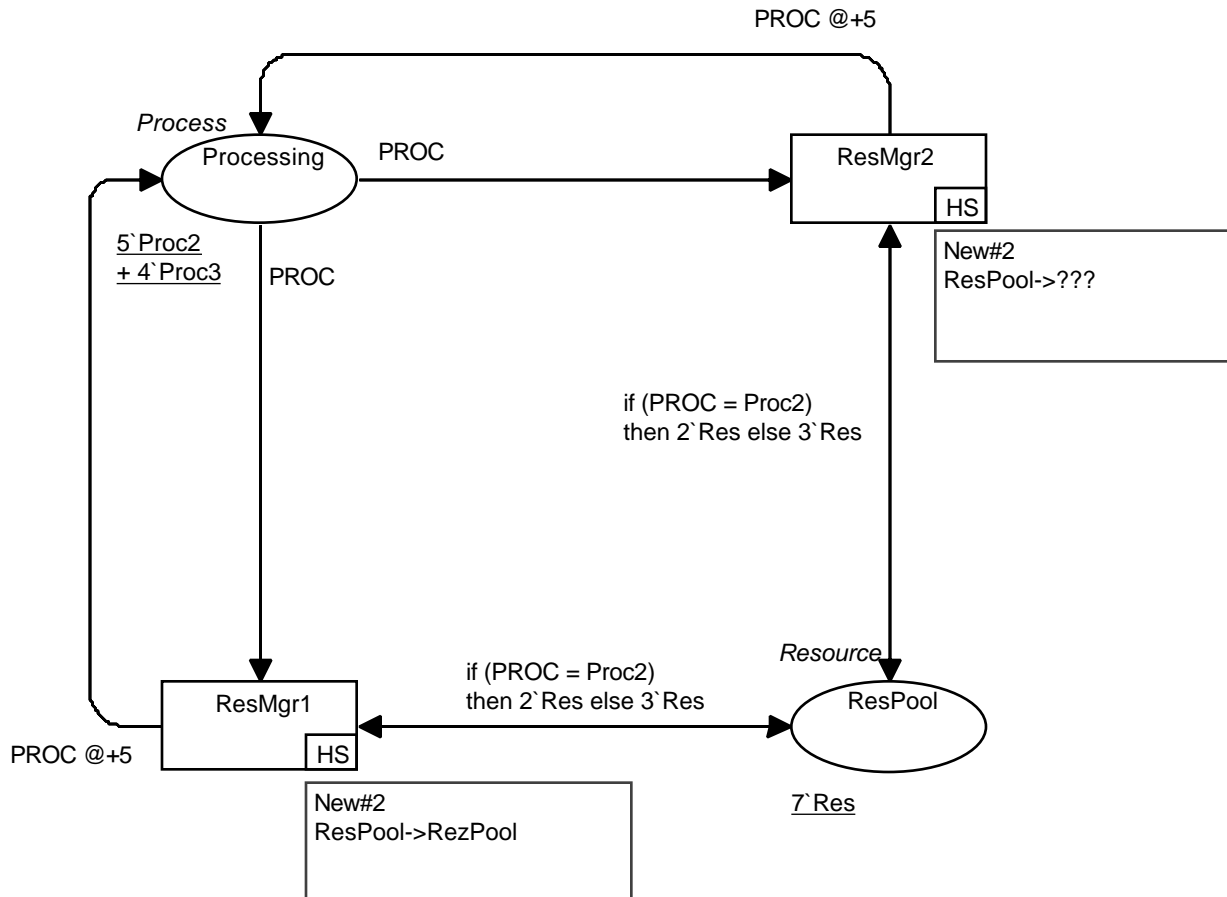
Design/CPN displays the hierarchy page.

- Click the mouse on the page node for `New#2`.

`Resmod#1` reappears. `ResMgr2` is again a substitution transition whose value is the subpage `New#2`.

## Substitution Transitions

- Expand the hierarchy region for ResMgr2:



(The region has been repositioned to make the figure fit on the page, but you don't need to bother changing your diagram.)

The port and socket named `Processing` have been matched automatically, so no reference to them appears in the hierarchy region. The region indicates that `ResMgr2` has one unassigned socket: `ResPool`.

Now let's make the necessary port assignment:

- Select `ResPool`.
- Choose **Port Assignment** from the **CPN** menu.

The status bar displays Select one of the substitution transitions. We're working with `ResMgr2` now, so:

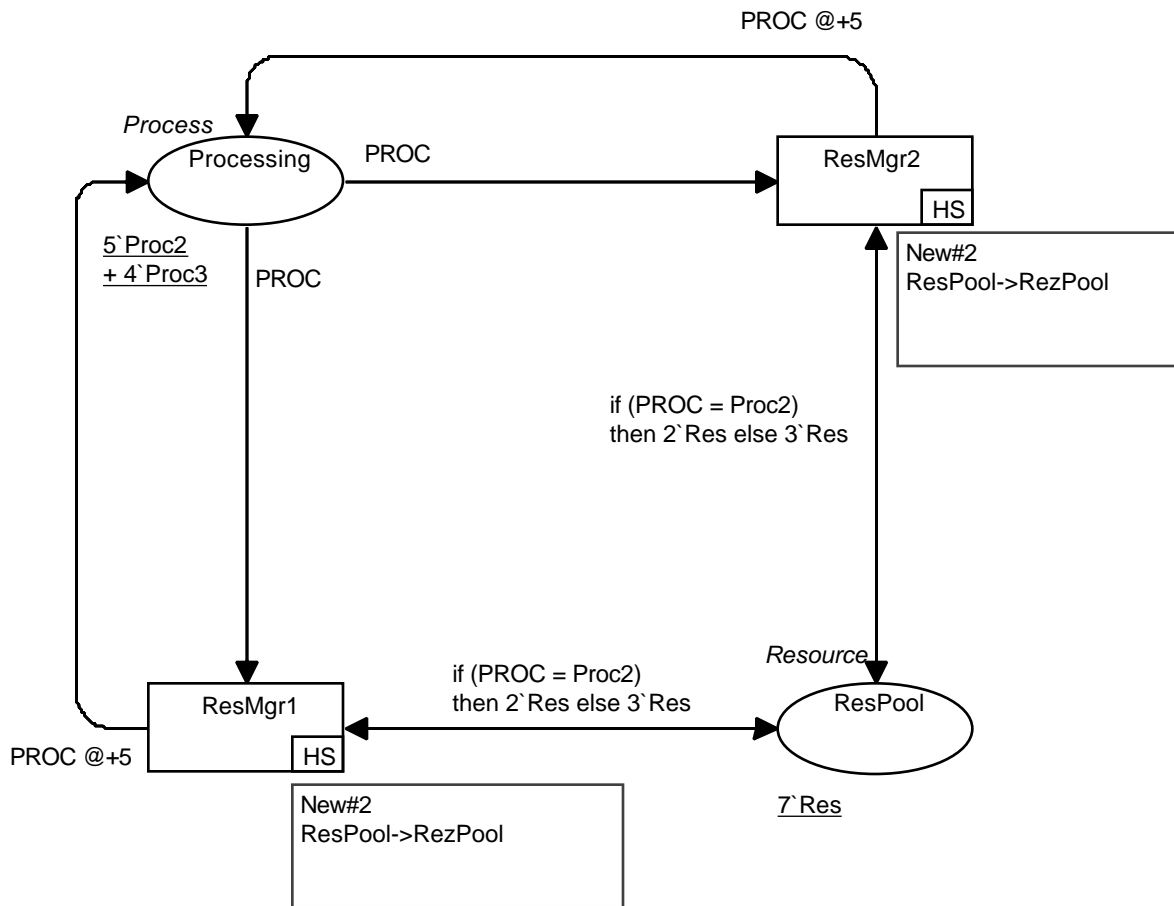
- Click on `ResMgr2`.

## Design/CPN Tutorial

Design/CPN displays the subpage for ResMgr2, so that you can indicate which port you want.

- Click on RezPool.

Design/CPN returns you to Resmod#1. The hierarchy region for ResMgr2 now shows that the socket ResPool is associated with the port RezPool:



You can also use this technique to reassign ports and sockets. Just make the assignment you want, and the existing assignment will be replaced by the new one.

# **APPENDIX B**

## **The Sales Order Model**

# Chapter B1

## Introduction to the Sales Order Model

### Files for Use With This Appendix

The Sales Order Model consists of the following files:

1. BPMA\_Model\_V2 and BPMA\_Model\_V2.DB: These files contain the model itself. They are kept in the TutorialDiagrams directory that is supplied with this tutorial.
2. SalesOrderParmsV2: This file holds parameters that control the behavior of the Sales Order Model. It is kept in the TutorialDiagrams directory.

The exercises in this appendix ask you to modify SalesOrderParmsV2, so that file is not locked. To make it easy for you to restore the file to its original state, TutorialDiagrams also contains a locked copy of it, under the name SalesOrderParmsV2.ORIG.

### Overview of the Sales Order Model

The Sales Order Model is a high-level representation of a generic Sales Order System. The model is designed to be executed (run) repeatedly with different values representing attributes of the system. After each run, the model displays charts that summarize the results of the run.

The charts that are displayed at the end of a run may suggest changes that would result in a more efficient system. The effect of making such changes can then be investigated by setting appropriate attributes in the model, running it again, and examining resulting charts.

The Sales Order Model does not specify what product is being ordered, and omits many of the details that would be part of a real sales order processing operation. Its goal is to demonstrate various basic techniques for making models and using them to analyze sys-

tem performance. These techniques, once understood, can be used to add additional capabilities to the model as needed to meet the requirements of a particular situation.

This chapter gives an overview of the model. Chapter B2 shows how to execute it. Chapter B3 shows how it might be used to identify and test possible improvements to the Sales Order System. When you have read these chapters you will be able to use the Sales Order Model to perform a wide variety of observations and experiments.

## Entities Represented in the Model

The Sales Order Model represents entities of three types:

1. Customer Requests. These are orders for the product being sold. Customer requests come in two varieties, designated Big and Small. For brevity, this document generally refers to customer requests as jobs.
2. Staff members. These are people who process jobs. Staff members come in two varieties, designated Expert and Novice.
3. Equipment. These are pieces of machinery that are used by staff members to process jobs. Equipment comes in two varieties, designated Fast and Slow.

## Action Cycle for Processing Orders

Jobs are processed through a cycle of actions that occurs once for each request. Those actions are:

1. A job begins to be processed.
2. A staff person is assigned from a staff pool to handle the request. A big job requires an expert staff person; a small job may be assigned either an expert or a novice.
3. The staff person enters the order into the system in some way.
4. The staff person obtains a piece of equipment from an equipment pool, uses it in some way to process the order, then returns the equipment to the pool. A big job requires a fast piece of equipment; a small job may be assigned either fast or slow equipment.

5. The staff person creates an invoice and ships it along with the requested product.
6. The staff person returns to the staff pool.
7. Processing of the job is complete.

## Inefficiency in the Sales Order System

There are several possible sources of inefficiency in the system that the Sales Order Model depicts. They are:

1. There may be more staff members and/or equipment pieces than are needed to keep up with the job stream. Inefficiency results because staff members must be paid, and equipment pieces maintained and amortized, whether or not they are actively producing revenue.
2. There may be too few staff members and/or equipment pieces to keep up with the job stream. Inefficiency results because jobs that could be generating revenue are instead waiting to be processed.
3. There may be a mismatch between the composition of the job stream and the composition of the staff and/or equipment pools. For example, if the job stream consists mostly of big jobs, but most staff members are novices and/or most equipment is slow, much unproductive waiting will occur. Conversely, if the job stream consists mostly of small jobs, but most staff members are experts and/or most equipment is fast, there will be a wasteful usage of expensive resources on low-revenue jobs.

## Using the Model to Reduce Inefficiency

The Sales Order Model can be used to detect inefficiency of the described types, and to identify and test possible changes that would reduce it. The general method for using the model is:

1. Set the properties of the job stream, staff pool, and equipment pool.
2. Execute the model given these properties. The model will gather and display statistics on the efficiency of the sales order operation.
3. Examine the results of the simulation and identify changes that are likely to improve efficiency.



4. Execute the model again with different properties that reflect the changes identified in Step 3.
5. Iterate until a satisfactory result is obtained.

An example of this method appears in the next chapter.

## Simulation Parameters

To allow for the testing of different combinations of system load, staff composition, and equipment composition, details on these properties are not wired into the model: they are supplied as parameters in a file that the model reads at the beginning of each simulation run. This file is an ordinary ASCII file, and may be edited with any text editor to specify the properties for a run.

### Job Stream Parameters

Jobs are created and entered into the model in batches at regular intervals during execution of the model. The parameters that control the stream of jobs are:

1. The number of batches to be generated during the run.
2. The time interval between batches.
3. The number of big jobs in each batch.
4. The number of small jobs in each batch.

### Job Value Parameters

Big and small jobs differ in the gross revenue that a request of each kind generates. The parameters are:

1. The gross revenue generated by a big job.
2. The gross revenue generated by a small job.

### Staff Parameters

The parameters that control the staff composition are:

1. The number of expert staff members.
2. The cost per time unit of an expert staff member.

3. The speed of an expert staff member. The fastest member has a speed of 1; slower members are represented by larger numbers.
4. The number, cost and speed of novice staff members.

### Equipment Parameters

The parameters that control equipment composition are exactly analogous to those that specify the staff parameters. The parameters are:

1. The number of fast pieces of equipment.
2. The cost per time unit of a fast piece of equipment.
3. The speed of a fast piece of equipment. The fastest piece has a speed of 1; slower pieces are represented by larger numbers.
4. The number, cost and speed of slow pieces of equipment.

### Gathering and Displaying Statistics

As the model executes, it gathers statistics that measure the performance of the Sales Order System. These statistics can be used to reveal inefficiency caused by excessive, insufficient, inefficient, or and/or inappropriate staff and equipment, and to identify changes that are likely to result in a more efficient operation.

The statistics are accumulated using the Design/CPN Statistical Variables facility. This facility allows many types of statistical analysis to be performed on the data that results from a simulation run. When a run is complete, the results are displayed in five bar charts. These charts are produced by the Design/CPN Charting facility.

The statistics gathered by the Sales Order Model are of two kinds: revenue statistics and efficiency statistics.

### Revenue Statistics

These statistics show the overall financial performance of the Sales Order Model for a run, specifically:

1. The gross revenue brought in by the jobs processed during the run.
2. The operating cost incurred in processing the jobs.

3. The profit generated by the run. This is just the difference between the gross revenue and the operating cost.

### Efficiency Statistics

These statistics provide a view of the internal behavior of the Sales Order Model. Their goal is to reveal areas of inefficiency, and thereby indicate changes that might decrease operating cost and hence increase profit. The statistics kept are:

1. The time spent and expense incurred in processing big jobs. This figure is divided into useful time and expense, and wasted time and expense caused by inefficiency.
2. The time spent and expense incurred in processing small jobs. This figure is divided into useful and wasted time and expense.
3. Statistics on productive and nonproductive time and expense for expert and novice staff members.
4. Statistics on productive and nonproductive time and expense for fast and slow equipment.

### Using the Sales Order Model

Now that you have a general understanding of what the Sales Order Model is and does, you can go on to use it to measure the efficiency of the Sales Order System, and to identify and test possible improvements to that system. Methods for performing these operations are described in the next two chapters.

# Chapter B2

## Running the Sales Order Model

This chapter tells you how to start and run the Sales Order Model. Methods for using the model to analyze and improve the Sales Order System are covered in the next chapter.

Before you proceed, be sure you know where to find the following files:

1. BPMA\_Model\_V2: This file is used to open the model.
2. SalesOrderParmsV2: This file holds parameters that control the behavior of the model.

### The Simulation Parameter File

Before we run the model, let's take a look at the parameter file.

- Open the file SalesOrderParmsV2 using any text editor.

The file as supplied by Meta Software looks like this:

```
(* This file contains the parameters for
the BPMA Sales Order model. *)

(* Parameters controlling the incoming
stream of Customer Requests: *)
(* Total number of batches of requests;
Time between batches; Big requests per
batch; Small requests per batch: *)
3      5      5      5

(* Parameters establishing the dollar
value of Customer Requests *)
(* Big request value; Small request value:
*)
200    50

(* Parameters establishing the Staff: *)
(* Number of Expert staff; Expert cost fac-
tor; Expert speed factor: *)
2      20    1
```

```
(* Number of Novice staff; Novice cost fac-
tor; Novice speed factor: *)
2      10      5

(* Parameters establishing the Equipment:
*)
(* Number of Fast equip pieces; Fast cost
factor; Fast speed factor: *)
2      2      1
(* Number of Slow equip pieces; Slow cost
factor; Slow speed factor: *)
2      1      5

(* End Of File. *)
```

The lines may not wrap on your screen exactly as shown above, but the information should be the same.

### Restoring the Simulation Parameter File

If the parameter file does not match the above figure, if you have doubts about its correctness for any reason, or if you ever want to restore its original values after experimentation, make a new one by copying SalesOrderParmsV2.ORIG, which should be in the same directory as SalesOrderParmsV2. See the beginning of the previous chapter for details.

### System Properties Specified by These Parameters

The parameters shown above specify a Sales Order System with the following properties:

1. A job stream consisting to 3 batches of jobs. A batch will enter the system every 10 simulated time units. Each batch will consist of 5 big jobs and 5 small jobs.
2. Job values of 200 value units for a big job, and 50 for a small job.
3. A staff pool consisting of 2 experts and 2 novices. Each expert costs 20 value units per time unit, and has a speed of 1, the fastest possible speed. Each novice costs 10 value units per time unit, and has a speed of 5, denoting a speed that is one fifth that of an expert.
4. An equipment pool consisting of 2 pieces of fast equipment and 2 pieces of slow equipment. Each fast piece costs 2 value units per time unit, and has a speed of 1, the fastest possible speed. Each slow piece costs 1 value unit per time unit, and has a speed of 5, denoting a speed that is one fifth that of a piece of fast equipment.

Note that it does not matter what real-world measures we equate with a “time unit” or a “value unit”. These are just abstract measuring units. To produce exact time and cost calculations, we could equate them with any appropriate standard measures. If we care only about relative changes in system performance, as will be the case in this chapter, no particular identification with standard measures is needed.

### Analysis of the Initial Parameters

Before we execute the model with these parameters, let's try to anticipate the results by looking at the figures. Do the values shown seem likely to give efficient performance?

There is really no way to tell. Nothing is glaringly wrong, but the nonlinear relationships between the speed and the cost of a staff member or equipment piece makes it hard to be sure. This is typical of situations in which modeling and simulation are useful: there is just no way to tell what will happen by thinking about it. If there were, there would be no need for modeling, and no profit in simulation.

### Starting the Model

To start the Sales Order Model:

- Start Design/CPN as you would any X-Windows application.

You are now in Design/CPN, and the editor is active.

- Choose **Open** from the **File** menu.

The **Open File** dialog appears.

- Navigate to the directory that contains the file BPMA\_Model\_V2 and open the file.

The Sales Order Model diagram opens.

- Choose **ML Configuration Options** from the **Set** menu.
- Click **Load**.
- Click **OK**.

The necessary information about the ML process is copied to the diagram. We don't want to change the model, but only to execute it, so:

- Choose **Enter Simulator** from the **File** menu.

### Running the Model

We are now ready to run the model.

- Choose **Automatic Run** from the **Sim** menu.

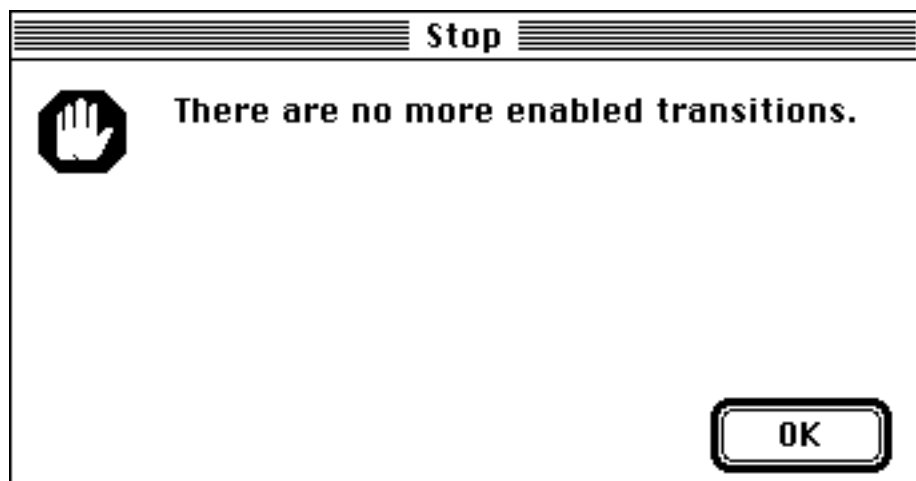
Before the model can execute, it must read in the simulation parameters from the parameter file. This requires indicating the location of the file to the file system. The **Open File** dialog appears.

- Open the parameter file.

The **Open File** dialog disappears. The mouse becomes the Wait Timer.

When all jobs have been processed, the Wait Timer disappears and an output file is written. This file contains a record describing each job that was processed. In order to write the output file, the computer needs to know where to put it. It will request this information by displaying the **Save File** dialog. Use this dialog to indicate where you want the output file to go. It is generally most convenient to put it in the same directory as the model.

After the output file has been written, five bar charts that display the revenue and diagnostic statistics are created (this takes a few minutes), and a dialog appears:



Model execution is now complete.

- Click **OK**.

The dialog disappears. You may now examine the charts.

### **Analyzing and Using Simulation Results**

Now that you know how to run the Sales Order Model, you can begin to use it to examine and improve the Sales Order System, as described in the next chapter.





# Chapter B3

## Using the Sales Order Model

This chapter shows you how to interpret the results of an execution of the Sales Order Model, identify possible improvements, modify the simulation parameters to represent the improvement, and examine the results.

### Interpreting the Results of a Simulation Run

The results of a run of the Sales Order Model are written to an output file and displayed in five bar charts.

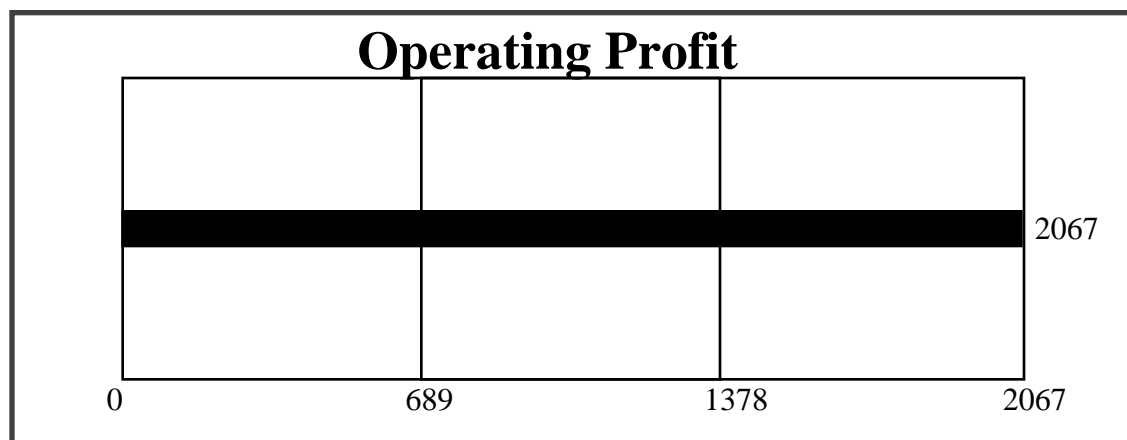
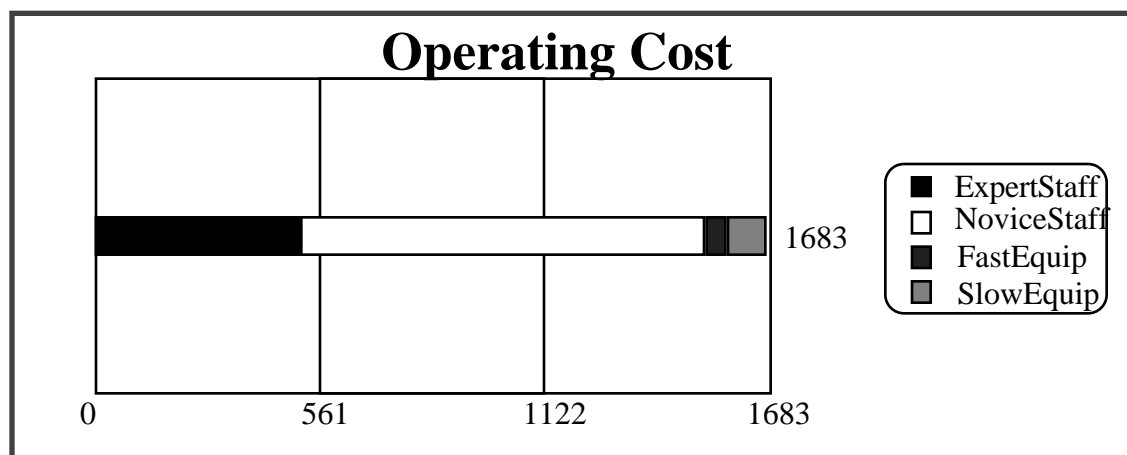
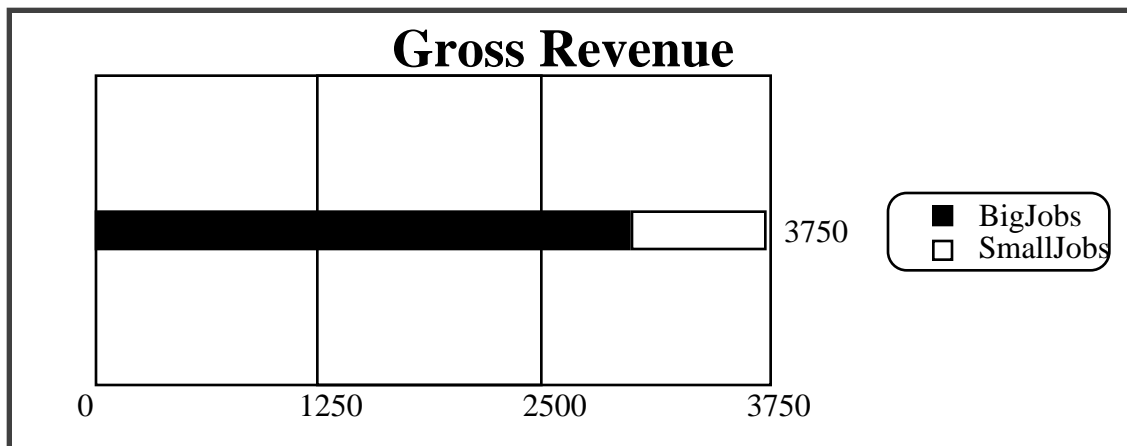
The output file contains a record for each job that was processed. These records represent the invoices and products that the Sales Order Model produces as it operates. The information in this file does not relate to analysis of system performance, though such information could of course be included in it, so we will not consider it further.

The information in the five bar charts can be used to analyze system performance. It is displayed in three charts giving information about revenues, and two giving information about efficiency.

### Examining the Revenue Charts

First let's look at the revenue charts that result from executing the model given the parameters shown at the beginning of the previous chapter. If you just executed the model, as described in the previous chapter, these results are already on display. If not, execute the model now.

The revenue charts that you see should look about like this:



(The charts you see may have slightly different numeric values, caused by random differences in the details of model execution, and their appearance will be somewhat unpolished. The appearance of a chart can be polished by using the Design/CPN Editor.)

These charts show a gross revenue of 3750 and a profit of 2067. The expenses are 1683, divided as shown in the Operating Cost chart.

In absolute terms, the profit is large; but the question is not whether profit is large, but whether it could be larger. These three charts do not help in answering that question. To answer it, we must look at the efficiency charts.

### Examining the Efficiency Charts

There are two efficiency charts. One of the charts depicts time elapsed during the simulation run; the other depicts costs incurred. Due to the possibility of varying speed and expense parameters independently in the parameter file, the pictures presented by the time and cost charts may be very different.

The efficiency charts are on the page immediately below the revenue charts.

- Choose **Open Page** from the **Page** menu.

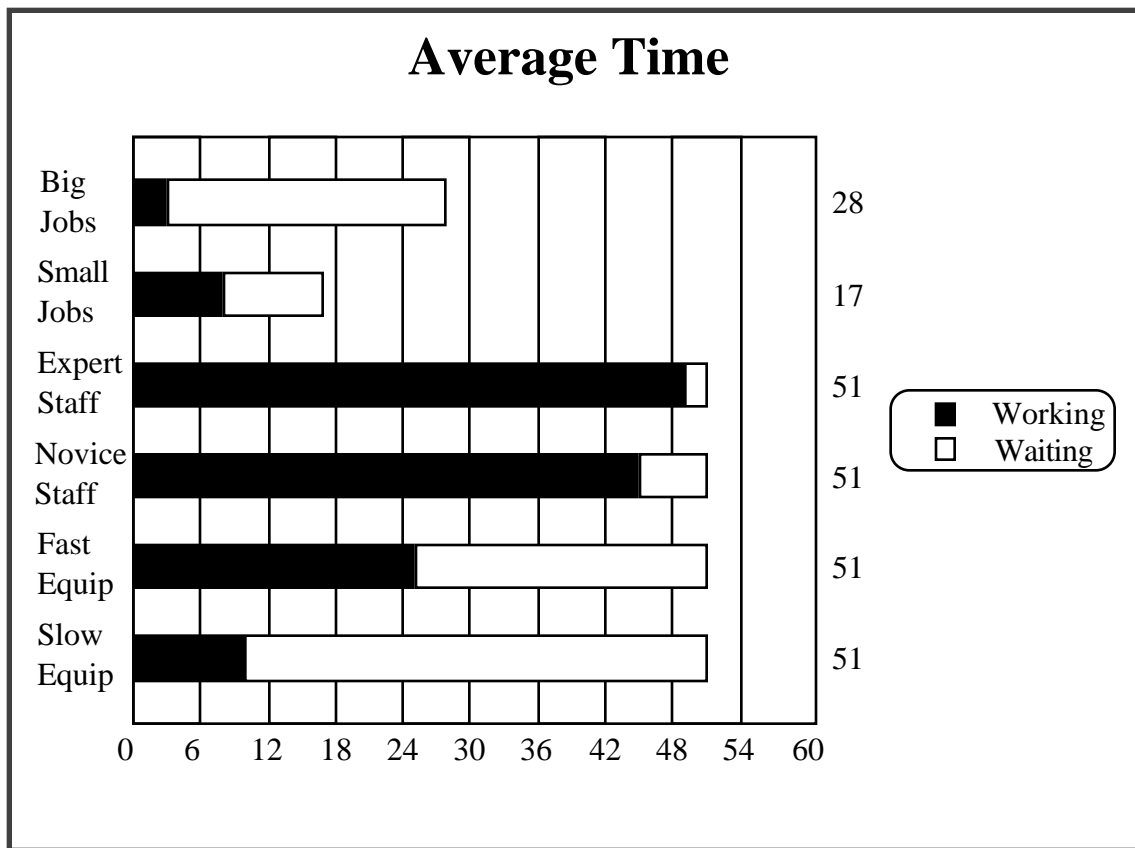
A page called the *hierarchy page* appears. This page contains a node called a *page node* for every page in the diagram. Note the page node named BAR#8. This node represents the page we were just looking at, containing the revenue charts. Next to the BAR#8 node is a node named BAR#7. That node represents the page that contains the efficiency charts.

- Use the mouse to select the node BAR#7.
- Choose **Open Page** from the **Page** menu.

The page BAR#7 opens. It contains a time chart and a cost chart.

### Examining the Time Chart

The time chart should look about like this:



The information displayed by this chart, in order left to right and top to bottom, is:

1. The average time required to completely process a big job.
2. The average time during which a big job waited for equipment.
3. The same averages for small jobs.
4. The average time an expert staff member worked processing jobs
5. The average time during which an expert staff member waited for a job to handle or for equipment to become available.
6. The same averages for novice staff.
7. The average time a fast equipment piece worked processing jobs
8. The average time a fast equipment piece waited because there was nothing for it to do.

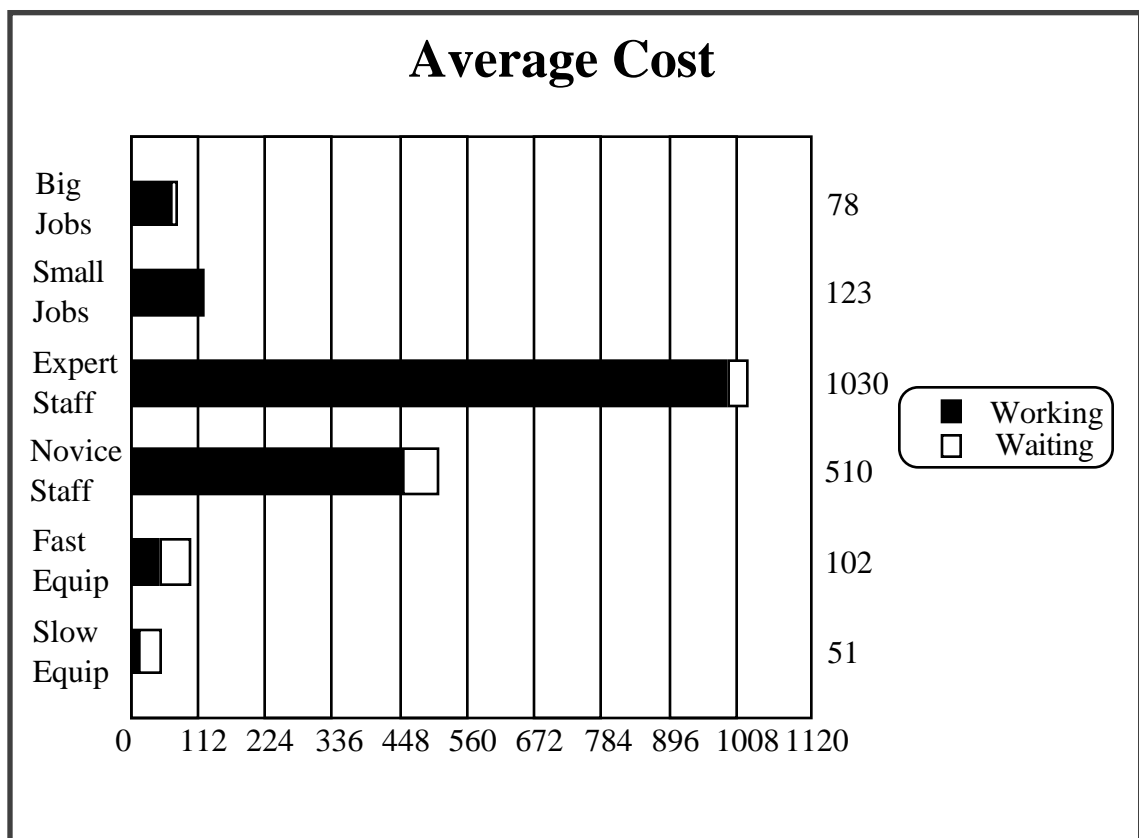
## Using the Sales Order Model

### 9. The same averages for slow equipment.

It is immediately obvious from this chart that there is something very wrong with the Sales Order System. Jobs spend a lot of time waiting to be processed, equipment spends a lot of time unused, and the staff is being used nearly to full capacity. There is obviously considerable inefficiency. Since only the staff is close to full utilization, the probable cause is that there is too little staff capacity.

### Examining the Cost Chart

Now let's look at the cost chart:



The information displayed by this chart is the cost corresponding to each time measurement shown in the time chart.

A comparison of the time and cost charts supports the hypothesis that there is insufficient staff. The time chart showed that jobs are waiting a long time on the average, yet the cost chart shows that this waiting is not causing an accumulation of expense. The only explanation is that the jobs are waiting without using staff or equipment: that is, they are waiting to enter the system in the first place. Since the only requirement for a job to enter the system is a staff person to

be assigned to the job, this is further evidence that more staff is needed.

## Experimenting With Possible Improvements

It would be nice if there were some algorithmic way to examine the results of simulation and determine exactly what changes will most improve a system, but for realistic systems there usually is not. If there were, the algorithm could have been applied in the first place, and there would have been no need for simulation. However this does not mean that one is reduced to guesswork. Examination of the results of simulation usually suggests one or several promising approaches; it just does not provide exact figures.

The way to determine how best to improve a modeled system is to use simulation to perform experiments. The general algorithm is:

- Change some property or parameter of the model.
- Run the simulation again.
- Compare the results to those of the previous run(s).
- Iterate to zero in on a solution.

In the case of the Sales Order Model the obvious hypothesis, based on the charts shown above, is that there is not enough staff capacity to either process all the jobs or use all the equipment. There are two ways in which the staff capacity might be increased: by adding more staff members, or by increasing the efficiency of existing staff members. These approaches could also be used in combination.

For our first experiment, let's try adding more staff members. The efficiency charts show that big jobs are doing much more waiting than small ones. Since big jobs can be handled only by expert staff members, we should probably add more experts than novices. Let's add two experts and one novice, resulting in a staff pool of four experts and three novices, and see what happens.

## Changing the Simulation Parameters

Since the composition of the staff pool is specified in the simulation parameter file, rather than being wired into the model, it can be changed easily by editing the file. The parameter file is called SalesOrderParmsV2. It is in the directory TutorialDiagrams, unless it was placed elsewhere when the model was installed.

- Open the parameter file using any text editor.

## Using the Sales Order Model

---

The parameter file is just an ASCII text file. The file appears as follows:

```
(* This file contains the parameters for
the BPMA Sales Order model. *)

(* Parameters controlling the incoming
stream of Customer Requests: *)
(* Total number of batches of requests;
Time between batches; Big requests per
batch; Small requests per batch: *)
3      5      5      5

(* Parameters establishing the dollar
value of Customer Requests *)
(* Big request value; Small request value:
*)
200      50

(* Parameters establishing the Staff: *)
(* Number of Expert staff; Expert cost fac-
tor; Expert speed factor: *)
2      20      1
(* Number of Novice staff; Novice cost fac-
tor; Novice speed factor: *)
2      10      5

(* Parameters establishing the Equipment:
*)
(* Number of Fast equip pieces; Fast cost
factor; Fast speed factor: *)
2      2      1
(* Number of Slow equip pieces; Slow cost
factor; Slow speed factor: *)
2      1      5

(* End Of File. *)
```

To change the parameters:

- Edit the parameter file to specify 4 expert and 3 novice staff members. Be sure not to make any other changes.

(Be careful not to make any changes to the overall format of the file, such as adding or deleting lines or parameter fields: the Sales Order Model assumes the exact format shown. If the model fails to work after you have edited the parameter file, you may accidentally have changed the file's format. Restore the file as described in the previous chapter and try again.)

The file should now look like this (changed figures are shown in bold):

```
(* This file contains the parameters for
the BPMA Sales Order model. *)
```



```
(* Parameters controlling the incoming
stream of Customer Requests: *)
(* Total number of batches of requests;
Time between batches; Big requests per
batch; Small requests per batch: *)
3      5      5      5

(* Parameters establishing the dollar
value of Customer Requests *)
(* Big request value; Small request value:
*)
200    50

(* Parameters establishing the Staff: *)
(* Number of Expert staff; Expert cost fac-
tor; Expert speed factor: *)
4      20      1
(* Number of Novice staff; Novice cost fac-
tor; Novice speed factor: *)
3      10      5

(* Parameters establishing the Equipment:
*)
(* Number of Fast equip pieces; Fast cost
factor; Fast speed factor: *)
2      2      1
(* Number of Slow equip pieces; Slow cost
factor; Slow speed factor: *)
2      1      5

(* End Of File. *)
```

When you have verified the changes:

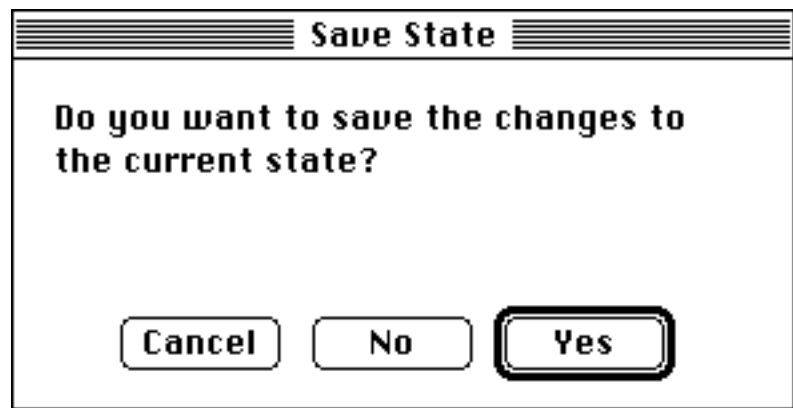
- Save the file as a text file (rather than in the editor's internal format, if any).
- Enter the simulator.

## Performing the Experiment

We cannot just re-execute the model, because it still contains the parameters and results of the previous run. We must first reset the model to its initial state.

- Choose **Initial State** from the **Sim** menu.

A dialog appears:



- Click **No**.

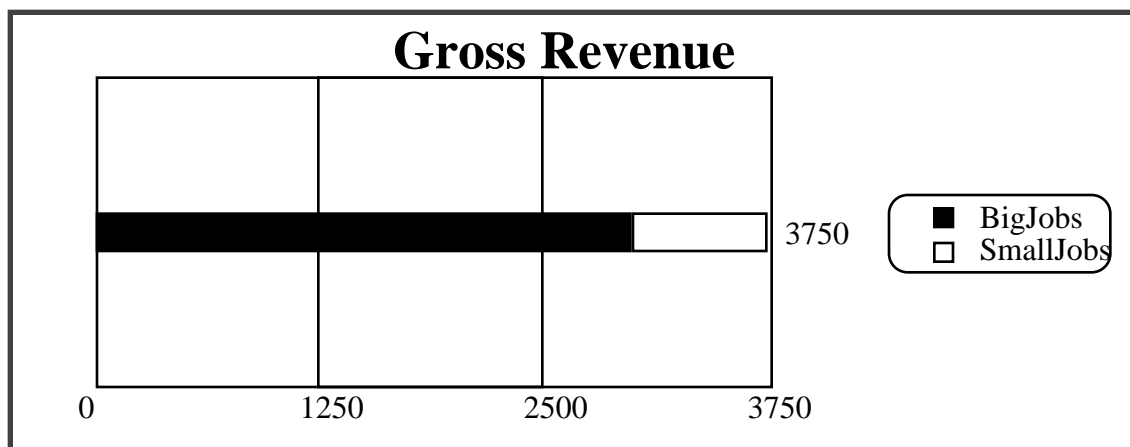
The net is now in the same state that it was in when we first entered the simulator.

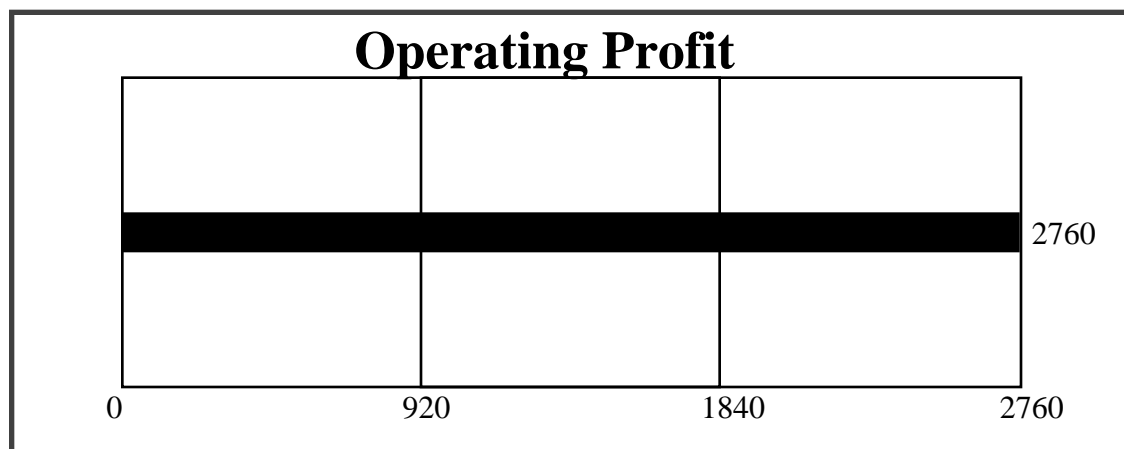
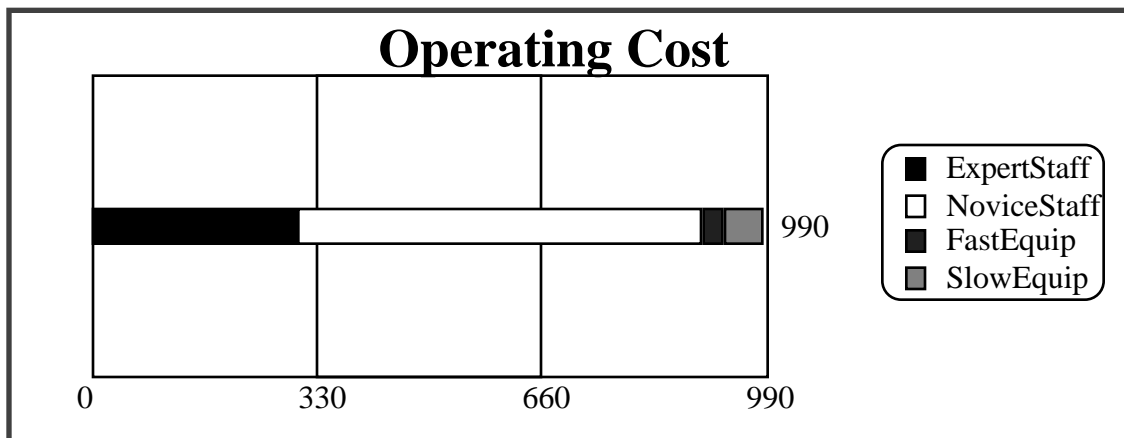
- Execute the model again, as described in the previous chapter.

When execution is complete, a new set of bar charts appears.

## Interpreting the New Results

The revenue charts you see should look about as follows:



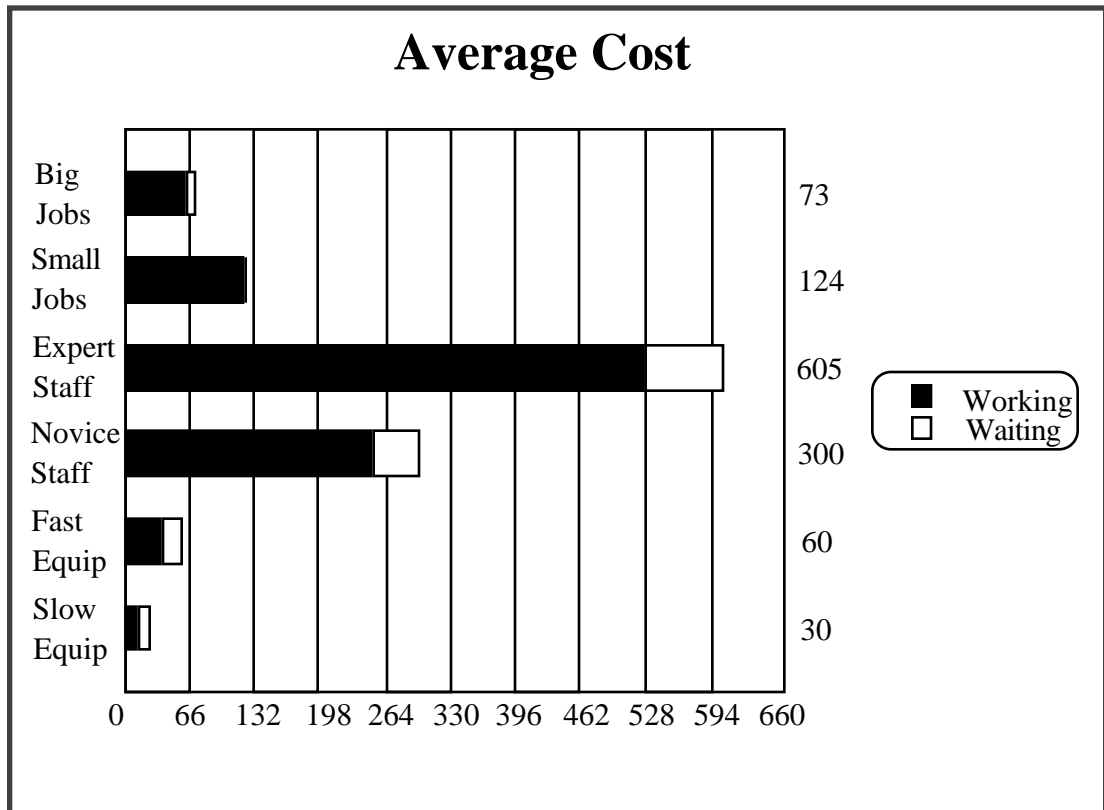
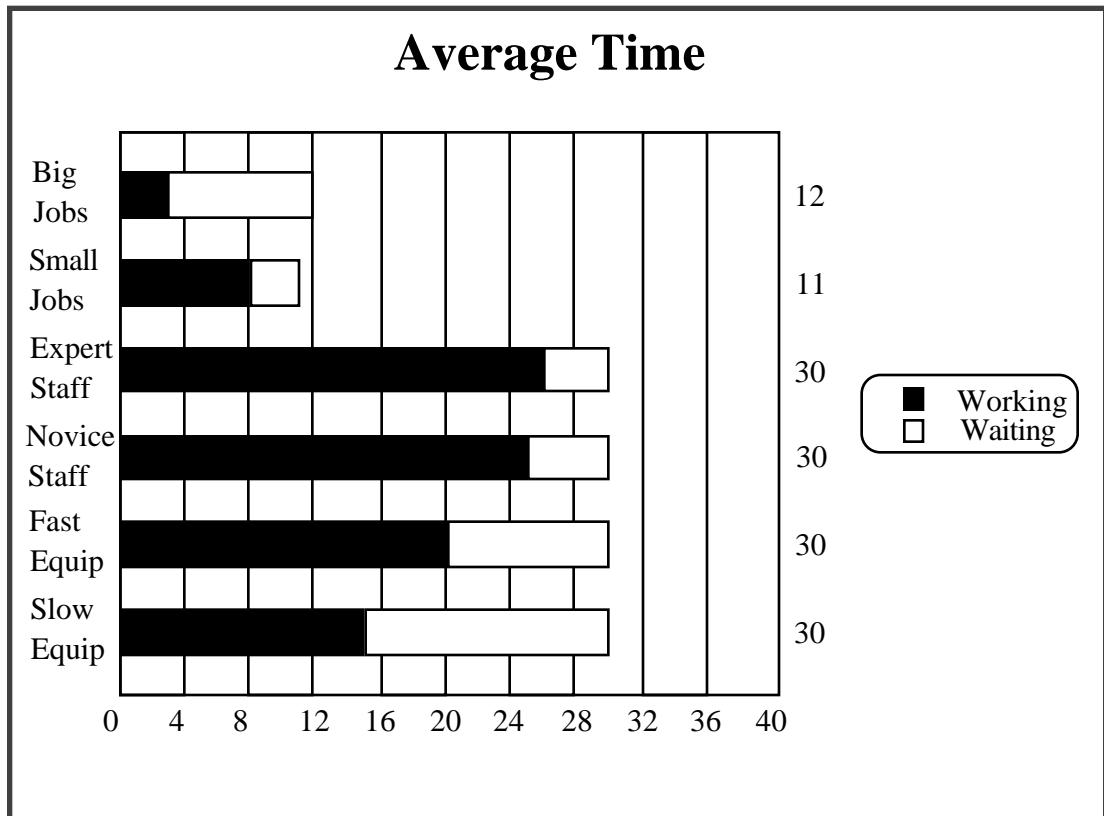


The previous charts showed a gross revenue of 3750 and expenses of 1683, yielding a profit of 2067. These charts show the same gross revenue, expenses of 990, and a profit of 2760. This is a very substantial improvement.

Now let's look at the efficiency charts.

- Use **Page** menu commands to go to the page BAR#9.

The efficiency charts should look about like this:



A comparison of these charts with the efficiency charts from the first run makes the cause of the improvement shown in the revenue charts obvious: job waiting time has been significantly reduced, and throughput has almost doubled. Clearly the hypothesis that more staff was needed was correct, and the addition of two experts and one novice was a step in the right direction.

Look again at the original values in the parameter file, earlier in this chapter. These values do not give clear evidence of a staff shortage or of any other problem. Yet the efficiency charts from the first simulation run made the problem immediately obvious, and suggested a solution that has proved effective. Modeling and simulation have taken a thoroughly recondite set of figures and derived from them a result that can be seen at a glance and tested in a few minutes.

## Additional Experiments

It is rare for the first experiment with a model to produce the best possible results. Adding two experts and one novice has greatly improved performance; perhaps additional staff would improve it further. On the other hand, there is more staff waiting in the second run than in the first. Have we added too much additional staff? Perhaps only one additional expert was needed; perhaps the additional novice was not needed at all.

- Experiment with different staff compositions and note how performance changes.

Equipment typically contributes much less direct cost than staff; the parameter file values for equipment cost reflect this fact. Excess equipment is therefore not likely to be a major handicap, but of course any unnecessary cost is worth reducing. On the other hand, insufficient equipment can cause substantial indirect cost by creating delays that tie up staff members, and by reducing overall throughput.

- Experiment with changes to the equipment pool and note how performance changes.

Your experiments with staff and equipment compositions should yield results that would substantially improve performance if implemented. Obviously one would not want to hire and fire actual staff members, or buy and discard real equipment, in trying to make such an improvement. It is much faster and cheaper to work with a model, performing experiments until the requirements of the situation become clear, and only then modifying the system that the model represents.

### More General Use of the Sales Order Model

The preceding exercises illustrate the techniques necessary to use the Sales Order Model to frame and test hypotheses. But the exercises do not begin to exhaust the possibilities of the model: they have dealt only with minor changes to the staff and equipment pools.

The ability to specify all simulation parameters in a file allows you to specify and test models of drastically different Sales Order Systems. You can vary the characteristics of the job stream. You can specify different ratios of speed to cost for staff and/or equipment. You can experiment with the advantages of upgrading existing staff/equipment performance versus adding more staff/equipment of the kind currently used. These are only a few of the possible options.

Much more could be said, and many suggestions made, about experimenting with the Sales Order Model. But the best way to learn about it, or about any model, is to use it, hands on, framing and running experiments much as a scientist explores the unknown. In modeling and simulation, no discussion, however exhaustive, can substitute for hands-on experience.

- Make a major change to the job stream parameters, and determine by experiment what staff and equipment capabilities give good performance under the new conditions.

### Improving the Sales Order Model

As you experimented with the model, you probably noticed that your experiments tended to fall into two classes. Sometimes there was an obvious bottleneck, shown by some type of staff or equipment being at or near 100% utilization. Other times there was significant waiting of all types. The system never reached 100% efficiency, and the efficiency charts did not indicate the reason.

Apparently the capabilities provided by changing the simulation parameters do not provide sufficient control, or they could be used to produce 100% efficiency. The information displayed by the charts is also insufficient, or it would reveal why 100% efficiency is not obtainable.

These insufficiencies do not of course indicate that the parameters and charts are valueless. As we have seen, they can provide very useful information. Furthermore, without the capabilities that they provide we could not have become aware that we need something more; we would have remained entirely in the dark. This is a common phenomenon in modeling: a model that was designed to explore one type of problem has revealed the existence of other problems that the model is not equipped to explore.

### Analyzing the Problem

To get an idea of where to start, let's look at the overall structure of the Sales Order System that the model represents. It contains an obvious potential cause of trouble: the rule that a big job must be assigned an expert staff member and fast equipment, while a small job may use any staff member or equipment. This rule creates opportunities for big and small jobs to interfere with each other in ways that are not at all obvious from cursory examination, and that inevitably cause inefficiency.

Obviously it would be useful to experiment with different staff and equipment allocation rules, just as we experimented with different simulation parameters. But the allocation rules used by the Sales Order Model cannot be changed by altering the parameter file: they are wired into the model.

It would be possible in principle to put information about allocation rules into the parameter file, but this method would not provide a sufficiently flexible approach to experimenting with such rules. In general, parameter files are useful for supplying data to a model, but not for specifying the model's properties. These are specified in the model itself when it is created, and changed by changing the model.

### Changing the Model

Methods for creating and modifying CP net models are covered in the Design/CPN Tutorial. If you learn the material presented there, you will be able to alter the structure of the Sales Order Model to use more efficient rules for staff and equipment allocation, and in any other way that might be useful. You could then derive information by iteratively experimenting not only with the parameters of the model, but the model itself.

## Ending a Session With Design/CPN

When you are through experimenting and want to exit Design/CPN:

- Choose **Quit** from the **File** menu.

A dialog appears that offers an opportunity to create a saved state. If you want to execute the net in the future without having to rebuild an ML file, and/or want to preserve the charts you have made for future examination:

- Click **Yes**.
- Save the net in NewTTDiagrams.

Otherwise:

- Click **N o**.

Design/CPN quits. You are back in the environment from which you started.





# **APPENDIX C**

## **Troubleshooting**

# Chapter C1

## Troubleshooting

This appendix describes various problems that you may encounter when you attempt to run Design/CPN, and tells you how to solve them. All of the problems relate in some way to the interface between Design/CPN and the computer on which it runs. The problems described are:

- CPN Settings file is missing or obsolete.
- ML configuration is unspecified or incorrect.
- ML Interpreter cannot be started.

When one of these problems occurs, Design/CPN displays a descriptive dialog box. These boxes, the problems they indicate, and the solutions to those problems, are described in this appendix.

### CPN Settings File Missing or Obsolete

When you try to start Design/CPN, and the .CPNSettings file is missing or obsolete, Design/CPN displays a dialog that states:

**Your .CPNSettings file was unreadable or not found. To start up, generation of a new .CPNSettings file is required.**

The choices offered are **OK** and **Cancel**.

- Click **Cancel**.

Design/CPN quits.

### Problem Description

When Design/CPN is installed, a directory called Design/CPN is created. This directory contains a file called .CPNSettings. In order for Design/CPN to run, this file must be copied to your home directory. If the file was not copied there, or was subsequently renamed

## Design/CPN Tutorial

---

or removed, Design/CPN cannot find the settings it needs in order to run correctly. It therefore displays the above dialog.

### Problem Solution

If you have a copy of .CPNSettings in your Design/CPN directory:

- Copy .CPNSettings to your home directory.

If you do not have a copy of .CPNSettings in your Design/CPN directory:

- Reinstall Design/CPN from the source tape.
- Copy .CPNSettings to your home directory.

After you have copied the settings file:

- Start Design/CPN.

The application should now start without problems relating to CPN settings.

## ML Configuration Unspecified or Incorrect.

Before you can run the ML interpreter to syntax check or execute a diagram, you must use the **ML Configuration Options** command (**Set** menu) to tell Design/CPN:

- **Hostname:** Where the ML interpreter is to be run.
- **Port number:** Which port is used by the daemon is to run the interpreter.
- **ML image:** Where the interpreter is located in the file system.

Design/CPN keeps this information as system defaults. In order for a particular diagram to use it, the information must be present in its diagram defaults.

### Identifying the Problem

If any ML configuration information is missing or incorrect, Design/CPN will be unable to start the ML interpreter, and will display a dialog box. The dialog displayed depends on which information is faulty. If more than one of the parameters is faulty, the dia-

log that appears will describe the first erroneous parameter encountered.

If the **Hostname** is missing or wrong, the dialog offers an opportunity to log in, with **OK** and **Cancel** buttons. Attempting to log in will fail, so:

- Click **Cancel**.

If the **Port number** is missing or wrong, the dialog states:

**Network not responding. Connection with host could not be established.**

The dialog contains an **OK** button.

- Click **OK**.

If the ML image is missing or wrong, the dialog states:

**ML could not be started. Check available memory and presence of default ML file.**

The dialog contains an **OK** button.

- Click **OK**.

You must now supply the necessary information. You may be able to do this by copying the system defaults, or you may have to supply correct information by typing it into the **ML Configuration Options** dialog.

### Copying Diagram Default ML Configuration Options

When the system defaults are correct, but the diagram defaults are not, the system defaults must be copied to the diagram defaults. This is typically necessary when the diagram was created on some other computer. It is also necessary if the system defaults have changed since the diagram was created. To update the diagram defaults:

- Choose **ML Configuration Options** from the **Set** menu.

The **ML Configuration Options** dialog appears.

- Click **Load**.
- Click **OK**.

## Design/CPN Tutorial

---

The necessary information about the ML process is copied from the system defaults to the diagram defaults. Design/CPN can now start the ML process and use it to syntax check and execute the diagram.

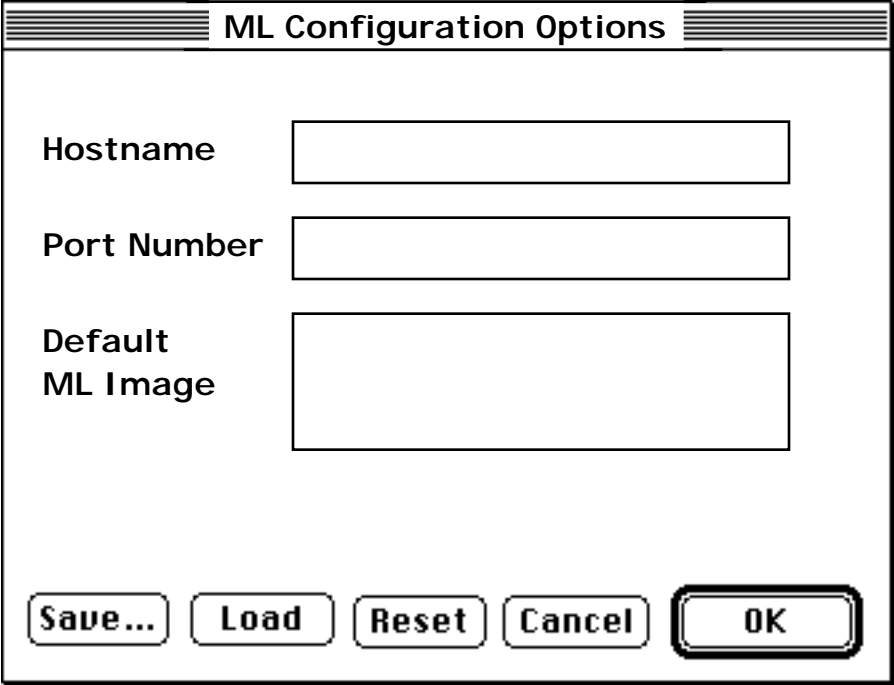
When the ML process fails to start, loading the system defaults is the thing to try first. If Design/CPN is correctly installed on your system, it will always work, and it can never do any harm since the diagram defaults were wrong in any case. If it does not work, the system default ML configuration options must be established or corrected.

### Setting ML Configuration Options

To establish new values for any or all ML configuration options:

- Choose **ML Configuration Options** from the **Set** menu.

The **ML Configuration Options** dialog appears:



The dialog box titled "ML Configuration Options" contains three input fields and five buttons. The fields are labeled "Hostname", "Port Number", and "Default ML Image". The buttons at the bottom are "Save...", "Load", "Reset", "Cancel", and "OK".

Field Label	Field Type
Hostname	Text input
Port Number	Text input
Default ML Image	Text input

Buttons: Save..., Load, Reset, Cancel, OK

- Enter correct values as needed for the **Hostname**, **Port number**, and **ML image** fields.

See the Design/CPN Installation Notes for information on determining these values.

If you want the new values to become system defaults:

- Click **Save**.

A confirmation dialog appears.

- Click **OK** in the confirmation dialog.

The values in the **ML Configuration Options** dialog are now the system defaults.

To make the new values the diagram defaults:

- Click **OK** in the **ML Configuration Options dialog** .

The values in the **ML Configuration Options** dialog are now the diagram defaults. The dialog closes. If the values are correct, you will now be able to use the ML interpreter to syntax check and execute the diagram.

## ML Interpreter Cannot Be Started

In order to run, the ML interpreter needs:

- A least 32 meg of RAM.
- At least twice as much swap space as RAM.

Larger allocations will result in better performance.

If the ML interpreter does not have enough RAM or swap space to run in, attempting to start it will display a dialog that mentions a lack of available memory. There are two possible solutions:

- Terminate other processes that are running in your memory space, freeing their memory allocations for use by the ML interpreter.

If there are no such processes, or terminating them does not free enough memory:

- Ask your system administrator to increase your memory allocation.





