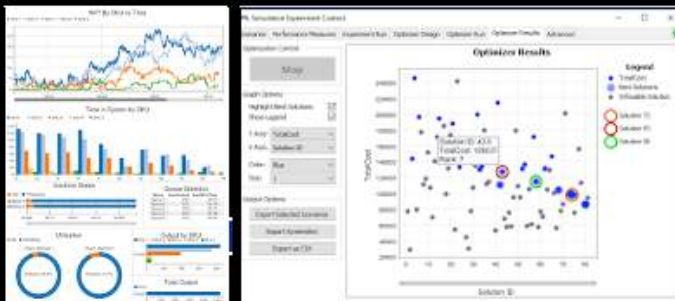
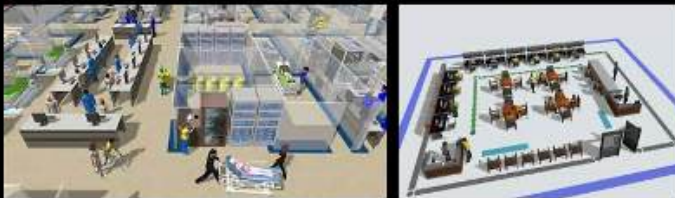




FlexSim Simulation Software

Primer



5th Edition

February 2025

Software Version 2025.0



Allen G Greenwood, Ph.D., P.E. (retired)

Simulation Education Specialist, Autodesk
Professor Emeritus, Mississippi State University

allen.greenwood@autodesk.com

Copyright © 2018 - 2025 Autodesk Inc. All rights reserved.

Published by FlexSim Software Products, Inc., The Landmark @ One Market, Ste. 400, San Francisco, CA 94105 USA.

FlexSim software and books may be purchased for educational, business, or sales promotional use. For more information, please contact our sales department +1-801-224-6914 or flexsim.sales@autodesk.com.

The *Autodesk* logo and the *FlexSim* logo are registered trademarks of Autodesk Inc. Other registered trademarks belonging to third parties are used within this work. Where those designations appear in this book, and FlexSim Software Products, Inc. was aware of a claim, the designations have been printed in italics, caps, or initial caps.

While every precaution has been taken in preparing this book, the publisher and the author assume no responsibility for errors, omissions, or damages resulting from the use of the information contained herein.

Version 1.1, for software version 2017 Update 2, May 2018

Version 2.0, for software version 2018 Update 2, September 2018

Version 3.0 for software version 2019 Update 2, December 2019

Version 4.0 for software version 2020 Update 1, June 2020

Version 5.0 for software version 2025, February 2025



ABSTRACT

This primer provides detailed, step-by-step instructions for building and analyzing a comprehensive simulation model in the *FlexSim* simulation software. It uses a single comprehensive example and a sequential development process. The system that is modeled is small but not simple. While focusing on basic concepts and constructs in *FlexSim*, the primer introduces some of the advanced features and capabilities available in *FlexSim*. The primer describes the following in detail.

- *FlexSim*'s basic fixed and mobile 3D objects (e.g., processors, conveyors, operators) that are powerful yet easy to use
- Building custom logic through the flowchart-like tool Process Flow
- Using powerful modeling tools such as tables, lists, reliability, charts, etc.
- Designing and running experiments with a simulation model to analyze the operational dynamics of a system
- Utilizing *FlexSim*'s array of modules to model AGVs, warehouses, people, etc.

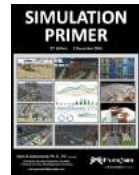
In addition to basic instructions, the primer provides insight and rationale for the described modeling actions and introduces good modeling and analysis practices.

The primer is not a software user manual, which are oftentimes reference documents. The primer provides a means to learn the basics of *FlexSim* simulation software, appreciate *FlexSim*'s power and capabilities, and apply the simulation problem-solving process.

The primer is mainly intended for someone who has little or no familiarity with *FlexSim* simulation software. However, in addition to new users, it is hoped that existing *FlexSim* users will find the primer helpful in clarifying some aspects of the software and learning something new.

While it is preferable that the reader be familiar with the basics of discrete-event simulation, it is not required. However, a basic introduction to simulation is available the following primer.

Greenwood, A. *Simulation Primer*, FlexSim Software Products, Inc., December 2024.



A folder of supporting files that are referenced in the primer are available, as are the *FlexSim* model files.

PREFACE

This primer is based on many years of experience and my long-time passion for simulation. I started learning simulation in graduate school and applying it in practice in my first position as an industrial engineer in the mid-1970s. Simulation has been a significant part of my professional life in various settings and situations. These include teaching many simulation courses at the undergraduate and graduate levels at several universities, both in the US and other countries; developing and delivering short courses for practicing engineers in a variety of industries; writing, presenting, and publishing research papers on various aspects of simulation modeling and analysis; and, carrying out many simulation projects in a range of industries.

I consider simulation an applied technology that is an essential tool for effectively supporting a broad range of problem-solving and decision-making processes. It is especially valuable for designing and managing systems in many domains, such as manufacturing, material handling, warehousing, logistics, healthcare, mining, business processes, etc. All of these domains are naturally complex – the complexities are due to the systems having many disparate elements, the relationships among those elements, inherent variability, and intrinsic dynamics. Simulation facilitates and encourages exploring and assessing multiple ideas and alternatives to “optimize” system performance. All of this is done virtually without disturbing an existing system or before a system exists.

Many good resources are available to learn about simulation modeling and analysis in general and *FlexSim* in particular, including textbooks (such as *Applied Simulation Modeling and Analysis Using FlexSim* by Beaverstock, Greenwood, and Nordgren), training and teaching materials, user manuals, tutorials, blogs, videos, etc. Each resource has its own objectives and is devised to meet specific needs. However, none of the resources by themselves meet all of the primer’s objectives:

- Provide detailed, step-by-step instructions for building a comprehensive simulation model.
- Use a single comprehensive example and sequential development process to effectively move from introducing and applying simple concepts and methods to describing those that are more complex.
- Offer insight and rationale for modeling actions rather than just specifying rote commands.
- Introduce good modeling and analysis practices.
- Introduce fundamental concepts of simulation.
- Offer references for further reading.
- Provide an awareness of some of the more advanced features available in *FlexSim* without covering the details at that time.

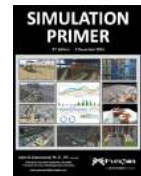
The primer focuses on the modeling and analysis of operations systems to understand and analyze their dynamic behavior and performance (referred to as *operations dynamics*) using discrete-event simulation. In the broadest sense, operations systems transform input into output through a set of related activities and processes that require various resources, such as equipment, material, people, and information. Transformations may be tangible (e.g., machining, inspecting, or delivering material in manufacturing) or intangible (e.g., diagnosing or treating patients in healthcare).

Simulating operations dynamics requires an accurate representation of the system being considered, including physical aspects (locations, distances, sizes, speeds, etc.) and the salient underlying logic (e.g., what action to take, when and where to perform an action, and using what resources to use). Visualization of the behaviors resulting from the interaction of the physical and logical aspects greatly increases the acceptance of modeling and analysis by decision makers and those who are not experts in simulation technologies and methodologies.

Visualization also enhances model validation and verification. *FlexSim* has rich visual and logical functionality and is easy to use. Of course, ease of use is relative, depending on a user's background and experience. Since *FlexSim* is so comprehensive, it is not feasible to cover all its capabilities in this introductory guide. The primer is designed to help users navigate the extensive capabilities of *FlexSim*. Completing the primer should provide the user with the background needed to explore more advanced features through the *FlexSim User Guide* or other resources.

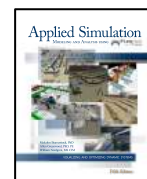
Similarly, the field of simulation is extensive; therefore, the theory and methods of simulation modeling and analysis are beyond the scope of this basic primer. Thus, for an introduction to simulation, the following primer is suggested. (This primer's author is also the simulation primer's author.)

Greenwood, A. *Simulation Primer*, Autodesk Inc., December 2024.



The following textbook is suggested for a more detailed discussion of simulation modeling and analysis concepts and practices and how they can be implemented in *FlexSim*. (The author of this primer is a coauthor of the textbook.)

Beaverstock, M., Greenwood, A., and Nordgren, W. *Applied Simulation Modeling and Analysis Using FlexSim*, 5th Edition, FlexSim Software Products, Inc., 2017.



Other sources of information on *FlexSim* include the *FlexSim User Manual*, in-software links to the manual, tutorials, and web resources, especially *FlexSim Answers*, which is a shared searchable knowledge base of questions and answers from FlexSim's worldwide community of users.

Again, it is important to note that this primer is not user manual for the software. User manuals are good reference resources to describe specific aspects of the software. In contrast, this primer is a means to learn both the basics of *FlexSim* simulation software and to be introduced to its power and capabilities in a way that demonstrates how it relates to the simulation problem-solving process.

It is suggested that this primer be covered carefully and methodically in the order it is written. The material builds from very simple aspects of simulation and modeling to the more complex. However, someone with more experience with simulation and the software may skim the early material, but it still should be considered since, at a minimum, it provides context. Then, later sections may then be considered in more detail to enhance understanding of a concept or software capability. Since all the models described in the primer are available, more experienced readers can start at a later section without building all previous models. Of course, providing all models also benefits those new to the software.

Allen G Greenwood

revised February 2025, High Point, NC, USA
revised December 2019, Greensboro, NC, USA
September 2018, Blowing Rock, NC USA

TABLE OF CONTENTS

ABSTRACT	2
PREFACE.....	3
TABLE OF CONTENTS	5
1 INTRODUCTION.....	10
1.1 OBJECTIVES	10
1.2 STRUCTURE	10
1.3 APPROACH.....	13
1.4 TESTIMONIALS	14
1.4.1 <i>Why FlexSim simulation software?</i>	14
1.4.2 <i>The road to choosing FlexSim</i>	16
PART I – GETTING STARTED WITH FLEXSIM.....	18
2 GETTING STARTED WITH FLEXSIM SIMULATION SOFTWARE.....	19
3 FUNDAMENTAL CONCEPTS.....	22
3.1 SIMULATION MODELING AND ANALYSIS	22
3.2 SIMULATION USING <i>FLEXSIM</i>	23
3.2.1 <i>Object Library</i>	23
3.2.2 <i>Toolbox</i>	24
3.2.3 <i>Analysis</i>	24
3.3 NOTATION AND FORMATTING CONVENTIONS	25
4 FLEXSIM’S MODELING ENVIRONMENT	27
4.1 MAIN MENU AND MAIN TOOLBAR	29
4.2 MODEL EXECUTION TOOLBAR.....	30
4.3 3D MODEL VIEW WINDOW, MODELING SURFACE, AND MOUSE OPERATIONS.....	31
4.4 OBJECT LIBRARY AND TOOLBOX	32
4.5 OBJECT INTERFACE.....	33
4.6 HELP.....	36
4.6.1 <i>User Manual</i>	36
4.6.2 <i>Online help</i>	37
5 BUILDING THE SIMPLEST SIMULATION MODEL.....	38

PART II – MODELING THE FINISHING AREA	44
6 THE EXAMPLE MODEL FOR THIS PRIMER	45
6.1 DEFINING THE SYSTEM AND PROBLEM	45
6.2 MODELING APPROACH	46
7 BASIC FIXED RESOURCES, FLOWITEM BIN, AND INITIAL CUSTOMIZATION	49
7.1 BASIC OBJECT PROPERTIES AND STRUCTURE	49
7.2 SOURCE OBJECT	51
7.3 FLOWITEM BIN	55
7.4 QUEUE OBJECT	58
7.5 PROCESSOR OBJECT	58
7.6 SINK OBJECT	60
8 BASIC MODEL OUTPUT	61
8.1 OBJECT STATISTICS AND PROPERTIES	62
8.2 DASHBOARDS	62
8.2.1 <i>Time-series plot</i>	64
8.2.2 <i>Time-series plot by type</i>	66
8.2.3 <i>Histogram plot</i>	67
9 TASK EXECUTERS	71
9.1 BASIC TASK EXECUTER CONCEPTS	71
9.2 ADDING A FINISHING OPERATOR TO THE MODEL	72
10 BASIC LOGIC WITHIN THE PROCESSOR OBJECT	79
PART III – FURTHER DEVELOPMENT OF THE FINISHING AREA	83
11 CUSTOMIZE OBJECTS TO REPRESENT SYSTEM BEING MODELED	84
11.1 CHANGING OBJECT GRAPHICS	84
11.2 CHANGING THE FINISHING MACHINE’S (PROCESSOR) GRAPHIC	85
11.3 CHANGING THE CONTAINER STORAGE’S (QUEUE) GRAPHIC	87
11.4 ITEM ROUTINGS	90
11.5 CHOOSING A ROUTING BASED ON AVAILABILITY	91
11.6 CHOOSING A ROUTING BASED ON CURRENT SYSTEM CONDITIONS	92
11.7 CREATING MODEL VIEWS	93
12 EMPIRICAL DISTRIBUTION AND DISTRIBUTION FITTING	95
12.1 USING AN EMPIRICAL DISTRIBUTION FOR PRODUCT MIX	95
12.2 USING AN EMPIRICAL DISTRIBUTION FOR DISTRIBUTION FITTING	98

13	MODEL PARAMETER & GLOBAL TABLES	102
13.1	MODEL PARAMETER TABLES	102
13.2	GLOBAL TABLE TO STORE ARRIVAL TIMES.....	104
14	DOWNTIME	107
14.1	TIME TABLES	107
14.2	CHART OF OPERATOR UTILIZATION AND STATES	112
14.3	RELIABILITY	114
14.4	CONSTANT MTTF/MTTR; MTBF BASED ON CLOCK TIME; NO RESOURCE FOR REPAIR	114
14.5	COMPOSITE STATE CHART	118
14.6	RANDOM MTTF/MTTR; MTBF BASED ON SYSTEM STATES; RESOURCE FOR REPAIR	120
14.7	STATE CHART FOR EACH FINISHING MACHINE.....	123
	PART IV – MODELING THE PACKING AREA	125
15	MODELING THE PACKING AREA – PART 1	126
15.1	DESCRIPTION OF THE PACKING AREA AND MODELING APPROACH	127
15.2	FLOW ITEMS FOR COMPONENTS	128
15.3	STORE OPERATION DATA IN TABLES	129
15.4	CREATING BATCHES OF COMPONENTS.....	130
15.5	QUEUE OF BATCHES AWAITING PROCESSING	134
15.6	STORAGE AREAS FOR COMPONENTS	135
16	MODELING THE PACKING AREA – PART 2	138
16.1	UNPACK COMPONENTS USING THE SEPARATOR OBJECT	138
16.2	FINISHING OPERATOR TASKS.....	141
16.3	CONTAINERS ARRIVE BY CONVEYOR	144
16.4	PACKING CONTAINERS BY TYPE USING A COMBINER OBJECT.....	146
16.5	PLACING COMPONENTS IN CONTAINERS BY ROBOT	152
16.5.1	<i>Defining the robot's operations.....</i>	<i>152</i>
16.5.2	<i>Downtime in the Packing Area</i>	<i>153</i>
16.6	CHECK COMPONENT INVENTORY LEVELS.....	157
	PART V – RESOURCE TRAVEL, EXPERIMENTATION, CONVEYORS, AND LISTS	160
17	MANAGING OPERATOR TRAVEL.....	161
17.1	CONTROLLING TASK EXECUTER TRAVEL WITH PATH NETWORKS.....	162
17.2	CONTROLLING TASK EXECUTER TRAVEL WITH A* NAVIGATION	168
17.2.1	<i>The basics of A* using a simple study model.....</i>	<i>168</i>
17.2.2	<i>Implementing the A* algorithm in the primer model.....</i>	<i>172</i>
18	EXPERIMENTATION IN FLEXSIM	176
18.1	EFFECT OF BUFFER SIZE ON PERFORMANCE	178

18.2	EFFECT OF COMPONENT REPLENISHMENT PLAN ON PERFORMANCE	183
19	MATERIAL MOVEMENT VIA CONVEYORS	187
19.1	GENERAL DESCRIPTION OF CONVEYOR OBJECTS.....	187
19.2	INCORPORATING CONVEYORS INTO THE PRIMER MODEL	191
20	USING AN ITEM LIST FOR COMPLEX MODELING LOGIC.....	199
20.1	PARAMETER UPDATES	200
20.2	SIMPLE PRIORITY ROUTING USING AN OBJECT TRIGGER	203
20.3	MULTIPLE-CRITERIA ROUTING USING LISTS	204
PART VI - MODELING USING PROCESS FLOW + WAREHOUSING AND AGV		212
21	INTRODUCTION TO PROCESS FLOW	213
21.1	BASIC PROCESS FLOW CONCEPTS AND MODELING ENVIRONMENT	214
21.2	SET UP TO USE PROCESS FLOW TO MODEL INVENTORY POLICY	217
22	MODELING INITIAL INVENTORY	219
22.1	A SIMPLE APPROACH	219
22.2	A MORE GENERAL APPROACH.....	225
23	MODELING INVENTORY REORDER POLICY	234
23.1	CHANGES IN 3D OBJECTS FOR USE IN PROCESS FLOW	234
23.1.1	<i>Add storage tables.....</i>	234
23.1.2	<i>Reset storage colors</i>	235
23.1.3	<i>Create a component order item.....</i>	236
23.1.4	<i>Remove objects for scheduled orders</i>	236
23.1.5	<i>Create a storage Group</i>	237
23.1.6	<i>Add and update component properties in tables</i>	237
23.2	IMPLEMENTING REORDER-POINT INVENTORY LOGIC USING PROCESS FLOW	239
23.2.1	<i>“Check inventory level” logic</i>	241
23.2.2	<i>“Reorder” logic</i>	245
23.2.3	<i>“No Reorder” logic.....</i>	255
23.3	VALIDATION & MODIFICATION	256
24	USE OF RACKS TO STORE CONTAINERS IN THE WAREHOUSE	261
25	ORDER FULFILLMENT SUBMODEL.....	269
25.1	DEFINITION OF THE ORDER-FULFILLMENT PROCESS	269
25.2	IMPLEMENTATION OF THE ORDER-FULFILLMENT PROCESS	270
25.3	MODELING ORDER-FULFILLMENT IN PROCESS FLOW	281
25.3.1	<i>Order-Fulfillment Process Part 1 - Generate Orders</i>	282
25.3.2	<i>Order-Fulfillment Process Part 2 - Complete Orders.....</i>	288
25.3.3	<i>Order-Fulfillment Process Part 3 - Fulfill Orders</i>	292

25.3.4	<i>Order-Fulfillment Process Part 4 – Initial Inventory Orders</i>	<i>295</i>
25.3.5	<i>Define and chart performance indicators for the order-fulfillment process</i>	<i>300</i>
26	AGV TRANSPORT BETWEEN PACKING AND WAREHOUSE	305
26.1	AGV PATH.....	307
26.2	CONTROL POINTS	309
26.3	AGV.....	311
26.4	CONTROL AREA	312
26.5	DASHBOARD CHARTS FOR AGV AND OTHER RESOURCES	315
	PART VII – SUMMARY AND APPENDICES	319
27	SUMMARY OF THE PRIMER MODEL.....	320
27.1	DESCRIPTION OF MODELING EACH MAJOR AREA OF THE FACILITY	321
27.1.1	<i>Finishing Area and Conveyor Transport.....</i>	<i>321</i>
27.1.2	<i>Packing Area and AGV Transport</i>	<i>324</i>
27.1.3	<i>Warehousing Area and Order Fulfillment.....</i>	<i>326</i>
27.2	KEY PROPERTIES OF EACH ASPECT OF THE MODEL.	328
27.3	EPILOGUE	334
	APPENDIX A – GLOSSARY OF KEY TERMS	336
	APPENDIX B – PROGRAMMING-BASED PROPERTY VALUES.....	345
	APPENDIX C – MODEL SUMMARIES	347
	ABOUT THE AUTHOR	351

1 INTRODUCTION

This section provides the primer's objectives, structure and organization, and general approach.

1.1 Objectives

As stated in the Preface, the objectives of this primer are to

- Provide detailed, step-by-step instructions for building a comprehensive simulation model
- Use a single comprehensive example and sequential development process to effectively move from introducing and applying simple concepts and methods to describing those that are more complex
- Offer insight and rationale for modeling actions rather than just specifying rote commands
- Introduce good modeling and analysis practices
- Introduce fundamental concepts of simulation
- Offer references for further reading
- Provide an awareness of some of the more advanced features available in *FlexSim* without covering the details at that time

1.2 Structure

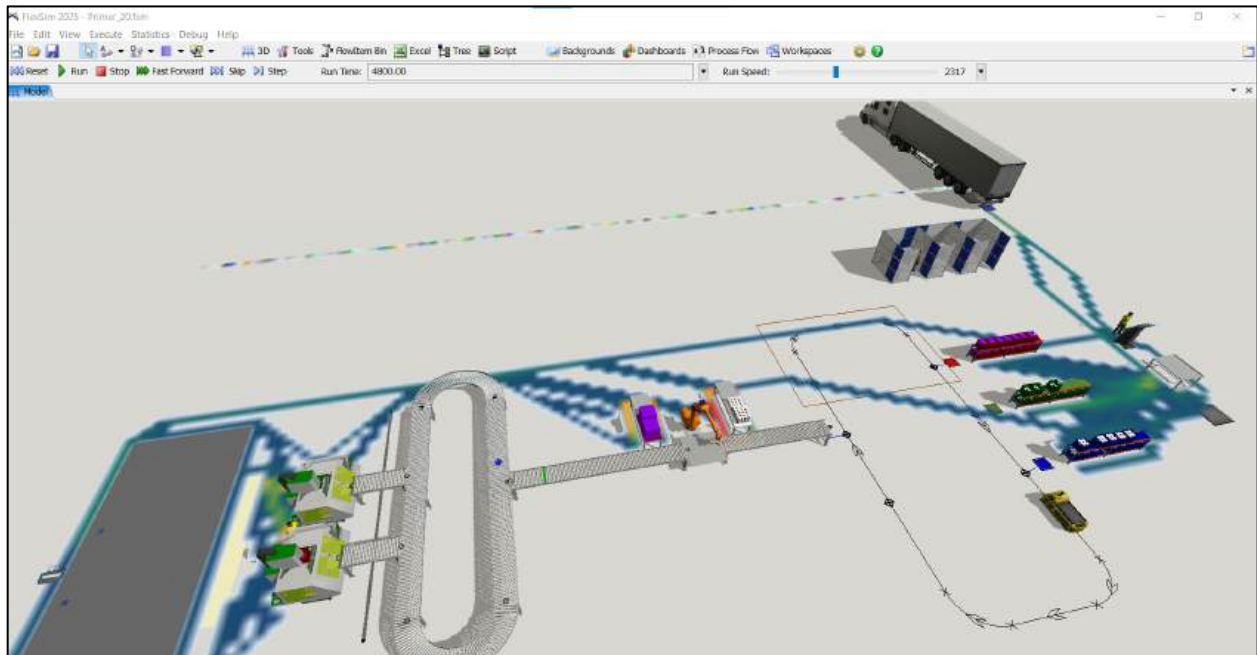
To meet these objectives, the primer is divided into seven sections and composed of 27 chapters and three appendixes.

This first chapter defines the primer's objectives, structure, and approach. It also addresses a common question that often arises from those unfamiliar with *FlexSim* simulation software – *Why should I use FlexSim?* Two long-time users of *FlexSim* provide testimonials about why it is their simulation software of choice and their path to choosing it. One testimonial is from an academic perspective, and the other is from an industry perspective.

After this introductory chapter, Part I – *Getting Started with FlexSim* is comprised of four chapters.

- Chapter 2 describes *FlexSim*'s features and capabilities at a very high level as well as licensing and how to obtain the software.
- Chapter 3 defines general simulation terms and terms that are specific to *FlexSim*, introduces major parts of the software, such as the Object Library, Toolbox, and means for analysis, and defines the notation and formatting used in the primer.
- Chapter 4 explores the *FlexSim* modeling environment, including menus, toolbars, modeling workspace, interfaces, etc.
- Chapter 5 describes in detail how to build the simplest simulation model, a simple queueing system.

The remainder of the primer describes simulation modeling and analysis using FlexSim by developing the model shown in the following screenshot. It is the conceptual design of a manufacturing facility composed of six main areas: finishing (machining), transport by conveyor, packing, transport by AGV, warehousing, and order fulfillment.



Part II – *Modeling the Finishing Area*. The modeling of the primer example begins by defining the system that the model will represent. It then describes the development of a simulation model of the first part of the production system, the Finishing Area.

- Chapter 6 defines the system that is modeled throughout the primer. It also describes the modeling approach that is used.
- Chapter 7 creates the initial model of the Finishing Area and introduces *FlexSim's* basic Fixed Resource objects.
- Chapter 8 describes basic object statistics and creates a dashboard with time series and histogram plots of measures of performance for the Finishing Area.
- Chapter 9 introduces mobile resources, referred to as Task Executors in *FlexSim*, and adds a Finishing Operator to the model.
- Chapter 10 looks into the Processor's operation, which is one of the basic 3D objects. It is meant to help the reader better understand what is happening behind the scenes in the basic 3D objects. Thus, this chapter does not add features and capabilities to the primer model.

Part III – *Further Development of the Finishing Area*. The model of the Finishing Area is enhanced by changing object graphics and routings and introducing some of *FlexSim's* many modeling support tools.

- Chapter 11 changes the Finishing Machine and Container Queue graphics and employs new item routing rules. It also introduces model views.

- Chapter 12 uses the Empirical Distribution tool to define the product mix of containers entering the Finishing Area. It also uses the Empirical Distribution tool to fit system data to a probability distribution that best describes the arrival process of containers to the Finishing Area.
- Chapter 13 introduces two types of data tables that help organize information used in *FlexSim* models: Model Parameter Tables and Global Tables.
- Chapter 14 adds several types of downtime to the model. Break and lunch times are added for the Finishing Operator, i.e., when the operator is unavailable to do system work. Two types of downtimes are considered for the Finishing Machines. The first is a quality check that occurs on a fixed clock-based schedule; the second is randomly occurring breakdowns that are based on the object's state. Machine breakdowns require a Finishing Operator for repairs; the quality checks do not require an operator. Pie charts are introduced to summarize the utilization and percentage of time the Finishing Machines are in various states. The charts are added to a new dashboard.

Part IV – *Modeling the Packing Area*. This section involves modeling the packing of finished containers with components.

- Chapter 15 first provides an overview of how the packing area works. Subsequently, component items are defined, as is how and when they are created (in batches). A data table is created to store information on the operation. Storage areas for the components are added to the model.
- Chapter 16 uses the Separator object to unpack batches of components when they arrive. The unpacking task requires the Finishing Operator. The Combiner object is used to model the packing of containers with components. A robot places components in containers. The Packing Robot is subject to state-based downtime like the Finishing Machines, and the Finishing Operator performs the repairs. A time series plot is added to show the inventory levels of the components over time.

Part V – *Resource Travel, Experimentation, Conveyors, and Lists*. The primer's simulation model is further developed by exploring and using several powerful features of *FlexSim* – alternative means to control task executor travel, experimentation and the analysis of scenarios, the use of conveyors to transport items, and the use of the List tool to implement more complex routing logic.

- Chapter 17 considers two ways to control Task Executor travel: via a path network and the A* algorithm.
- Chapter 18 introduces *FlexSim's* Experimenter, which provides a convenient means to consider simulating multiple scenarios and replicating each scenario. The Experimenter is used to study two aspects of the system modeled so far – (1) the effect of the size of the buffer of containers prior to Finishing on performance and (2) the effect of component replenishment plans on performance.
- Chapter 19 introduces the Conveyor objects and adds conveyors in the model to transport containers between the Finishing and Packing Areas and after packing to the warehouse.
- Chapter 20 adds more complex routing logic in the Finishing Area in terms of how containers are selected for finishing. The logic is implemented using the List tool.

Part VI – *Modeling Using Process Flow + Warehousing & AGVs*. This section uses *FlexSim's* powerful logic builder to complete the primer model. After an introduction to Process Flow, it is used to include initial component inventory in the model, implement a means to manage component inventory through a reorder-point system, and represent the order-fulfillment process. This section also introduces two additional *FlexSim* constructs that are key to modeling many operations systems – Racks and Warehousing objects and AGVs and their associated objects.

- Chapter 21 provides an introduction to Process Flow, including basic concepts and the modeling environment. It also describes some additions to the model that support the use of Process Flow.
- Chapter 22 describes several approaches to modeling the creation of an initial component inventory and explains how to implement the more general approach.
- Chapter 23 uses Process Flow to incorporate a reorder point inventory system to manage components.
- Chapter 24 introduces the Rack object and describes its use to store packed containers in the warehouse.
- Chapter 25 defines the order-fulfillment process and describes how to represent it in the model by using Process Flow. The process includes generating orders for containers, having an Order Picker gather the appropriate containers, delivering a completed order to a fulfillment area, and completing orders by updating an information system. Also, an output table is created that captures information on each order, including the contents of each order and the time it takes to fulfill orders. Charts are also added to track the time it takes to fulfill an order and how many orders are waiting to be processed.
- Chapter 26 implements an AGV system to transport containers from the Packing Area to the Warehousing Area. A Control Area is added to the AGV network to restrict traffic in the area to one task executor. A chart is added to track the AGV's utilization.

Part VII – *Summary and Appendices*. This section provides a summary of the primer model, three appendixes, and a brief bio of the author.

- Chapter 27 describes each major area of the model – Finishing Area, Conveyor Transport, Packing Area, AGV Transport, Warehousing, and Order Fulfillment – and summarizes key properties of the model.
- Appendix A is a glossary of key terms used in the primer.
- Appendix B describes and explains elements of *FlexScript* (a subset of C++) that are encountered in the primer when discussing setting some property values.
- Appendix C provides a brief description of the concepts and software features added in each primer model and a reference to the section where the model is discussed.
- About the Author provides a brief bio of the primer's author.

1.3 Approach

As the objectives above mention, the primer introduces and explores *FlexSim* through a single model. The model is small but not simple. Its complexity evolves throughout the primer as various aspects of the simulation software are presented.

Since the model of the example system evolves throughout the primer, it is recommended that each iteration of the model be saved. After each save, it is suggested that the current model file be copied and renamed so that one can always revert to a previous version of the model. This is just good modeling practice – models should always be developed in steps – build, test, and validate – and all models should start simple and then add complexity as needed. That is, the final model evolves from a number of simpler models with increasing representation of the real system. One should never try to incorporate all capabilities at the beginning of the modeling process.

To facilitate learning, all of the models developed in the primer are available so that if a reader has a problem developing any one of the primer models, they can access and view the correct implementation and then

proceed with the primer. Access to the primer models also allows a reader with some knowledge of *FlexSim* to skip the simpler parts of the model-building process and start at any section of the primer. For example, if a reader is well versed in the 3D aspects of *FlexSim*, but needs to become more familiar with Process Flow, then they can start the primer in Part VI and begin with the model from the end of Part V.

The *FlexSim* models from throughout the primer and model resource files (3D representations of work area objects and an *Excel* data file are in a folder named Resources) are available.

1.4 Testimonials

This section provides two testimonials, one from an academic perspective and one from an industry perspective, that discuss why *FlexSim* is their simulation software of choice and their path to choosing *FlexSim*. Each testimonial is from a long-time user of *FlexSim* and co-author of the textbook *Applied Simulation Modeling and Analysis Using FlexSim*.

1.4.1 Why *FlexSim* simulation software?

Provided by: Allen G Greenwood, Ph.D., P.E.
Professor Emeritus, Industrial & Systems Engineering, Mississippi State University
Simulation Education Specialist, Autodesk Inc.

There are a number of simulation software products available in the marketplace. Some have been around for many years, while others are quite new. A common question in industry and academia is why *FlexSim* should be used instead of other software available in the market.

I believe that choosing a software tool should align with a person's general approach to modeling and simulation. Therefore, I briefly introduce my background to give context and perspective to my response to the posed question. I have been involved with simulation for nearly 50 years – using it to help solve problems in a variety of industries and teaching numerous simulation courses, both in the U.S. and abroad. Over the years, I have used numerous simulation software products in industry projects, in support of research, and in the classroom. I started using *FlexSim* in 2006 for industry projects and then, a few years later, transitioned my simulation courses to use *FlexSim*. Since then, I have co-authored the textbook *Applied Simulation Modeling and Analysis using FlexSim* (now in its fifth edition) and two primers - one on simulation in general and this one, which focuses on modeling and analysis using *FlexSim*. One of my long-term, foundational professional goals has been to enhance and increase the application of simulation, both in practice and academia, to support problem-solving and decision-making.

While many good simulation products are available in the marketplace, I find *FlexSim* to be the best for my approach to solving problems and educating others. Hopefully, this essay will articulate why *FlexSim* became my simulation software of choice.

FlexSim is a modern, comprehensive simulation modeling and analysis environment that supports problem solving and decision making in industry, as well as supports teaching and research in academia. However, it was *FlexSim* - more than any other software - that made it possible for me to focus on solving industrial problems

or teaching simulation concepts. It is a software tool that is extremely powerful yet easy to use and transition, as needed, from small, simple models to larger, more complex ones. This is important in practice for incremental model building – starting simple and adding complexity as required – as it allows all those involved in the simulation process to follow and understand the underlying approach. It is especially important in teaching to have students quickly build simple models that illustrate key concepts and then add the desired enhancements. *Flexsim* makes it easier to concentrate on solving problems or learning simulation concepts rather than dealing with the idiosyncrasies and learning curve associated with other software.

Being involved in various industries and trying to give my students a basic understanding of the applicability and power of simulation, I find that *FlexSim* can easily model a wide range of operations systems including, but not limited to, discrete-part and continuous (e.g., oil and gas, food processing) manufacturing, transportation, logistics, healthcare, mining, construction, business and service processes, etc. *FlexSim* also offers a variety of simulation modeling approaches. While the core technology is discrete-event simulation, continuous and hybrid systems can be modeled through a library of pre-defined yet customizable fluid objects. Agent-based simulation and Monte Carlo simulation can also be performed within *FlexSim*.

Standard discrete-event and continuous modeling functionality are implemented through easy-to-use 3D objects. Most importantly, they can be customized through common and intuitive user interfaces. The objects' processing logic, behavior, and appearance are modified through extensive drop-down menu lists and direct parameter specifications. An intuitive, flowchart-like logic builder is available for defining and refining inter-object and intra-object relationships and behaviors. While some software claim they do not require the use of detailed computer code, there are times when it is necessary. Many simulation projects I have been involved with during my career required incorporating some specialized, more complex logic and behaviors. I found that, when needed, *FlexSim*'s process logic builder (Process Flow) and inherent scripting language *Flexscript* (a subset of C++) are effective and convenient means for representing complex operations. Numerous commands and templates, i.e., prebuilt sets of code and logic, are available to facilitate customization and reduce the amount of programming that is required. *FlexSim*'s open, object-oriented, hierarchical software architecture makes it a powerful and effective modeling environment.

FlexSim models are built in a native 3D environment. Their effective animation facilitates model validation, enhances stakeholder communication and understanding, and increases confidence in modeling and analysis. From a student perspective, the models look realistic, which provides needed relevance and helps make simulation exciting. In addition to the objects, I regularly use the many powerful modeling tools that are available in *FlexSim*, such as dashboards, time tables for managing work schedules, reliability tables for managing downtimes, global data tables, global variables, lists, macros, animation editor, video and model fly-through support, etc.

The simulation of an operation is only as good as the model's representation of the system and the analyses that are performed using the model. I believe this is fundamental for students to realize. One of the reasons that I like *FlexSim* is that it provides a comprehensive and easy-to-use set of analysis tools, such as an experimenter for running multiple scenarios and replications simultaneously in a designed experiment (while taking advantage of the number of cores on your computer), statistical measures of performance, extensive plotting and charting capabilities, etc.

Simulation is only one tool for problem-solving; thus, communication with other supporting software is critical. For example, *FlexSim* provides a convenient means to transfer model data to and from spreadsheets, databases, and the popular high-level scripting language *Python*. *FlexSim* also offers a seamless interface for *OptQuest* to optimize performance by intelligently searching for the best solution, i.e., the best operating conditions for the system being modeled.

Another very important characteristic of simulation software is the company behind the product. FlexSim Software Products, Inc. was founded in 1993, and the first version of the *FlexSim* software was released in 2003. Customer support – for both commercial and educational users – is excellent. They genuinely want to help everyone effectively solve problems through simulation. The company continuously improves its software through new and improved modeling and analysis capabilities, more effective interfaces and methods, better graphics, etc. FlexSim strongly supports education through student and education licenses and various learning materials.

1.4.2 The road to choosing *FlexSim*

Provided by: Mal Beaverstock, Ph.D.
Manager of Business Simulation, General Mills (retired 2008)

Over a 12-year span, more than \$150 million in savings were attributed to simulation at General Mills, Inc. (GMI). Now retired, I look back and realize that it was achieved through maintaining a problem-solving approach to simulation and the tool that made it possible—*FlexSim*.

Early in my career, I lost faith in modeling and simulation. Attempts to use this technology to solve problems met with frustration and little results. Computing technology in the late 1960's made detailed mathematical models expensive, cumbersome, time-consuming, and widely inaccurate. Problems were actually solved sooner by the involved engineers before a modeling approach could gain any momentum.

The change started in the early 1990's when a co-worker at International Paper solved a problem by modeling the mechanics of a machine aided by software that greatly facilitated building differential equations. The speed and ease of using such a tool re-ignited my interest in modeling and simulation.

While managing a control and simulation group at General Mills in 1995, an *Extend* simulation showed a savings of over \$1M by eliminating a packing line in a new design. That got management's attention! However, the work was still slow going. Using simulation software meant spending more time working with their idiosyncrasies than thinking about the problem that had to be solved.

Despite the drawbacks, successful simulations still helped by increasing the productivity of a complex conveyor system, resolving design issues when sizing surge and storage tanks, and improving a batch operation to meet new production levels without any capital expense. However, using simulation remained a specialty for a few individuals and not a general tool.

A search began for a software tool that could be accepted and used by all levels of individuals interested in simulation, provide for easy communication of assumptions and results, allow a focus on process dynamics, be

flexible enough to handle complex functionality and be a fully supported commercial product. Simulations were developed using a variety of software, including *Extend*, *Witness*, *ProModel*, *Arena*, and *Excel*. None matched the requirements.

In 2002, General Mills acquired Pillsbury, where Taylor ED was used for simulation, and through their contacts, I learned about *FlexSim*. This newly-developed software showed great potential. It was based on their incorporating object-oriented software constructs – an approach I immediately recognized from my work some years earlier as head of Systems Research at the Foxboro Company. Their use of functional objects combined with state-of-the-art graphics was exceptional. It was the first time I saw such techniques applied to building a simulation tool and it set them apart from the rest of the field.

Bill Nordgren, President of FlexSim Software Products, Inc., had a similar vision to mine for a simulation tool. The resulting symbiotic relationship resulted in a product that met our functional requirements. A graduate student majoring in art at the University of Minnesota spent a summer creating 3D objects based on actual pieces of manufacturing equipment as well as objects representing General Mills products. The object interfaces were modified to reflect GMI terminology and contain design information about equipment, such as standard speed, costs, and reliability information. Stored in libraries, equipment could be inserted into a simulation model without any other programming. While looking like a custom tool, the simulation program was actually a library within standard *FlexSim* and could be changed back with a single click. Since *FlexSim* served as the functional base, complex logic and dynamic simulations were possible.

As a result of *FlexSim*, by 2004 all interested individuals, including engineers, managers, and production personnel on the plant floor, could interact and contribute to the building, running, and analysis of the simulations that were helping them. Because of the visualization, people could see **their** equipment making **their** products.

The interest in simulation increased exponentially. Simulations were built 30% faster with *FlexSim*. Capital projects using simulation came in on time and under budget. *FlexSim's* flexibility allowed for simulations to quickly provide a feasibility estimate for new manufacturing concepts, help in the design of new equipment by analyzing operator actions, optimize material prep operations, study personnel assignments and requirements, determine yogurt scheduling approaches, resolve problems in delivery and shipping areas; review scheduling impacts on entire production lines, and many others. The consistent *FlexSim* environment made the simulation effort easier to develop and understand when more complex logic was required.

While the innovative software design, development execution, and tool-like approach to *FlexSim* were major factors in my selection, the vision, responsiveness, and commitment of the entire FlexSim staff sealed the deal. As far as I was concerned, the decision to use *FlexSim* was a “no-brainer” and proved itself in practice.

PART I – GETTING STARTED WITH FLEXSIM

- Chapter 2 describes *FlexSim*'s features and capabilities at a very high level as well as licensing and how to obtain the software.
- Chapter 3 defines general simulation terms and terms that are specific to *FlexSim*, introduces major parts of the software, such as the Object Library, Toolbox, and means for analysis, and defines the notation and formatting used in the primer.
- Chapter 4 explores the *FlexSim* modeling environment, including menus, toolbars, modeling workspace, interfaces, etc.
- Chapter 5 describes in detail how to build the simplest simulation model, a simple queueing system.

2 GETTING STARTED WITH FLEXSIM SIMULATION SOFTWARE



Chapter 2 describes *FlexSim*'s features and capabilities at a very high level as well as licensing and how to obtain the software.

FlexSim is a powerful, yet easy to use, environment for developing and analyzing simulation models of complex operations systems. Some of its features and capabilities include the following

- Comprehensive libraries of modeling objects and tools
- Modeling directly in 3D
- Complementary means to develop simulation models, such as customization of standard objects via consistent, intuitive interfaces, developing complex process logic through a flowchart-like logic builder, and specifying behavior through custom coding with a scripting language that is a subset of C++
- Open, object-oriented, hierarchical architecture
- Links to spreadsheet, database, *Python*, statistical, optimization, and graphics software
- Experimenter for designing, executing, and analyzing multiple scenarios and direct connection to *OptQuest* for optimization
- General modeling environment to support problem solving and decision making in diverse domains and applications.



- ...

This primer is based on the *Student* version of *FlexSim*, which is a limited version of the *Enterprise* or *Educational* versions. The Student version is limited in the number of objects and Process Flow activities that a model may contain, but has all of the capabilities and feature of the professional version. All of the versions require a license.

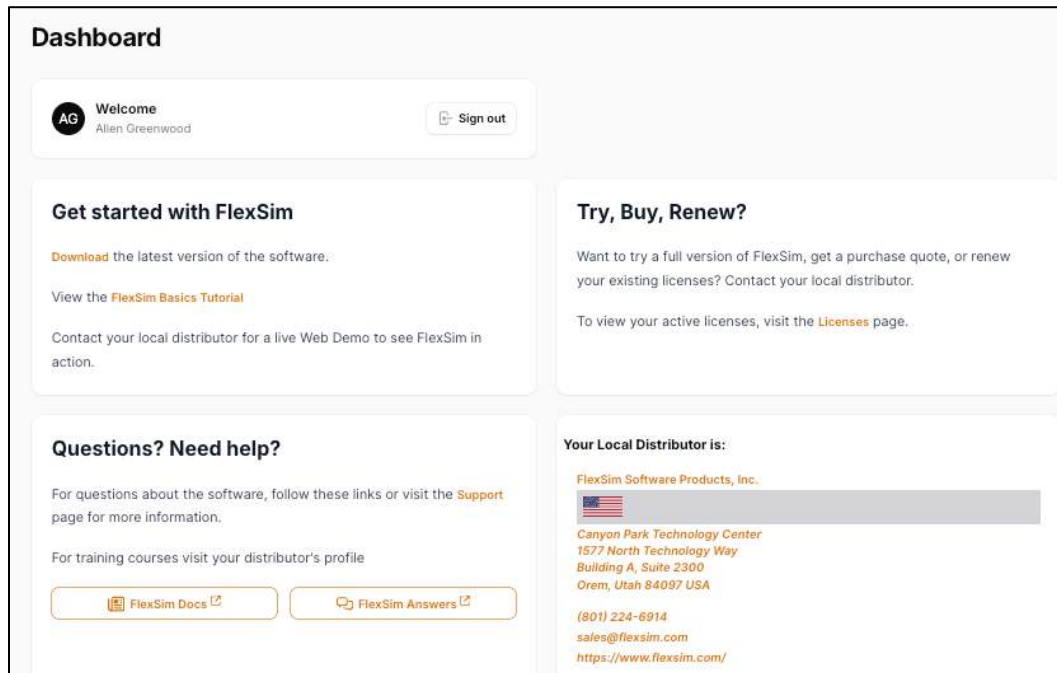
FlexSim Express is available for free download from flexsim.com, but it is primarily for evaluation purposes. As a result, the number of objects that can be used in a model is very limited, and many of the key features of *FlexSim*, such as full customization and the Experimenter, are unavailable.

While FlexSim Software Products, an Autodesk company, provides access to all versions of its software on its website, flexsim.com, it promotes two versions of *FlexSim* at any time - Current and LTS (Long-Term Support). At the time of writing this primer, the current is version 25.0.2 (release date 2025-01-17), and the LTS is version 24.0.8 (release date 2025-01_17). Three major releases occur within a calendar year: one at the beginning of the year (25.0) and two updates, one in the March/April timeframe (25.1) and one in the July/August timeframe (25.2).

The current version of the textbook *Applied Simulation: Modeling and Analysis Using FlexSim* 5th Ed. (2017) is based on the version 2017.0.13.

This primer is based on the Current version – 2025.0.2, release date 2025-01_17). While many of the basic operations of the software and user interfaces are the same or very similar, there may be differences with other versions.

To obtain this version, log into your FlexSim account to obtain the following dashboard. Obtaining an account is free and only requires some basic information.



Select **Download** from “**Download** the latest version of the software,” which will result in the following interface, then download the current version.

Downloads

FlexSim 2025

Current
Version 25.0.2
Release Date: 2025-01-17

[Release Notes](#)
[System Requirements](#)

Download
installer 687 MB

Other downloads:
[FlexSim_25.0.2_x64.msi](#)
[UserManualInstaller_25.0.2.msi](#)

More Versions

FlexSim 2024

LTS [Long Term Support](#)
Version 24.0.8
Release Date: 2025-01-17

[Release Notes](#)
[System Requirements](#)

Download
installer 678 MB

Other downloads:
[FlexSim_24.0.8_x64.msi](#)
[UserManualInstaller_24.0.8.msi](#)

More Versions



Software

Modules

FlexSim Models

Educator Materials

3rd Party Modules

Links

FlexSim 2025

Version 25.0.2 Release Date: 2025-01-17

[Release Notes](#)
[System Requirements](#)

Powerful computer software for modeling, analyzing, visualizing, and optimizing any imaginable process.

FlexSim_25.0.2_x64.msi	686 MB	msi	64bit	
UserManualInstaller_25.0.2.msi	349 MB	msi	64bit	
FlexSim_25.0.2_x64_installer.exe	687 MB	installer	64bit	

More Versions

FlexSim Healthcare 5.3.10

Release Date: 2019-02-19

[Release Notes](#)
[User Manual](#)

Experience our problem-solving, money-saving, stat-crunching, patient-centered healthcare simulation software.

Download
installer 368 MB

More Versions

FlexSim Webserver 2024

Version 24.2.0 Release Date: 2024-10-09

Compatible with FlexSim 24.2 and later.
A web communication interface for FlexSim that enables you to run FlexSim models on a web server and view them through a web browser on a client device.

Download
installer 1.52 MB

More Versions

For earlier software versions, click **More Versions** (located just below the Download buttons). The installers for each previous version of *FlexSim* are available for download.

3 FUNDAMENTAL CONCEPTS

Chapter 3 defines general simulation terms and those specific to *FlexSim*, introduces major parts of the software, such as the Object Library, Toolbox, and means for analysis, and defines the notation and formatting used in the primer.

This primer begins with a short introduction to simulation modeling and analysis, an overview of *FlexSim*'s general approach to simulation, and the notation and formatting conventions used in this document.

3.1 Simulation modeling and analysis

Simulation is used to analyze and solve problems and support decision-making. It is used to:

- Understand a system's behavior, especially its dynamics
- Analyze and predict a system's performance
- Compare alternatives for improvement
- Make the best decision for change

Simulation is composed of two key parts, modeling and analysis:

- Simulation **modeling** is a means for representing a system physically and logically to understand its behavior over space and time and to virtually assess possible consequences of actions.
- Simulation **analysis** uses a simulation model to experiment with and test ideas and alternatives before deciding actions and committing resources.

While many things can be simulated, this primer and *FlexSim* focus on the simulation of **operations systems**—systems that transform input into output through a set of related activities and processes requiring a variety of resources, such as equipment, material, people, and information. Transformations may either be tangible (machining, inspecting, or delivering material in manufacturing) or intangible (e.g., diagnosing or treating patients in healthcare).

In order to simulate operations systems, three key aspects must be addressed: **interactions**, **variability**, and **dynamics**, since these are all inherent in operations systems. There are complex relationships among a system's resources (material, equipment, people, and information), which interact in numerous and complex ways. In even the simplest system, there are sources of variability – arrival times and rates, process times, product mix, downtime, and quality level – to name a few. Some sources of variability are known, such as work schedules and possibly product mix (at least in the short term). Other types of variability are unknown, such as quality, process times, and breakdowns. Due to variability, the resource interactions change over time, thus resulting in the system's dynamics.

A simulation must represent the basic actions that occur in an operations system, e.g., processing, storing, and transporting items. The representation must consider physical aspects (e.g., size, distance, speed) and logical aspects (what, who, when, and where things are done, as well as how much and how long).

Four basic types of simulation are used to model and analyze operations systems: discrete-event simulation, Monte Carlo simulation, continuous simulation, and agent-based simulation. While all four of these types of simulation can be done using *FlexSim*, the most common, by far, is **discrete-event simulation** (DES). In DES,

the states of a system change at discrete points in time due to events occurring, such as a customer arriving at a system or the ending of a work shift. A system **state** is a condition of a system or value of a system variable, such as whether a resource is busy or idle or how many customers are waiting for a service.

3.2 Simulation using *FlexSim*

FlexSim is a comprehensive simulation modeling and analysis environment with extensive capabilities. Below are some, but not all, of its features and components. Most of these will be discussed in later sections of the primer; this is just an overview. Key terms are highlighted in bolded font.

3.2.1 Object Library

A library of modeling **objects** is available to rapidly represent key aspects of operations systems and create system dynamics. The objects are:

- Pre-built, yet customizable, representations of actions commonly found in operations systems, for example:
 - Planned and unplanned delays, such as process times, repair times, and waiting for resources
 - Transportation by means of both fixed objects (e.g., conveyors or robots) and mobile objects (e.g., operators, fork trucks, cranes, and AGVs)
 - Resource availability and reliability
 - Combining and separating items
 - ...
- Dragged from a library, dropped into a 3D model view, and connected to create a representation of the operation of a system.
- Customized to represent the characteristics of the system being studied by changing an object's **properties** or parameters.
 - The extensive set of object properties considers physical aspects (size, location, capacity, speed, etc.) and logical aspects (processing and routing rules, sequence of activities, availability, etc.), both of which either depend upon or influence a system's current state.
 - In addition to the many standard properties available in the objects, user-defined properties (called **labels**) can be defined, used, and updated anywhere and at any time during a simulation.
 - Each object has a consistent and friendly **user interface**. Object properties are conveniently grouped by panes on the object's property window. Many panes are the same across all types of objects, thus making learning the software much easier. Properties are specified through direct value entry and drop-down menu options on its pane.
 - Properties may also be customized and controlled through *FlexSim*'s built-in logic builder, **ProcessFlow**, or by writing computer code using *FlexSim*'s built-in scripting language, **FlexScript**, a subset of C++.
- Inherently 3D; thus, simulation models are created and run in a three-dimensional environment.
 - Any object's 3D shape can easily be customized. For example, 3D shapes created in *AC3D*, *3ds Max*, *SketchUp*, etc., can be directly imported into *FlexSim*, as can objects from the extensive, online, open-source *3D Warehouse*.
 - The software includes capabilities to “fly through” a model and to create videos of a model in action are built into the software.

3.2.2 Toolbox

The **Toolbox** is a library of tools that supports modeling building through, but not limited to:

- **Global Tables** store data and easily provide input to a model or obtain output from a model.
- **Lists** dynamically store and process information during a simulation.
- **Time Tables** specify deterministic resource availability, such as shift schedules, break times, periodic inspections, etc.
- **MTBF MTTR** provides means to specify downtime by:
 - Randomly making resources unavailable for a duration that is also oftentimes a random variable
 - Basing downtime on model states (e.g., processing and traveling) rather than clock time
 - Managing multiple or competing downtimes on the same object
- **Groups** denote objects that are similar or are acted upon in a similar manner.
- **Process Flow** builds complex inter-object and intra-object logic.
- **Dashboards** display model dynamics through charts, e.g., pie, histogram, and time series.
- **Excel Import/Export** provides direct links between *FlexSim* and *MSE_{Excel}* for inputting and outputting model data.
- Other types of tools include creating special events and actions, tracking and collecting information on a model variable, creating global variables and macros, developing a video of a model running, interacting with the popular scripting language *Python*, etc.

3.2.3 Analysis

Simulation models are built to perform analyses. *FlexSim* provides many tools for analytics, including:

- Information tracking
 - Automatically tracking many common measures of performance, such as throughput, content, utilization, etc.
 - Defining and tracking custom, system-specific measures
 - Charting to observe system performance on a dashboard as a simulation runs or to export results for reports and presentations
- Experimentation and optimization
 - Easy-to-use, built-in **Experimenter** for creating and comparing changes to sets of model parameters (**scenarios**), running multiple **replications** of a model, providing confidence interval estimates and other types of statistics, etc.
 - Direct interface to a leading **optimization** engine, *OptQuest*, to effectively search for the “best” set of model parameters to meet specified system objectives
- Export to analysis software – data can be exported to spreadsheets, databases, and graphing and special analysis software for further investigation.
- Data import – simulation models in *FlexSim* can be driven from:
 - Data stored in spreadsheets and databases
 - Data represented as an empirical distribution or fit to a probability distribution using *FlexSim*’s curve-fitting tool.

All model data are stored in an open, accessible hierarchical data structure referred to as the **Tree**.

Many of these features mentioned here will at least be introduced in this primer. However, a full discussion of many of them is beyond the scope of this introductory document.

The basic operation of *FlexSim* involves the creation and execution of **events** that are based on the logic specified in a model. The events generate actions and activities that occur over time. As a result of the events occurring and/or the current state of one or more objects, items move or flow from object to object. In *FlexSim*, the items that flow through a model are called **flow items** or just **items** for short. Items typically move between resources, either **fixed resources** (e.g., machines, conveyors, and storage areas) or mobile resources (e.g., operators, trucks, and AGVs). Items move into and out of objects via **ports** and contain user-defined characteristics called **labels**. Mobile resources in *FlexSim* are called **task executors** since they execute a sequence of tasks such as travel, load, travel, and unload.

Throughout a simulation, information on a system's conditions (or states) is gathered, summarized, and used for analysis. Typically, the summary information, such as average state values (average utilization, average number waiting for service, etc.), are used to compare the performance of alternative systems.

3.3 Notation and formatting conventions

To enhance the readability of this primer, certain formatting conventions and notation are used throughout the document.

As introduced above, *FlexSim* includes many objects and tools to facilitate simulation model building and analysis. For readability, the names of these entities are capitalized, e.g., objects such as Processor and Source, and tools such as Dashboard and Time Table.

A significant part of modeling is customizing general objects, such as a Source or a Processor, to represent the system being considered. In *FlexSim*, this involves specifying property values for objects through a user interface that employs drop-down menus, picklists, text boxes to directly specifying values, etc. Therefore, for readability, the following are the conventions used to reference parts of the software.

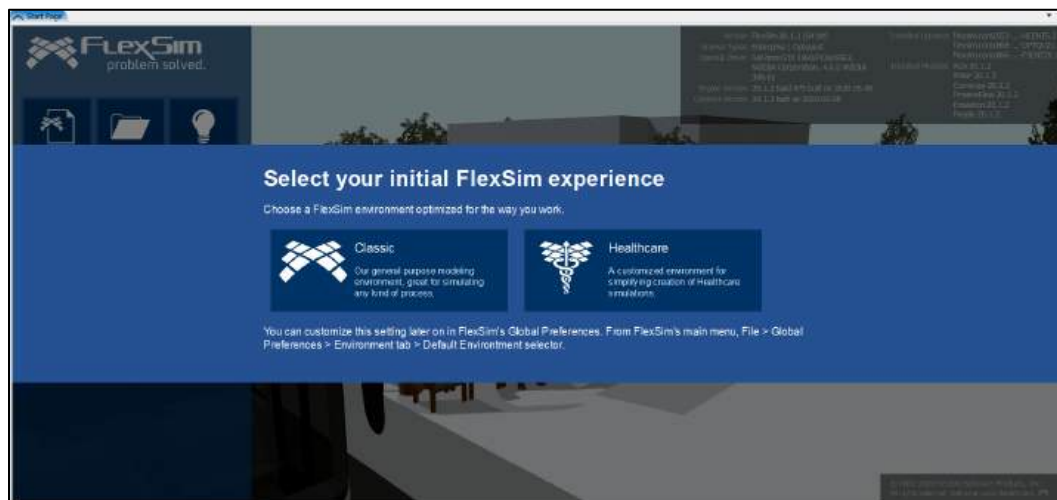
Conventions for referencing parts of the software

- Names of windows or interfaces are capitalized and in bold font, e.g., **Properties**, **Library**, **Toolbox**.
- Names of the window panes are capitalized and in bold italics font, e.g., ***Fixed Resources***, ***View Settings***, ***Visuals***.
- Names of Menus and Toolbars are capitalized and in bold font, e.g., **Main Menu** and **Main Toolbar**.
- Names of Menu and Tool options are capitalized and bold italics font, e.g., ***File***, ***Save 3D***, and ***Dashboards***.
- Names of objects are capitalized and in bold font, e.g., **Processor**, **Queue**, **Source**.
- Names of Process Flow activities are capitalized and in bold font, e.g., **Event-Triggered Source**, **Assign Labels**, **Decide**, **Delay**.
- Names of the properties or parameters of objects and Process Flow activities are in bold italics font, e.g., ***Process Time***, ***Send To Port***, and ***OnEntry***.
- The values of the properties, e.g. *1000.0*, *First Available*, *item.Type*, are italicized.
- Selection options within a drop-down menu, such as *Statistical Distribution*, *First Available*, and *Data > Set Label*, are italicized.
- Instructions to the reader for carrying out specific modeling actions are highlighted by separating them from the text and preceding them by the  symbol.

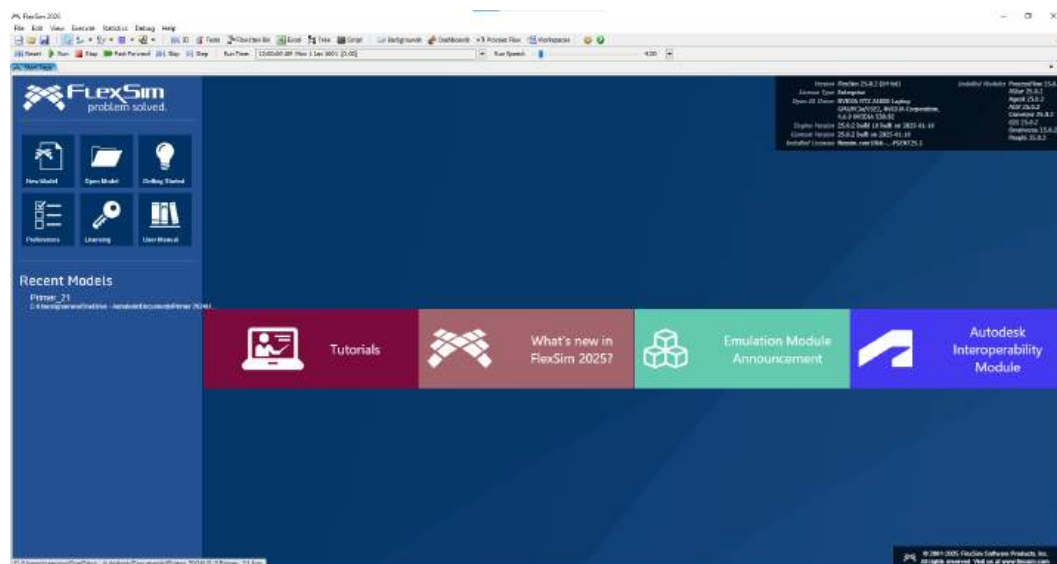
4 FLEXSIM'S MODELING ENVIRONMENT

Chapter 4 explores the *FlexSim* modeling environment, including menus, toolbars, modeling workspace, interfaces, etc.

When *FlexSim* is launched, by double-clicking the *FlexSim* icon, the first choice is the desired modeling environment, either **Classic** or **Healthcare**, as shown below. This primer only addresses the **Classic** environment.



Classic *FlexSim's* Start Page is shown below. The main interface is in the upper left-hand portion of the page, where buttons are provided for starting a new model, opening an existing model, setting preferences, checking licensing information, and accessing the *User Manual*. Below the buttons are a list of recently-accessed model files. A model file can be launched by clicking on its name in the list.



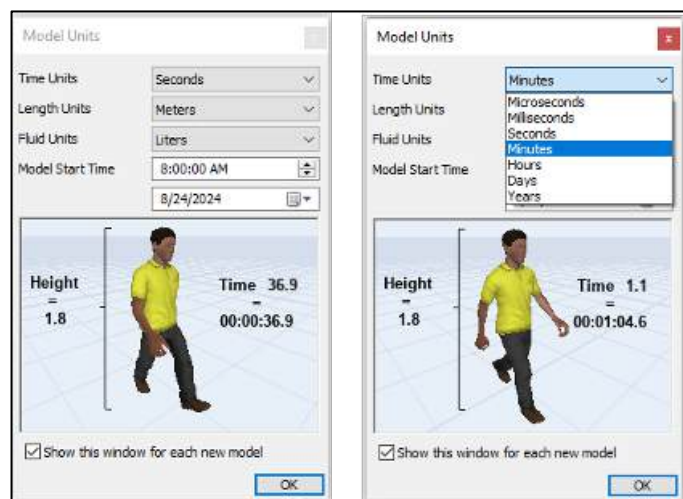
The upper right-hand portion of the interface provides detailed information on the software version.

The tiled images in the main portion of the interface link to FlexSim's website for more information on the software. These links will not be displayed if not connected to the Internet.

Once a model has been created, it can be accessed directly by double-clicking on the model file without going through the Start Page.

The file extension for *FlexSim* models is fsm. *FlexSim* automatically creates a backup of the current model through an autosave feature. The default time between saves is 10 minutes, which can be changed through Global Preferences. The autosaved file is named <your_filename>_autosave.fsm. Note that the model is only saved if it is not running; i.e., if a scheduled save occurs when a model is running, then the save does not occur.

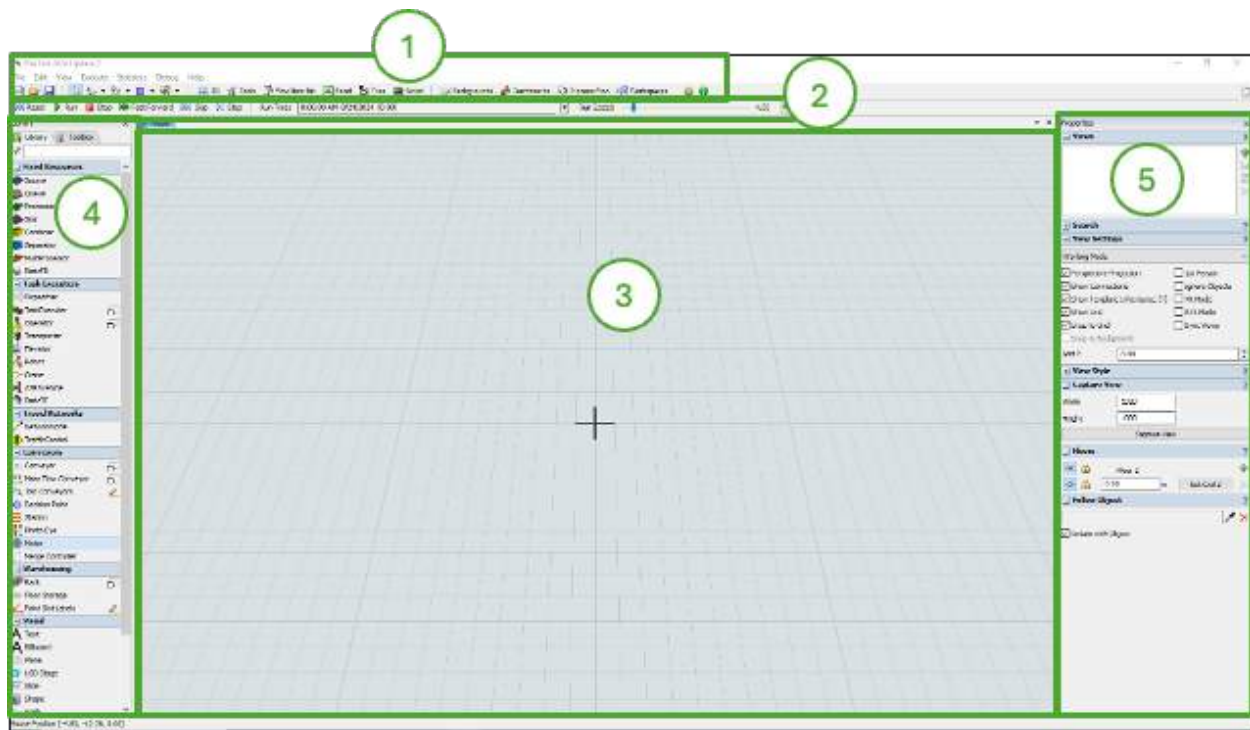
When starting a new model, the first information that is required is the model's units of measure. This is provided through the **Model Units** window, as shown in the figures to the right. *FlexSim* is unitless, i.e., simulations are conducted using general time units and distance units. Therefore, it is up to the modeler to specify the units appropriate for the system being modeled. As shown in the second figure to the right, different model units are available through a drop-down menu for each unit of measure. Note that the default units are metric; i.e., length (distance) is in meters, time is in seconds, and fluid (volume) is in liters.



This first step involves a very important decision since the specified **units cannot be easily changed in a *FlexSim* model once they are initially set.** However, there is a Measure/Convert tool within *FlexSim* to help with conversions.

Model Start Time can be changed at any time, but in most cases, its value can be ignored when starting a model.

The following figure shows the basic modeling environment and user interfaces in *FlexSim*. The numbers refer to the sub-sections below, where each interface is briefly described.



4.1 Main Menu and Main Toolbar

FlexSim's **Main Menu** and **Main Toolbar** are shown below.



The **Main Menu** functions the same way as in most *Windows* applications. The following defines some of the more commonly used options and those that may interest new users.

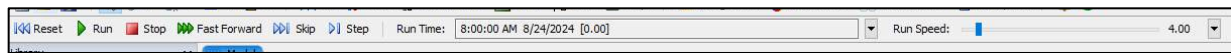
- **File** options are used to create new models, open existing models, save a model, save a model under a different name, and set Global Preferences for the *FlexSim* environment, including graphics card compatibility and customizing the toolbar.
- **Edit** options allow actions to be undone, as well as view and change model settings, etc.
- **View** is used to open various interface windows.
- **Execute** contains model-run controls like those in the Main Toolbar.
- **Statistics** is primarily used to launch the Experimenter.
- **Debug** is for advanced users and is not discussed here).
- **Help** provides access to the *User Manual* and license information.

The **Main Toolbar** is located below the **Main Menu**. It is composed of a set of icons that provide shortcuts to basic *FlexSim* operations and capabilities, such as saving a model, connecting objects in a model, opening

additional 3D views, accessing the **Toolbox**, linking to *MS Excel*, creating or accessing a Dashboard, creating or accessing Process Flow logic, etc.

4.2 Model Execution Toolbar

As the name indicates, the controls on this toolbar command a model's execution. As shown below, located below the **Main Toolbar**, it contains the following actions.



- Reset suspends model execution, removes all items from a model, resets all statistics, and sets the simulation clock back to 0.
- Run starts a model's execution either after a Reset or Stop.
- Stop pauses a model run; the run can be continued by pressing the Run button.
- Fast Forward moves a model forward to the next stop time.
- Skip advances a model to the next event's time. Since multiple events may execute at one time, this processes all events at that time.
- Step advances a model one event at a time.

Run Time shows the current simulation clock time, and its drop-down menu is used to set the planned stop time and any intermediate stop times.

There is no default planned stop time, so a model will run indefinitely unless suspended through Stop or Reset. The duration of a simulation run is in *simulated* time, not real time. As noted earlier, time in *FlexSim* is unitless; it is given context through the user's specification of time units (e.g., seconds, minutes, days) when a model is first created via the **Model Units** window.

If a user specifies the model units as seconds, then a Run Time of 10,000 is 10,000 seconds of simulated time (about 2.8 hours). Of course, it will take less than 10,000 seconds to run because simulations can run much faster than in real time. The model execution time depends on the Run Speed.

Run Speed is like *FlexSim*'s "gas pedal" – it controls how fast a simulation runs. It can be set with the slider bar or via its drop-down menu. If a model's units are seconds, then a speed of 1.0 means the simulation is running in real time, i.e., one second of simulation time is one second of real time. Similarly, if the speed is set to 100, the simulation is running 100 times faster than in real-time. Therefore, if a model's Run Time is 100,000 seconds (about 28 hours) and Run Speed is set to 100, then the 100,000-second (28-hour) simulation will take 1,000 seconds (about 17 minutes) to run. The Maximum option in the Run Speed drop-down menu is the fastest a simulation will run based on the host computer's capability. A simulation's speed can be adjusted as it runs to move quickly ahead in time to get to an interesting point in the model execution or slow down to watch a certain behavior carefully.

The **Experimenter** provides another means to run simulation models, which is discussed later in the primer.

4.3 3D model view window, modeling surface, and mouse operations

The model view window is *FlexSim*'s primary interface since this is where models are constructed in 3D. The modeling surface is a gridded infinite plane in the x and y directions and located at 0 in the z-direction. The x direction is to the left and right, the y direction is forward and backward, and the z direction is up and down.

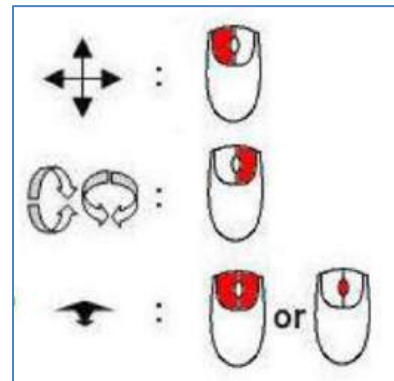
The black cross on the grid denotes the origin, where $x=0$, $y=0$, and $z=0$.

The grid units reflect the length units defined in the **Model Units** window when starting a new model.

As discussed in the next section, *FlexSim* objects are dragged into the model view and placed on the modeling surface. By default, they are placed on the surface, but they can be located above or below it based on the z-location parameter.

A mouse is used to navigate in the 3D model view. As shown in the figure to the right, the mouse movements are as follows:

- The left mouse button moves the view to the left and right and forward and backward.
- The right mouse button rotates the view.
- Using both mouse buttons or the scroll wheel zooms the view in and out.



To reset the view of a model so that it is centered at the origin and is in a flat, two-dimensional plane:

➤ Right-click anywhere on the modeling surface, then select *View >*, and then *Reset View*.

This two-dimensional view is helpful when placing objects in specific locations on the grid, connecting objects, and when a model appears lost somewhere in 3D space.

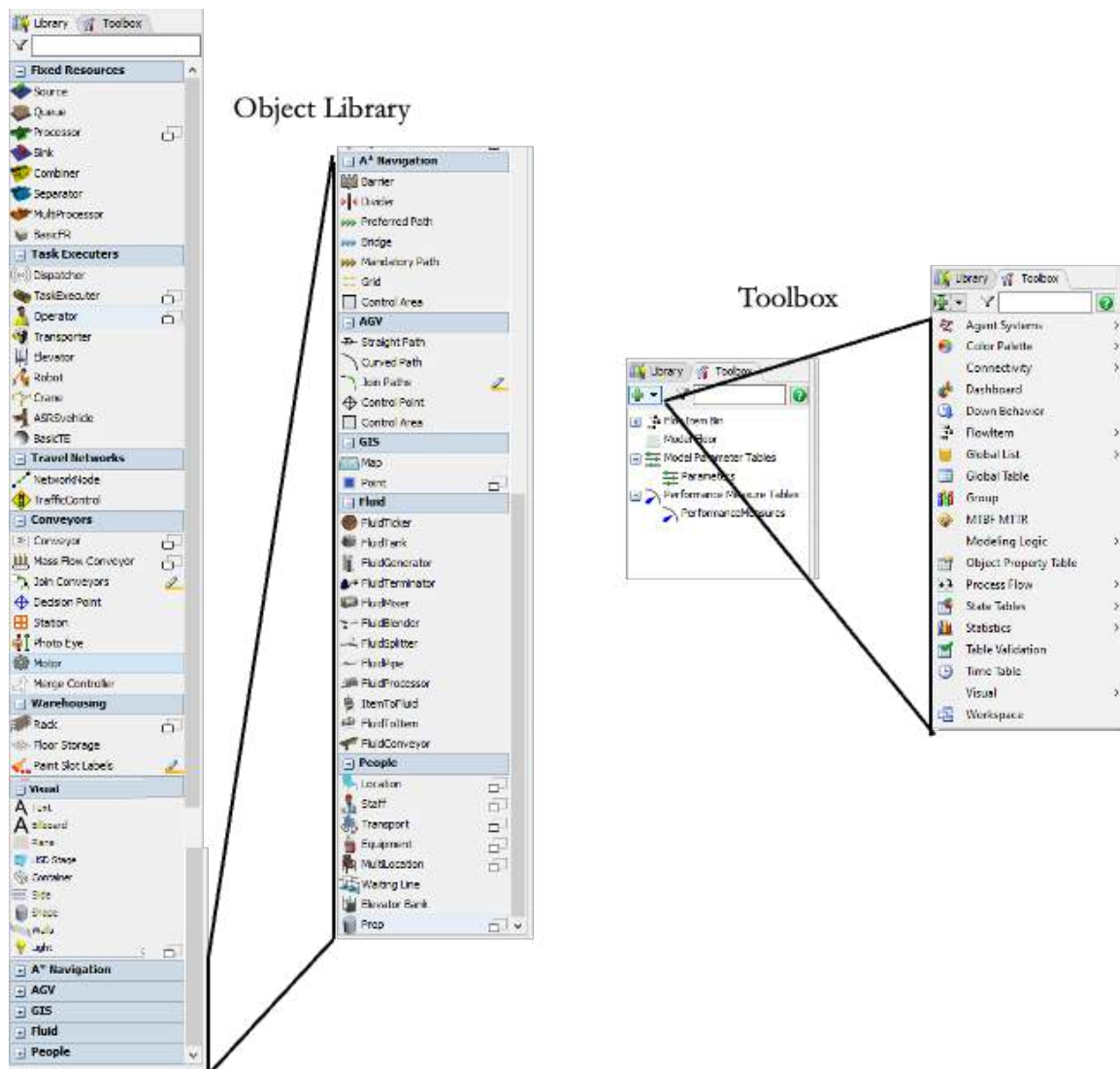
The modeling surface provides a 3D view of a model and is a tabbed interface in the modeling environment. As will be seen later, other things, such as dashboards and tables, can also be a tabbed part of the modeling environment. The tabbed views can be removed by clicking on the x button in the top-right portion of the view. In the case of the Model view, it can be reopened by pressing the **3D** button on the **Main Toolbar**. Therefore, *if you close the Model view, you only lose the view of the model, not the model itself!* Also, as discussed later, model views can be saved so that the saved view can be quickly recalled at any time.

Multiple model views can be created to view a model from various perspectives. Again, views are created via the **3D** button on the **Main Toolbar**. However, having too many views open can slow down model execution since the graphics on all of the views must be updated as a model runs.

Another approach for having multiple model views is via the **Views** pane in the **Properties** window. This will be discussed in a later portion of the primer. Again, closing views do not affect the model. Even if all views are closed, the 3D model is still there; it is just not displayed.

4.4 Object Library and Toolbox

Two of the main modeling resources in *FlexSim*, the **Object Library** and the **Toolbox** are shown in the figure below.



As mentioned earlier, models are created in 3D by dragging *FlexSim* objects from the **Object Library** to the **Model View** window and dropping them onto the modeling surface in the 3D space. The **Object Library** is also called the **Drag-Drop Library** in the **View** drop-down on the **Main Menu**. The library groups objects by major categories, such as Fixed Resources, Task Executors, Conveyors, etc. The library shown in the left portion of the figure above is the default view showing all object categories. The last category panes are expanded to the right, showing all the available objects in these categories.

Other types of object libraries can be used in a model, such as special-purpose and user-developed libraries of custom objects. However, this more advanced feature is only mentioned here to introduce a *FlexSim* capability.

The tab next to **the Object Library** is the tool library, referred to as the **Toolbox**. It provides access to a variety of modeling aids, such as data tables, timetables, dashboards, etc. Two versions of the Toolbox are shown in the figure above - the default list of tools used in a new model, and to its right, all of the types of tools that are available, such as Agent Systems, Color Palettes, etc. This list is accessed through the  button. Many of these will be discussed throughout the primer.

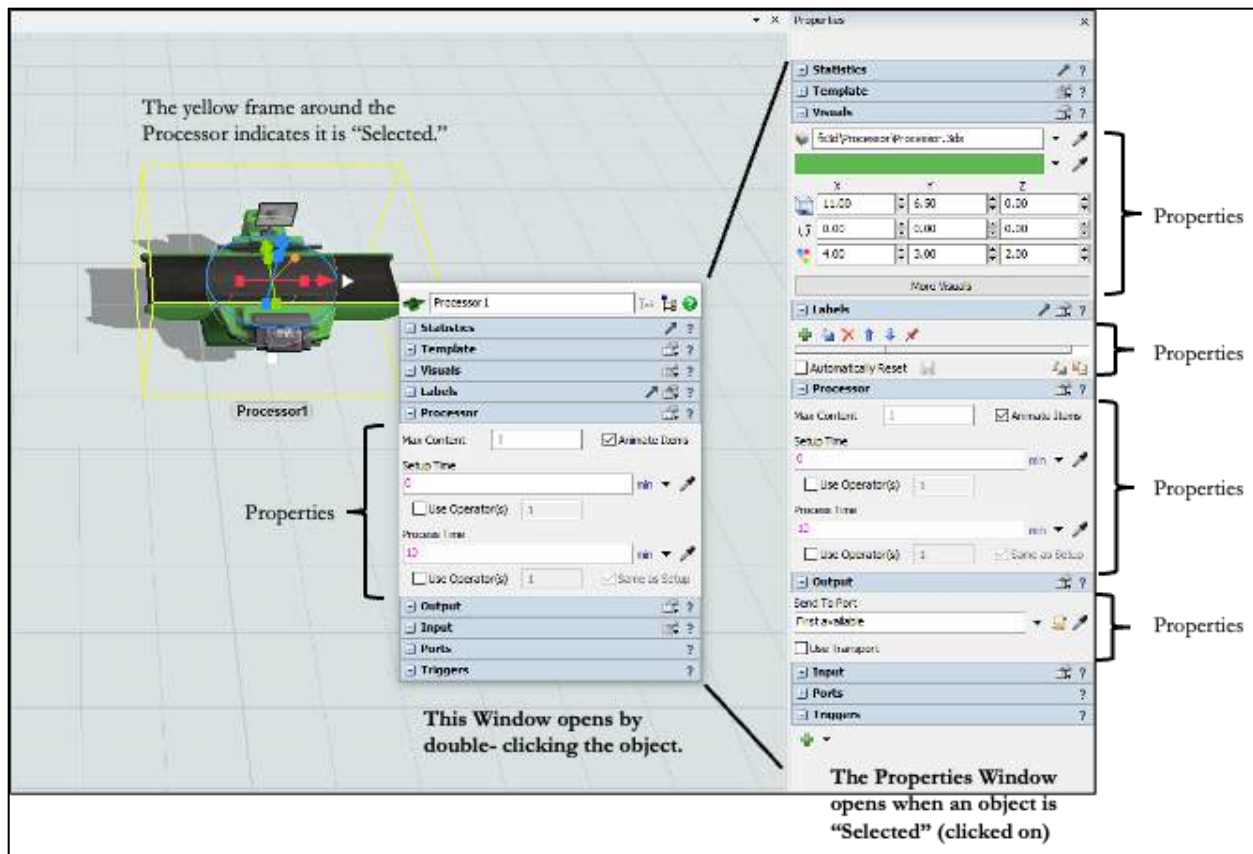
The various modeling objects and tools are introduced as the modeling example is built. While *FlexSim* contains a wide range of objects and tools to facilitate modeling diverse and complex systems, this primer only considers a subset of the capabilities. These key aspects are the ones needed to get started with *FlexSim*.

4.5 Object interface

As described earlier, at least for basic models, a significant part of model building is dragging and dropping pre-built 3D objects onto the modeling surface. The objects are selected, arranged, and connected to represent the behavior of the simulated system.

Each object represents a different functionality in operations systems and contains numerous properties available to customize the object's behavior during a simulation. The properties are grouped by panes on the object's user interface. Many of the panes are the same across all objects since all objects can be customized by size and shape, means for routing items to and from the object, defining custom attributes or characteristics (called labels in *FlexSim*), defining particular actions that occur within an object, etc. Having these common panes greatly facilitates learning and using the software. Each object has one or more panes containing properties specific to its functionality.

Using the Processor object as an example, the following figure illustrates some of the common aspects of 3D object interfaces. For the **Processor**, there is one unique property pane named **Processor**, and the common panes include **Statistics**, **Template Visuals**, **Labels**, **Output**, **Input**, **Ports**, and **Triggers**. The **Processor** object has numerous properties, such as **Max Content**, **Setup Time**, and **Process Time**. The properties' corresponding *values* are in the text boxes; for the properties above, the values are 1, 0, and 10, respectively. The values of the properties can be customized by the user/modeler.



The **Properties** window can be accessed in two ways. The first way is to "select" the object by clicking it once with the left mouse button. This results in the object's properties being displayed in the far right window of the environment, i.e., the Properties Window. If an object is selected, it will be outlined with a yellow box. The second way is to double-click the object. In this case, the **Properties** window will display next to the object.

The figure below shows the buttons *FlexSim* uses on its interfaces to help users provide property values. It also briefly defines the buttons' functions. The functions are grouped by those commonly used, especially by new users, and other buttons that are used to model more complex operations systems.

Property values can be set by many means, such as specifying a numeric constant, sampling from a probability distribution, looking up a value in a table, and following logic based on one or more states of the system. Specifying property values may involve providing parameters, such as the mean and standard deviation, when the property value is defined as a Normal probability distribution.

Functions of commonly-used interface buttons

-  Adds an item to a menu.
-  Deletes an item from a menu.
-  Opens a drop-down menu with choices for specifying the property's value.
-  The eyedrop-looking tool, called the “Sampler” is a convenient means to select other objects.
-  Opens the Users Manual to where the property is discussed, i.e., context-sensitive help.
-  Opens the pane in a separate window.
-  Indicates to enter text in the box.

Functions of other buttons on the interface

-  Opens the Users Manual to the Getting Started section.
-  Opens a view of the object in *FlexSim's* hierarchical Tree structure.
-  Opens the Code Editor to write custom FlexScript code to define custom logic.
-  Compares properties to other objects.
-  } Changes an object's properties by referencing another object.
- 

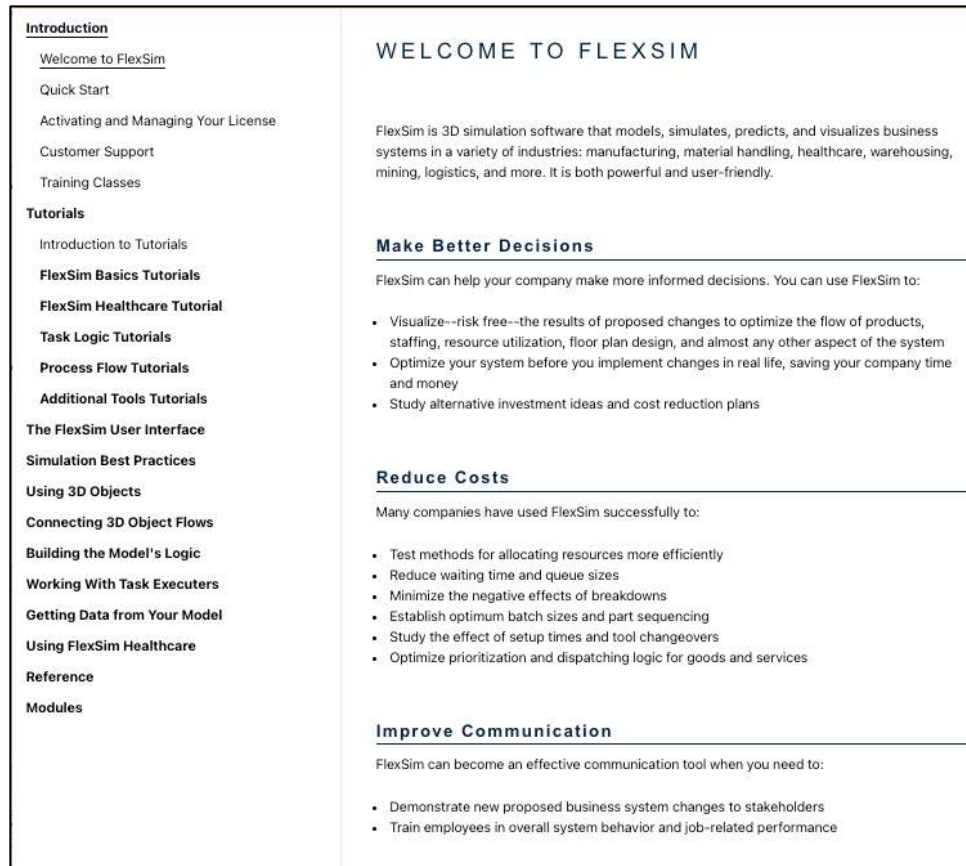
In addition to object properties, the **Properties** window is also used to change the properties of the 3D view. The view properties are available by clicking anywhere on the modeling surface. For example, multiple views of a model (e.g., an overview of the entire model or a closeup of an object or set of objects) can be defined in the **Views** pane. Another example is in the **View Settings** pane, which is used to toggle between the default *Working Mode* and *Presentation Mode*. The former is used for model building, and the latter hides the object-to-object connections, on-object statistics, etc., to make the model cleaner for others to view.

4.6 Help

FlexSim offers several ways to obtain clarifications and more details while using the software.

4.6.1 User Manual

One is via the **User Manual**; the first page shown in the figure below, along with its high-level table of contents. The manual is a key reference, but also contains step-by-step Tutorials, which are helpful for new users. It also contains a text search capability to find items of interest.



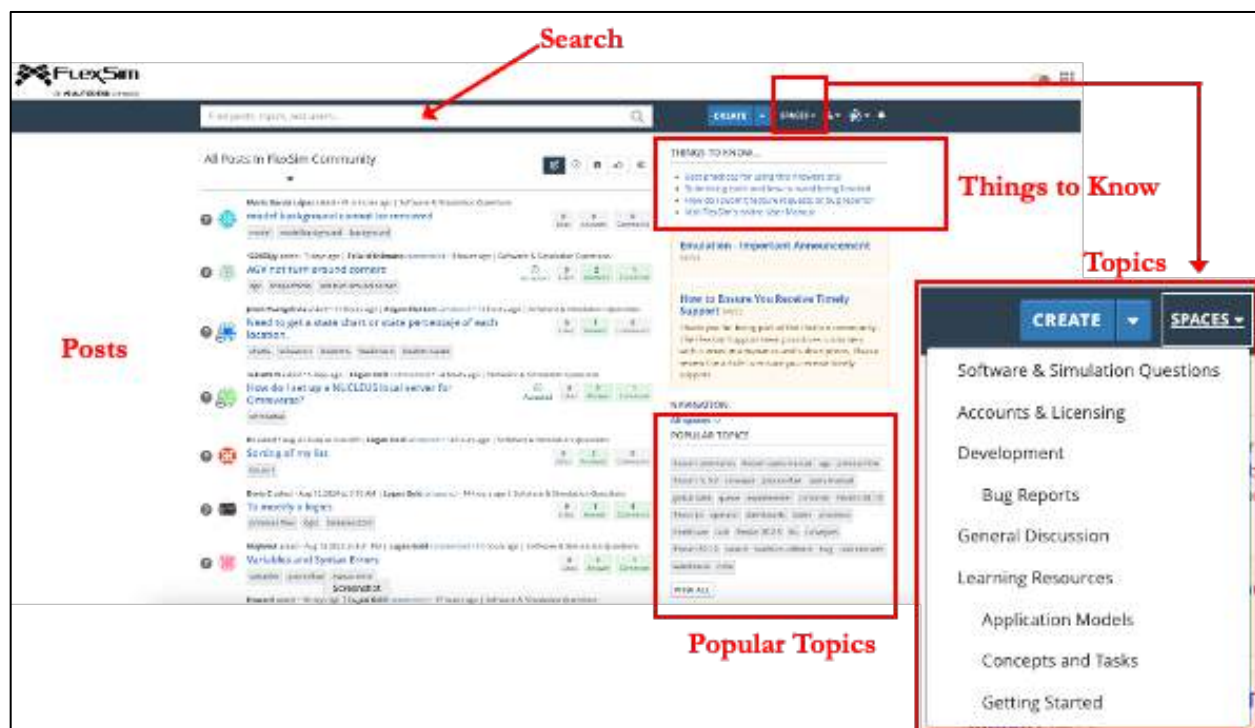
The manual can be accessed through several means in the software.

- The  button opens the **User Manual** at the location in the manual where the specific item in question is discussed, and the other button  opens the manual in the **Getting Started** section.
- Another option is via the **Main Menu** and select the **Help** option.
- The manual can be accessed outside of the software through FlexSim's website under the **Resources** menu.

4.6.2 Online help

Another source of help is the online site referred to as “**Answers.**” It is a part of FlexSim’s website and can be accessed under the **Services** menu or directly at <https://answers.flexsim.com/index.html>. As shown in the snapshot below, the site contains several helpful tools for obtaining help.

- **Posts** is a list of user questions and answers by FlexSim’s support team. It is a chronological list of posts, with the most recent at the top.
- **Search** is a means for users to enter the keywords of their question to see if it has already been answered. At least at the beginner’s stage, the question is usually addressed in the **User Manual** or through a post on this site.
- **Things to Know** – provides guidance on using the site effectively. All new users should review the first item, **Best Practice for Using the Forum**, before using the site.
- **Topics**, labeled **Spaces** on the site, groups the posts into main categories to help focus the question. The default Space on the site is *Software and Simulation Questions*.
- **Popular Topics**, as the name suggests, provides links to the most popular topics being addressed on the site.



5 BUILDING THE SIMPLEST SIMULATION MODEL

Chapter 5 describes in detail how to build the simplest simulation model, a simple queueing system.

Now that *FlexSim*'s basic modeling and analysis environment has been defined, let's build a model, albeit the simplest model! This model introduces the basic 3D modeling objects in *FlexSim* in terms of their functionality, structure, and properties and obtains basic output from the simulation.

Once some fundamental information is introduced, the primer provides a step-by-step guide to customizing the objects to represent a more complex system and introduce more of *FlexSim*'s functionality and capability. Part of that system will produce containers made on "finishing" machines.

In the primer, the ➤ symbol is used to highlight the steps the reader should take to build, run, and analyze a model.

To start a new model:

- Double-click the *Flexsim* program file or shortcut  and select "**New Model**" from the **Start Page**.

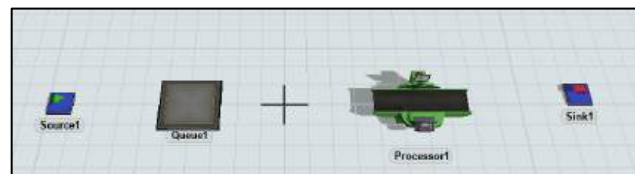
For this case, use the default units, except

- Set the time units to minutes; i.e., the model units should be meters, minutes, and liters. As discussed earlier, change the **Model Units** window as shown in the figure to the right.



Start with the most simple model.

- Drag the following objects from the **Object Library** to the 3D modeling view: **Source**, **Queue**, **Processor**, and **Sink**. The objects can be located anywhere but arranged similarly to the figure to the right.



These objects represent the first part of the production process, which later will be referred to as the finishing operation, and provide the basic functionality for the simulation model.

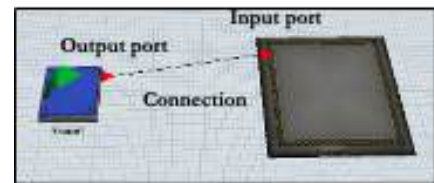
- The **Source** generates flow items, or for brevity, are referred to as items. When the object is customized to reflect our example, the **Source** will generate containers.
- The **Processor** represents the finishing operation in our example. By default, a **Processor** can only process one item at a time, but this property can be changed if needed.
- The **Queue** is used to retain items that have arrived but cannot access the finishing machine because it is busy.
- The **Sink** represents the next process, which is not modeled at this time. In our forthcoming example, items will move to a packing area, but that will be modeled later – we are starting small and simple.

While the four objects define the model's functional parts, the model is not yet complete - the relationship among the objects still must be defined. In this simple case, the objects are related by the sequential flow of the items between the objects. The item flow is defined by connecting the objects, and as with most aspects of modeling, there are several ways to do this. In *FlexSim*, there are several types of connections between objects; the item flow connection is referred to as an A-connection.

One approach for connecting objects for item flow, an A-connect, is to follow the sequence:

- Press the A key (notice the cursor changes from an arrow to links in a chain).
- Keeping the A-key pressed, click on the “from” object, where the flow item comes from; when doing so, the object becomes highlighted with a yellow box.
- Keeping the A-key pressed, drag toward the “to” object, where the flow item is going (notice a yellow line emanating from the highlighted object).
- Click on the “to” object.
- Release the A key.

An A-connect results in the two objects being connected with a thin black line as shown in the figure to the right. Also, two small red triangles appear on the edge of the two objects. The one on the right-hand side of the **Source** is its *output port*, from where items leave the object. The one on the left-hand side of the **Queue** is its *input port*, where items enter the object. Every A-connect automatically creates two ports – an output port on the “from” object and an input port on the “to” object.

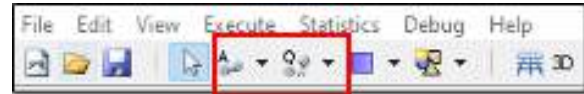


Be careful with the direction of the object connections since they affect the flow direction. The first object selected is the object *from* which items flow, and the second object selected is where items flow to. Of course, this can be checked by looking at the port connections, as shown in the figure above.

If a connection is to be removed, follow the same steps as defined above for the A-connection, but hold down the Q key instead of the A key. Notice the cursor changes from an arrow to links in a chain with one link broken.

An alternative connection approach is to use the **Connect Objects (A)** icon on the main toolbar next to the arrow icon, as shown in the red box in the figure to the right.

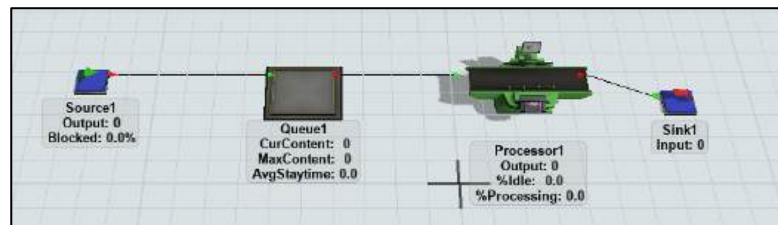
With this method, the A key is not held down, and objects are just clicked in the order that they need to be connected. Once all connections have been made, press the ESC key on your keyboard to return to Standard Mode, where the cursor appears as an arrow and not a chain. Similarly, the **Disconnect Objects (Q)** icon can disconnect objects.



A third way to connect objects is to use the orb surrounding an object when it is selected. As shown in the figure to the right, to connect the selected object to another object, select the small white triangle on the edge of the orb, shown in the red box in the figure. Then, drag the connector to the object to be connected until it is selected, i.e., highlighted by a yellow box. Again, an output port is created on the from object, and an input port is created on the to object.



- Connect the objects in the manner shown in the figure shown to the right using any of the three methods described above.

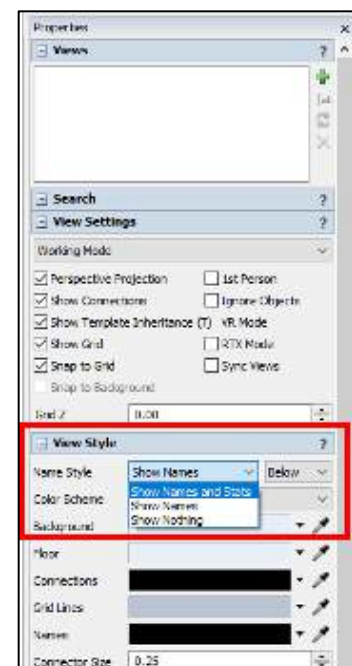


Note that a few statistics are shown below the objects in the figure above. The default option in *FlexSim* is to show only the object's name.

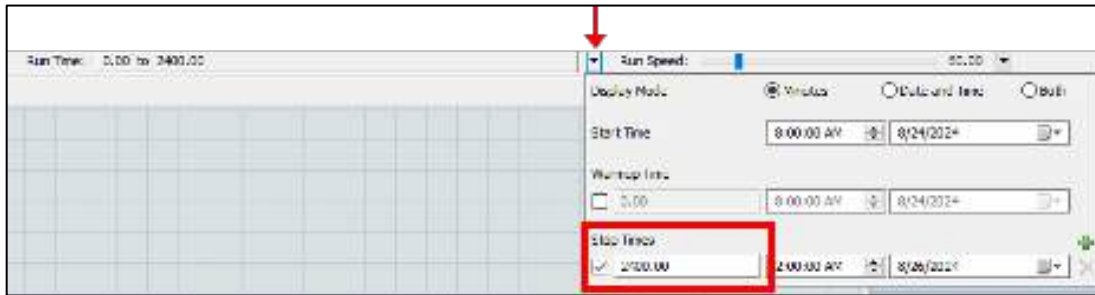
- To display the name and a few basic statistics, click anywhere on the modeling surface to obtain the **Properties** window shown in the figure to the right. Open the **View Style** pane and, as highlighted by the red box, use the dropdown menu to change the **Name Style** property from *Show Names* to *Show Names and Stats*. Notice that a third option is also available, *Show Nothing*, i.e., display no information about the objects.

The format of the information displayed for each object is a global property that pertains to all objects. The statistics displayed for each object are the *FlexSim* defaults, but as with most things in *FlexSim*, they can be customized.

The final task is to run the simulation. In this case, we'll run the model for 40 hours of simulation time. Since the model units are in minutes, the stop time is specified in minutes, i.e., 2400 minutes (40 hours * 60 minutes/hour). This is set in the **Model Execution Toolbar**.



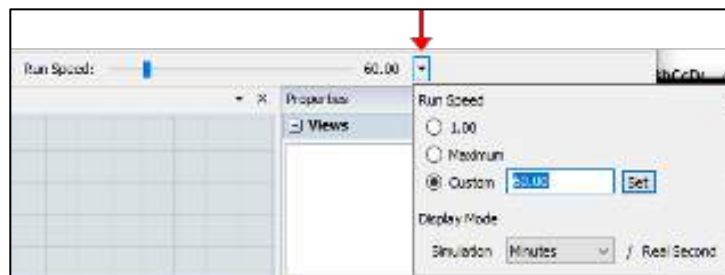
- Use the *Run Time* dropdown menu, as indicated by the red arrow in the figure below; note that it is just to the left of the *Run Speed* property. Change the **Stop Time** value to 2400. Once this change is made, the **Run Time** display will show 0.00 to 2400.00.



- Also, on the **Run Time** dropdown menu, change the *Display Mode* to *Minutes*.

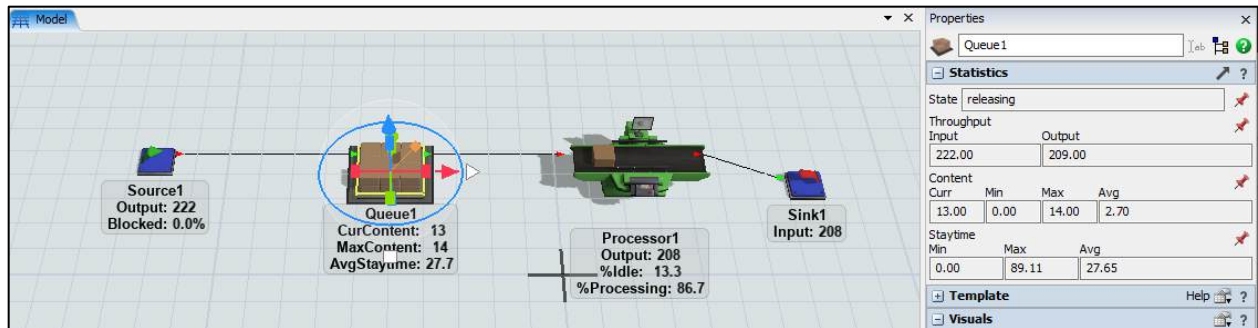
Now, consider how fast the model will run, which is controlled by the **Run Speed**, also on the **Model Execution Toolbar**. Since the model units are minutes and if the run speed is set to 1.0, the simulation will run one minute (60 seconds) of simulated time in one second. If the model units were seconds, then 1.0 would mean 1 second of simulated time would take about 1 second of real time, or basically, it would run at real-time speed. One of the benefits of simulation is being able to speed up (or slow down) time.

- Select the **Run Speed** dropdown menu as indicated by the red arrow in the figure below. In this case, change the **Run Speed**, as shown in the figure below, to 60 in the **Custom** dialogue box and press the **Set** button. Thus, the simulation will run 3,600 times (60*60) faster than real time, or 40 hours of simulated time should take about 40 seconds of real time.



- Now, press the **Reset** and **Run** buttons on the **Model Execution Toolbar**.

Once the run is complete, it should look similar to the following figure. Items (boxes) should move across the object named **Processor1** and accumulate in the object **Queue1**.



*Congratulations, you've built your first FlexSim model!
Hopefully, this will be the first of many helpful models for improving system performance.*

- When you select the **Queue1** object, its statistics are displayed in the **Properties** window, as shown in the figure above.

A few things to note in the figure above about the model's state at the end of the 40-hour run.

- According to the value of the **% Processing** statistic shown beneath the object, **Processor1** was busy 86.7% of the time over 40 hours.
- Over the 40 hours, 222 items entered the model through the **Source1** object, as per its **Output** statistic, and 208 items left through **Sink1**, as per that object's **Input** statistic.
- At the end of the run, the number of items awaiting processing is 13, as per the **CurContent** property for **Queue1**. Similarly, the maximum number of items waiting at any time since the simulation began is 14 (**MaxContent**). The average time an item spent waiting is 27.7 minutes (**AvgStaytime**).

This may or may not be an acceptable performance; it depends on the system. In a human-based system, waiting nearly a half hour for 10 minutes of service may not be acceptable. However, the wait time may be acceptable in a manufacturing or logistics system.

This model is running with all default parameters. While the basic flow logic is correct, the model does not use data that represents characteristics of the example system, such as inter-arrival times and process times. The next section defines the primer's example and begins the customization process.



To save the current model, use the Save option in the File menu. You can use whatever file naming convention you prefer, e.g., MyFirstModel. *FlexSim* models have the filename extension .fsm.



Use the **Save Model As** option in the **File** menu to make a copy of the existing model so that it can be customized beginning in the next section. Again, you can use any file name, but in the primer, the next model is referred to as Primer_1.

The following are a few tips for managing simulation models.

It is good practice to save a model frequently and as a new version, i.e., with a new name, whenever significant changes are to be made. This permits a rollback to a model that has been tested and works as desired if a mistake is made while making the change or if the change does not provide the desired result.

If a mistake is made in a model during the model-building process, the action can be undone by using the **Undo** command located under **Edit** in the **Main** menu.

FlexSim automatically saves a model in a separate file every 10 minutes. The default time can be changed on the **Environment** tab of the **Global Preferences**, located in the **File** menu. The backup file name is the current file name with _autosave appended, e.g., Primer_1_autosave.fsm.

PART II – MODELING THE FINISHING AREA

The modeling of the primer example begins by defining the system that the model will represent. It then describes the development of a simulation model of the first part of the production system, the Finishing Area.

- Chapter 6 defines the system that is modeled throughout the primer. It also describes the modeling approach that is used.
- Chapter 7 creates the initial model of the Finishing Area and introduces *FlexSim's* basic Fixed Resource objects.
- Chapter 8 describes basic object statistics and creates a dashboard with time series and histogram plots of measures of performance for the Finishing Area.
- Chapter 9 introduces mobile resources, referred to as Task Executors in *FlexSim*, and adds a Finishing Operator to the model.
- Chapter 10 examines the Processor's operation, which is one of the basic 3D objects. It is meant to help the reader better understand what is happening behind the scenes in the basic 3D objects. Thus, this chapter does not add features and capabilities to the primer model.

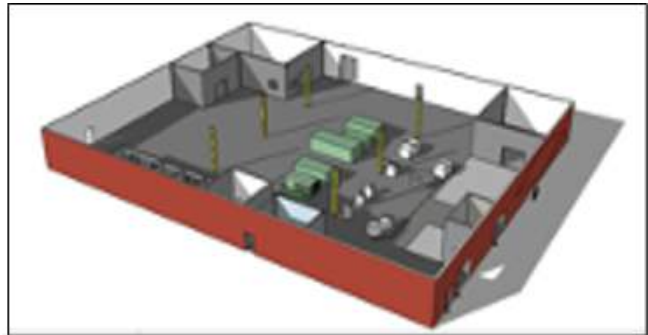
6 THE EXAMPLE MODEL FOR THIS PRIMER

Chapter 6 defines the system that is modeled throughout the primer. It also describes the modeling approach that is used.

This primer introduces *FlexSim*'s various features and capabilities using a single model and provides step-by-step instructions for building a simple yet comprehensive simulation model.

6.1 Defining the System and Problem

Dobry Products Limited (DPL) plans to reuse an area in its production facilities to finish, pack, and store “containers” for distribution. The existing facility is shown below. Any equipment will be moved out so that the space can be used to support the new production area.



The new production area will finish various types of containers and then pack them with components, the contents of which depend on the type of container. All process times in both finishing and packing depend on the container type. After packing, containers are placed in a warehouse to await being used to fill an order. The order fulfillment process is the final part of the model; this is where containers are pulled from the warehouse to fill individual order requests. Material handling within the facility is yet to be decided; simulation will help make those decisions. However, the initial thought is to use a combination of conveyors and AGVs (Automated Guided Vehicles) to move products between production areas.

At this stage in the design process, DPL is unsure how many types of containers they will produce, the demand for each type, and the components that will be packed into each container. However, they want to optimize operations and use simulation to help them design the facility and operations. Therefore, the simulation model will initially be built to reflect the general operations, yet it will contain numerous assumptions that will be modified as the project evolves. This approach is essential so that the project can be done in a timely manner - the simulation work must start well in advance of when all of the information is known. In fact, the simulation will be used to help make some of the design decisions and establish the system's characteristics and capabilities.

This is a very wise approach – *simulation is most effective when applied early in the design process* since things can easily be changed early in a project before too many resources are committed, too many costs are incurred, and too many decisions are made.

The modeling process begins by identifying and representing the essential components of production in the finishing and packing areas (equipment, material, operators, transporters, etc.) and defining the relationships among the components. Later, this is expanded or scaled up so that the model aligns with the company's production plans.

Modeling should be done incrementally – start small and simple and add complexity as needed. The best model is not the one that is the most complex – *the best model is the one that has the minimum amount of detail necessary to answer the questions being addressed*. Remember, no model can replicate the real system – the real system is too complex. Any model is a representation, albeit a simplification, of the system being considered.

6.2 Modeling approach

As with any simulation project, modeling and analyzing the system identified above is done in steps, moving from the simplest representation to the more complex. After each step, it is important to test and validate the model. The validated model is saved in a file with a different name so that if a subsequent modeling effort encounters a problem, modeling can “roll back” to the last saved and validated version.

One of the most difficult aspects of modeling is deciding how much detail should be incorporated into a model. This is often referred to as *model fidelity*. It is very tempting to model as much detail as possible, but in this case, more is not better. The more complex a model becomes, the harder it is to validate, verify, and maintain. Models should be as detailed as needed *to answer the posed questions*.

Of course, this is easier said than done; determining the proper fidelity comes with practice and experience. In any case, modeling should start simple and add complexity as needed. Start modeling with an extensive list of assumptions and a very simple model. Then, iteratively decide which assumptions need to be removed and remove them by adding features to the model that address the assumptions.

Be careful not to fall into the trap where a model incorporates features and detail because it can be done and not because it should be done.

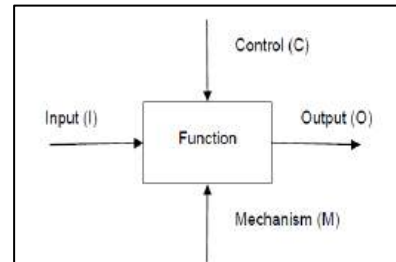
Another good modeling practice is to look for similarities within the system and model them as a single instance, i.e., create a base object. Test and validate the instance. Once validated, copy and paste the base object as many times as needed. For example, if a system has multiple packing stations, model, test, and validate one station, then duplicate as many as required in the model.

One way to do the above in *FlexSim* is to create a **template** object. With this approach, the base object becomes the parent, and any child will inherit its parent's properties. Each child's properties can be modified to make it unique or it can remain inherited. If an inherited property is changed in the parent, it is automatically changed in the child.

Another good modeling practice is to use small **study models** to develop and test concepts before implementation in the main model, i.e., develop and test in isolation, removed from the complexity of the main model. This applies to developing logic within a single object or a set of objects.

It is always good practice to *sketch* the system first. This helps to focus on how the system *operates*, not how the model will be *built*. Use sketches to discuss the system operations with domain experts. One diagrammatic methodology for representing dynamic operations systems is the Object Flow Diagram (OFD).

An OFD uses symbols to represent various functions in a simulation model, such as process, transport, convey, etc. The edge of each symbol has a specific meaning, as shown in the figure to the right. The Input, Output, Control, and Mechanism notation is derived from the IDEF0 methodology. A partial OFD for the example model that is used in this primer is provided in the next section.



[Section 17.3] Prepare a conceptual model.

The following section, Initial System Narrative, briefly describes the system that will be modeled throughout this primer. It provides a high-level description of the system's operation. As with many simulation projects, modeling can start with assumed logic and placeholder values until they can be estimated or determined.

Initial System Narrative

Unfinished containers arrive at the Finishing Area at a yet-to-be-determined rate. The number of types of containers and their properties are also not yet known. Each arriving container is loaded onto a Finishing Machine by a Finishing Operator. If no machine is available, the container waits in a buffer until it can be loaded. The size of the buffer and how the operator decides which container to process next must be studied. To maximize throughput, DPL is considering using the shortest-processing-time rule to load the Finishing Machines but is concerned some container types may wait too long for processing if it uses this approach.

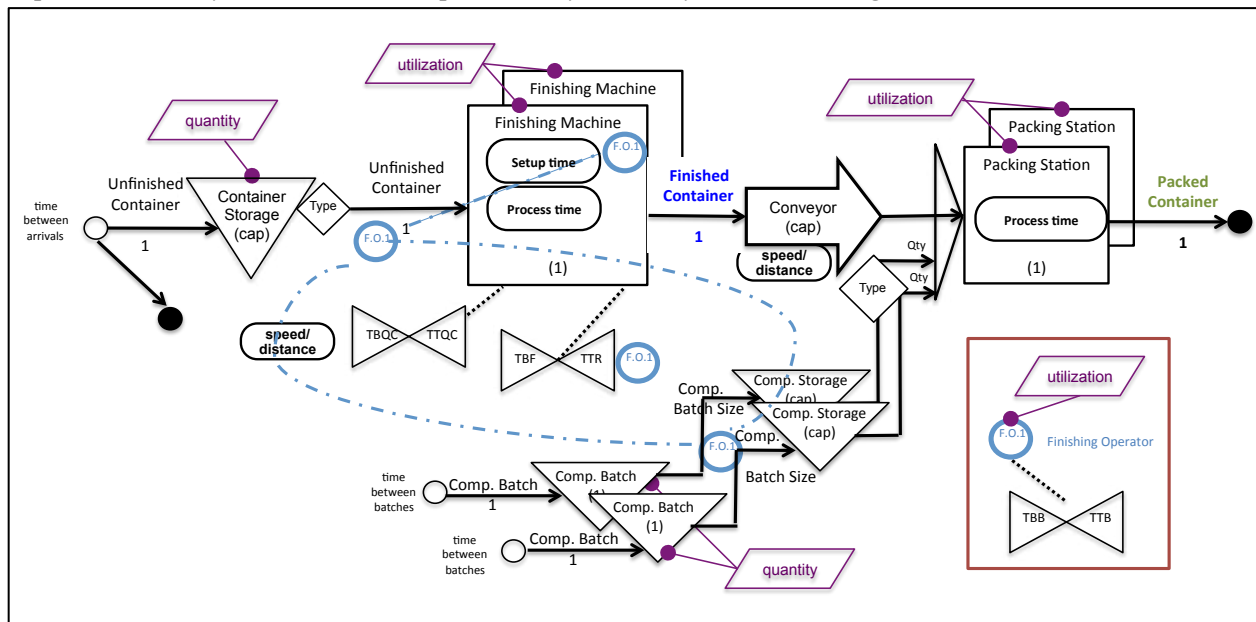
The finishing process transforms unfinished containers into finished containers. The time to finish a container depends on the type of container, but the types are still being determined. The finishing process is automated; i.e., it does not need an operator. However, it is expected that a setup operation is required on the Finishing Machine if the next container type to be processed differs from the previous one produced on the machine. The Finishing Operator is expected to set up the operation. The number of Finishing Machines needed to meet forecasted demand and product mixes is also yet to be determined – the simulation will help decide this. Each Finishing Machine needs to run a quality check periodically. It is also subject to breakdowns that require repair. The quality check activity does not require other resources; it primarily uploads data to a server. However, any repair that results from a breakdown is expected to be performed by a Finishing Operator.

After the finishing operation is complete, the finished containers move to the Packing Area via a conveyor. The Packing Area transforms an unpacked container into a packed container. At Packing, each container is loaded with a mix of components that depend on the container type. DPL is considering using a robot at each packing station to pack the containers. Components are delivered in batches to the Packing Area, and the Finishing

Operator unpacks the batches and makes them ready for packing. After Packing, packed containers move to the next process, Warehouse, where they wait in Racks until picked to fulfill customer orders.

Since there is uncertainty in the number of Finishing Machines, packing stations, Racks, Operators, etc., the primer will focus on developing the pieces of the system and their relationships; i.e., this model will be a small version of the final system. The model will include two Finishing Machines, one Finishing Operator, one Packing Station, one Rack for each product type, one Order-Fulfillment Station, and the means to connect these areas, i.e., conveyors and AGVs. Once this model is validated and verified, its pieces can be used to scale up the model and design the entire system.

A portion of the system narrative is represented symbolically in the following OFD.



Note that the key performance measures of interest – denoted by the “slanted rectangle” symbol are the contents of the storage buffers for the containers and components (denoted as quantity), utilization of the Finishing Machines, Finishing Operator(s), and Packing Station. Also, the item flowing through the model changes, or is transformed by the operations, from an unfinished container to a finished container to a packed container. Thus, the Finishing Operator moves material from the incoming buffer storage to the Finishing Machines, performs the setup and any repairs on the Finishing Machines, and unloads batches of components in the Packing Area. More information on constructing OFDs is available in Section 3 of Chapter 17 in the *Applied Simulation: Modeling and Analysis Using FlexSim* textbook.

Each of the basic modeling objects used in the simple model is explored, and in doing so, the objects are modified so that the simple model becomes a better representation of the real system considered in the primer.

7 BASIC FIXED RESOURCES, FLOWITEM BIN, AND INITIAL CUSTOMIZATION

Chapter 7 creates the initial model of the Finishing Area and introduces *FlexSim*'s basic Fixed Resource objects.

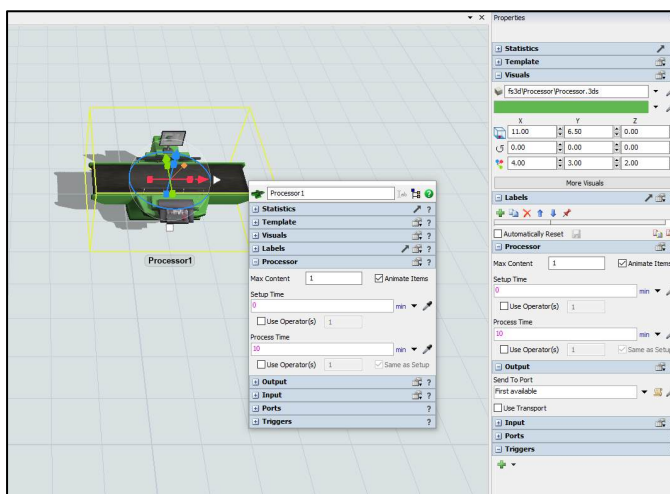
This section explores in more detail *FlexSim*'s basic fixed resources (**Source**, **Queue**, **Processor**, and **Sink**), which were used to develop the simple model. The objects will be customized to represent the system that is being modeled in this primer.

The base model for the additions described here is **MyFirstModel**, which was saved as **Primer_1** in Chapter 5.

7.1 Basic object properties and structure

Objects are customized by modifying their properties, which are accessed either by double-clicking on the object in the model view or through the **Properties** window when an object is selected. This was discussed earlier, but the figure is repeated to the right.

The number of object properties in *FlexSim* is extensive, thus providing a great amount of modeling flexibility and capability. Since this primer aims to introduce the basics of how *FlexSim* works and some of its capabilities, only the most basic and salient features and properties are presented. Therefore, it is beyond the scope of this primer to discuss every property on an object interface and every object and feature in *FlexSim*. With the foundation provided in this primer, the *User Manual* can be used to understand other properties and parameters.



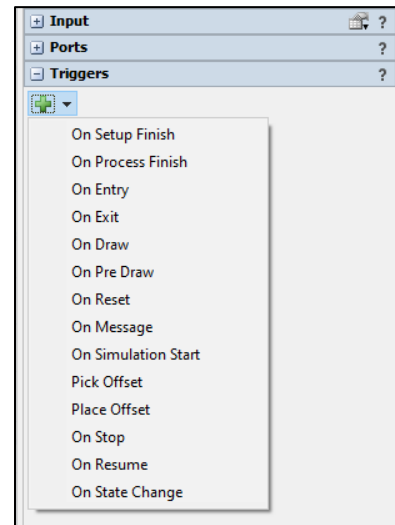
Once a property value is changed on an object, it is automatically saved.

As mentioned previously, the basic structure of *FlexSim* objects is the same, with properties grouped into panes, such as **Statistics**, **Visuals**, **Labels**, **Triggers**, etc. Having a common structure greatly enhances learning *FlexSim* and its overall ease of use. Each of the more common pane types is defined below. Many of these panes and their properties will be discussed in the primer.

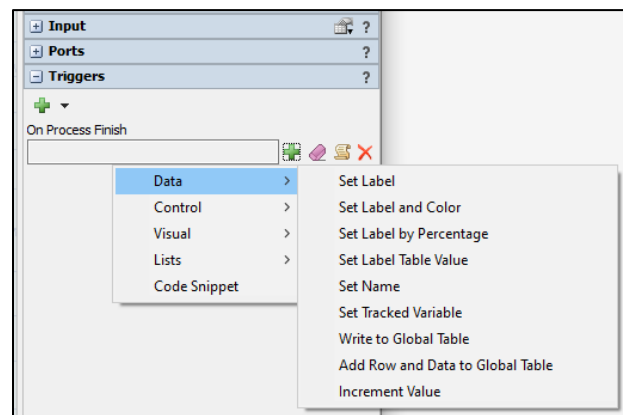
- **Statistics** pane contains basic performance measures for the object that are continually updated as a model runs.
- **Visuals** pane defines an object's size, 3D shape, color, position, etc., as well as flags for controlling what is displayed, the object's accessibility, etc.
- **Labels** pane defines user-created properties and their values, which are used or updated during a simulation. There are multiple types of labels, such as numeric, string, pointers, and arrays.

- **Triggers** pane invokes optional actions that can occur as a model runs. Triggers have many diverse uses and are grouped both by *when* the action occurs and by *what* type of action occurs. The type of trigger varies by object, but many triggers are common across all objects. When an action specified by a trigger is executed during a simulation run, it is said that the trigger “fires.” Trigger options are available through the  button on the pane interface.

- Triggers are first specified by *when* they occur, such as **On Reset**, **On Stop**, **On Entry**, and **On Exit** (when an item enters or exits an object, respectively), etc. The figure to the right shows when triggers can fire in a **Processor**. For example, the second option, **On Process Finish**, is used to add some action when the processing of an item is complete.



- Once the *when* condition of a trigger is defined, then the *what* action that is to occur is specified. The figure to the right shows the actions that are possible when an **On Process Finish** trigger fires. The actions are grouped by the type of actions, such as **Data** (e.g., set a label value), **Control** (e.g., open or close object ports), **Visual** (e.g., set color or size), **List** (e.g., push to or pull from lists), and **Code Snippet**.



In the case of the figure to the right, for an **On Process Finish** trigger on a **Processor**, a variety of **Data** actions are possible, such as *Set Label*, *Set Label and Color*, ..., *Increment Value*.

- All objects have triggers related to their type, e.g., **OnCreation** in a **Source**-specific object, **OnProcessFinish** is only available in a **Processor**, **OnResourceAvailable** is only found on task executer objects, etc.

All **Fixed Resources** have an **Output** property pane, except the **Sink**, i.e., no items flow out of that type of object. The **Output** property controls to which output port an item should be sent when it leaves the object. That decision may, for example, be based on the downstream object(s) availability or an item's attribute or label value. The **Output** property also specifies whether an item uses a transporter to reach the next object. If a transporter is used, it is requested based on specified criteria. The item waits in the current object until a transporter arrives and loads the item. Therefore, if an object's capacity is one item, then a new item is not moved into the object until the current object exits via a transporter (if required). Note that if no transport is

used, an item moves between two **Fixed Resource** objects in zero simulation time regardless of the distance between the two objects.

All **Fixed Resources** have an **Input** property pane, except the **Source**, i.e., no items flow into that type of object. The **Input** property enables “pulling” items into the object. Normally, items move through a *FlexSim* model by being “pushed” from one object to another; i.e., the current object determines to which subsequent object an item moves. Pulling enables an object to determine which item it processes next based on specified criteria.

Each object has a unique name. *FlexSim* must ensure each object has a unique name so it automatically assigns a name when an object is created by selecting it from the library and dropping it into the 3D modeling view. When a model has multiple instances of a type of object, even with a relatively small number of objects, it becomes difficult to recall which object does what in a model, e.g., the difference between *Source1* and *Source2*. Therefore, each object should have a meaningful name that reflects its role in the model; e.g., *ContainersArrive* is a much more meaningful name than *Source1*. The naming is for the modeler’s benefit and that of others who access the model, not for the software. The object names used in the primer are the author’s choice. While you do not need to use the same names, it is helpful to do so to make the primer easier to follow.

Each of the following sections defines one of the most basic objects and describes how to customize it to represent the primer’s modeling example.

7.2 Source object

The **Source** object creates a modeling boundary, a starting point for what is being considered in the system. For this example, what happens to the containers before they arrive at the Finishing Area is not a concern at this time.

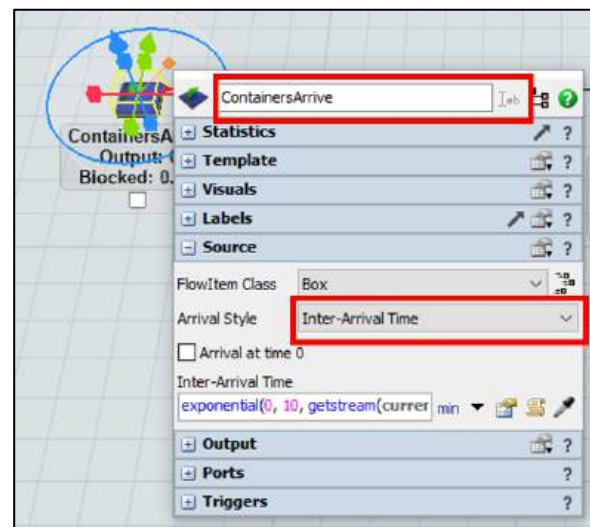
As stated earlier, object names should be meaningful.

- Name the **Source** object *ContainersArrive* in the text box at the top of the object’s window and as shown in the red box in the figure to the right.

Information about the item being created is specified on the **Source** pane. Items are created in one of three ways or modes based on the **Arrival Style** property:

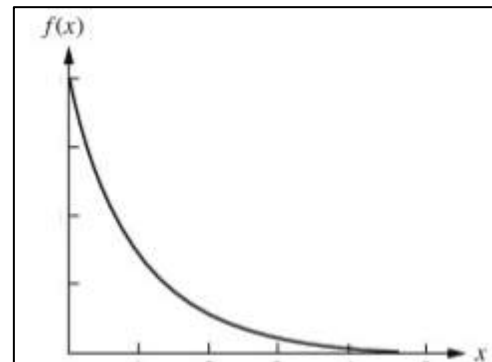
- **Inter-Arrival Time** uses an inter-arrival time function that determines the length of time until the next item is created and arrives in the model. Once an item exits the **Source**, the function generates the time until the next arrival, and the cycle repeats. Note that if an item cannot exit the **Source**, e.g., it is blocked by a downstream object, then the item will wait in the **Source** until it can be released. Once it is released, the time until the next arrival is determined. The first item may arrive at time 0 or when determined by the inter-arrival time function.

The default value for the **Arrival Style** property is **Inter-Arrival Time**, which is used in this example. The property is shown in the red box in the figure above.



- *Arrival Schedule* uses a user-defined schedule table for when items should arrive. The arrival time, name, number of items to create, and item labels are specified for each table entry. The schedule may be repeated. This option will be discussed later in the primer.
- *Arrival Sequence* is like the arrival schedule mode, except that there is no time associated with the arrivals. The source will create the flow item for a given table row, and then, as soon as the last flow item for that entry has exited, it will immediately go to the next row in the table. You can also repeat the arrival sequence.

The time between arrivals is defined by the **Inter-Arrival Time** property. The default value is `exponential(0, 10, getstream(current))`, i.e., each inter-arrival time is obtained by a random sample from an exponential probability distribution with a mean of 10 time units, 10 minutes in this example. The general shape of the probability density function for the exponential distribution is shown in the figure to the right. Technically, the exponential distribution is the negative exponential but is referred to as the exponential for brevity. The parameters of the exponential distribution are location (0), scale (10), and stream number (`getstream(current)`). Location is the lower bound of the distribution; scale is the mean of the distribution and sets the steepness of the descent of the curve; and `getstream(current)` is a *FlexScript* command that provides a stream number for *FlexSim*'s random number generator. The methodology for generating random numbers is not covered in this primer.



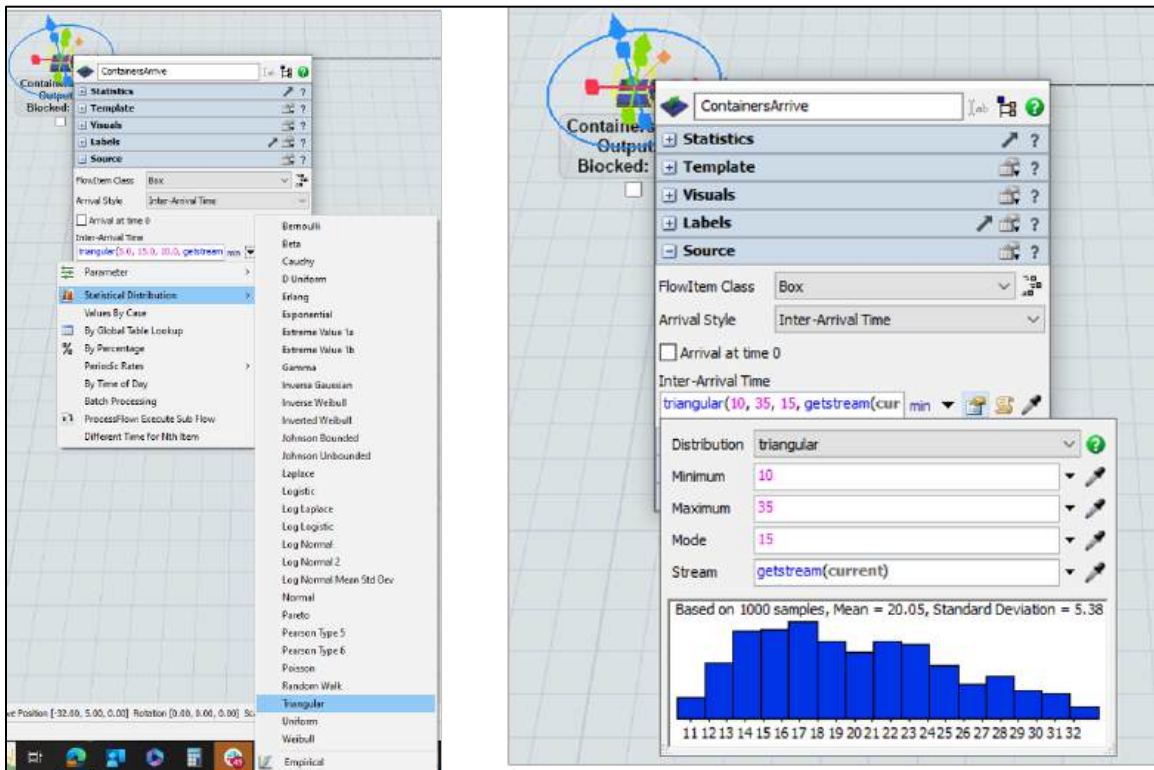
DPL is unsure of the arrival pattern for the upstream process that will produce the containers that must be finished in the new facility. A group of engineers and domain experts discussed this and decided an average rate of three containers per hour should support two finishing machines.

The average rate of three containers per hour is the same as saying the average time between arrivals is 20 minutes. Simulation typically uses times between arrivals and not rates when specifying the arrival process. The average time between arrivals is 20 minutes, but it is not a constant. Since there is variability, what probability distribution should be used to represent the arrival process?

For now, the engineers have decided to assume a triangular distribution. To specify this distribution, only the minimum time between arrivals, maximum, and most likely values are needed. Therefore, after some discussion, the engineers and domain experts decided the minimum time between arrivals is 10 minutes, the maximum is 35 minutes, and the most likely is 15 minutes.

For this example, and as shown in the screenshot on the left in the figure below, change the inter-arrival time distribution.

- For the **Inter-Arrival Time** property, use the dropdown menu and select the *Statistical Distribution* option, followed by the *Triangular* option in the submenu.



FlexSim offers many distributions, each with a different set of parameters. The triangular requires three parameters: the minimum, maximum, and most-likely (mode) times between arrivals.

- As shown in the screenshot to the right in the figure above, enter the triangular distribution's parameters for this example. The **Minimum**, **Maximum**, and **Mode** (most likely) values are 10, 35, 15, respectively.

To implement a distribution, the user interface provides the specified values to a *FlexSim* command so that it can be executed. In this case, it translates the interface values to `triangular(10, 35, 15, getstream(current))`

These specified parameters result in an average time between arrivals of 20 minutes, the sum of the three parameters divided by three. In general, the mean of a triangular distribution is the simple average of its three parameters.

Again, an average time between arrivals of 20 minutes corresponds to an average arrival rate of 3 customers/containers per hour. Rates are the reciprocal of time values; in this case, the average arrival rate is $60 \text{ minutes/hour} \div 20 \text{ minutes/customer} = 3 \text{ customers/hour}$. Simulation software typically uses average times rather than average rates.

As noted earlier, the last parameter in specifying any probability distribution in *FlexSim* is the number of the random number stream to be used for this source of variability. This is implemented through the command `getstream(current)`. Again, the discussion of random number streams is beyond the scope of the primer. However, it is fine to use the default for the **Stream** property. It is good general practice that stream values be

unique for each source of variability in a model – this is handled automatically by *FlexSim* through the `getstream(current)` command. *FlexSim* commands will be discussed later in the primer.

As indicated above, a detailed discussion of how probability distributions for interarrival times, process times, failure times, etc., are selected is beyond the scope of this primer. However, *FlexSim* provides a tool to help with the process; it is a curve fitter that fits sample data to common probability distributions. This feature will be discussed later in the primer.

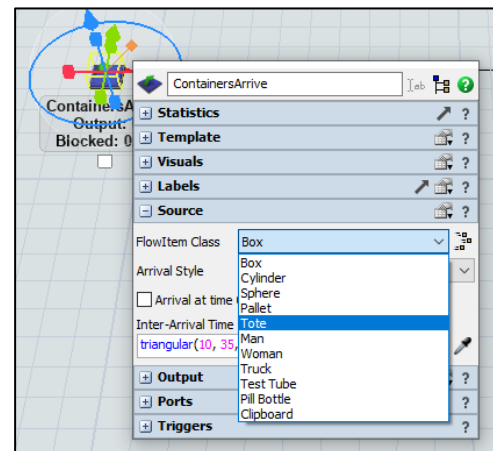
Also, it is generally a good idea to use bounded distributions so that the model does not use unrealistic, extreme values that can result from such distributions as the normal and exponential.

More information on selecting probability distributions to use in a simulation model is available in Chapter 8 of the *Applied Simulation Modeling and Analysis Using FlexSim* textbook. However, the decision on what probability distribution to use is briefly discussed later in the primer, in the sections on the empirical distribution and curve-fitting tools.

The **FlowItem Class** property specifies the type of item created by a **Source**. The default type is *Box*, but other options are available in the drop-down menu, as shown in the figure to the right. Custom items are easy to create in the **FlowItem Bin**, which is discussed later.

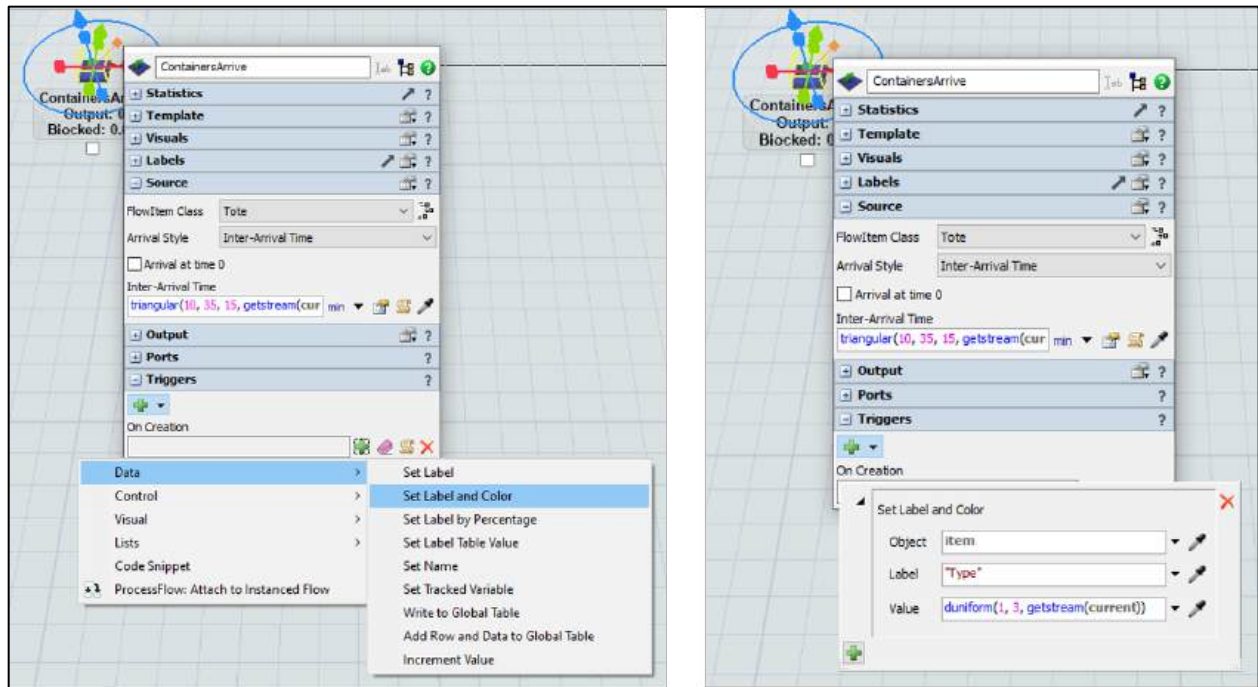
- As shown in the figure to the right, for this example, select the *Tote* as the **FlowItem Class** value.

The figure to the right shows what the basic **Tote** object looks like.



In this example, DPL produces multiple types of containers, all of the same class; i.e., they can all be represented by the same container item, *Tote*. For now, assume three equally likely types of containers are produced; this will be changed later in the primer. Each container will have a custom property or **Label** named *Type* with values of 1, 2, or 3. To easily distinguish the types in the model, each container object will be colored according to its *Type* value.

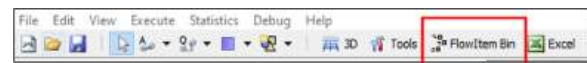
- To model this in *FlexSim*, use the **Source**'s *OnCreation* trigger, and select *Data* and then select the *Set Label and Color* option, as shown in the left screenshot in the figure below.
- Use the default properties for this option as shown in the screenshot on the right side of the figure below.



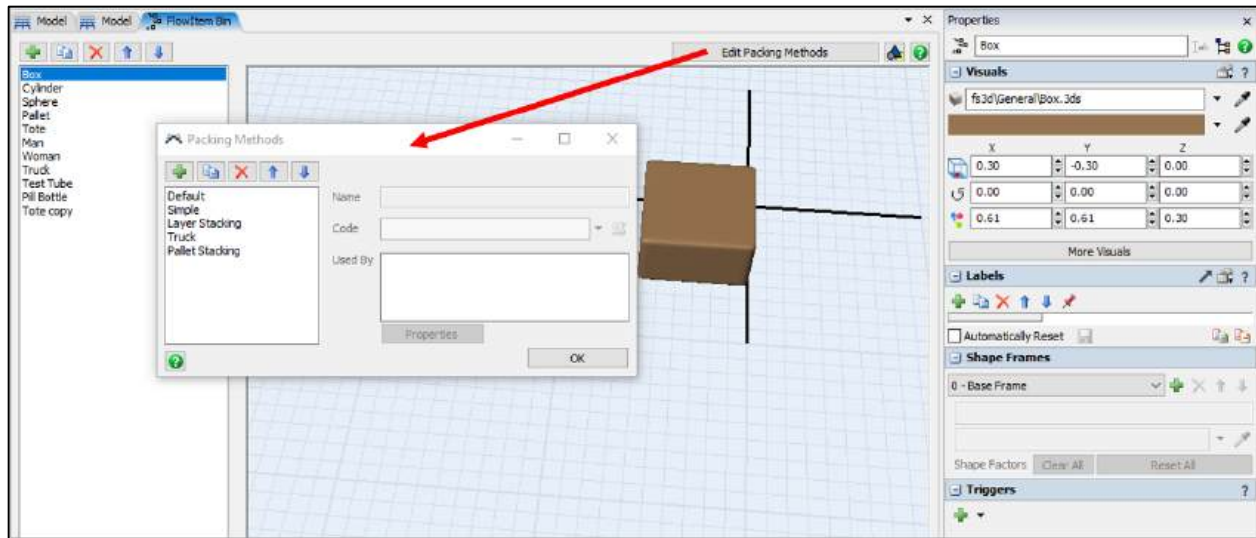
Thus, the **OnCreation** trigger randomly assigns a value to the label **Type** and colors the item accordingly when it is created at the **Source**. The value of **Type** results from a random sample from a discrete-uniform distribution with possible values of 1, 2, or 3, with each value equally likely. This action is accomplished by the *FlexSim* command `duniform(1, 3, getstream(current))`. The item's color is based on its **Type** value and uses *FlexSim*'s default color scheme, where 1 is red, 2 is green, 3 is blue, 4 is yellow, etc. Of course, a custom color scheme can be defined.

7.3 FlowItem Bin

Items are customized through the **FlowItem Bin**, which is accessed, as shown in the figure to the right, via its button on the **Main Toolbar**.



The figure below shows an example of the **FlowItem Bin**. The left-hand portion of the interface shows the default classes of items (*Box*, *Cylinder*, *Sphere*, ...). The toolbar above the list of items (in the red box) is used to add, copy, delete, or reorder item classes. The insert in the figure, obtained by pressing the “Edit Packing Methods” button, shows the packing methods available for the container items; these methods are defined below. The right-hand panel, the **Properties** window, shows the properties of the selected item, where size, shape, color, etc. are defined.



There are two general categories of flow items – Basic and Container.

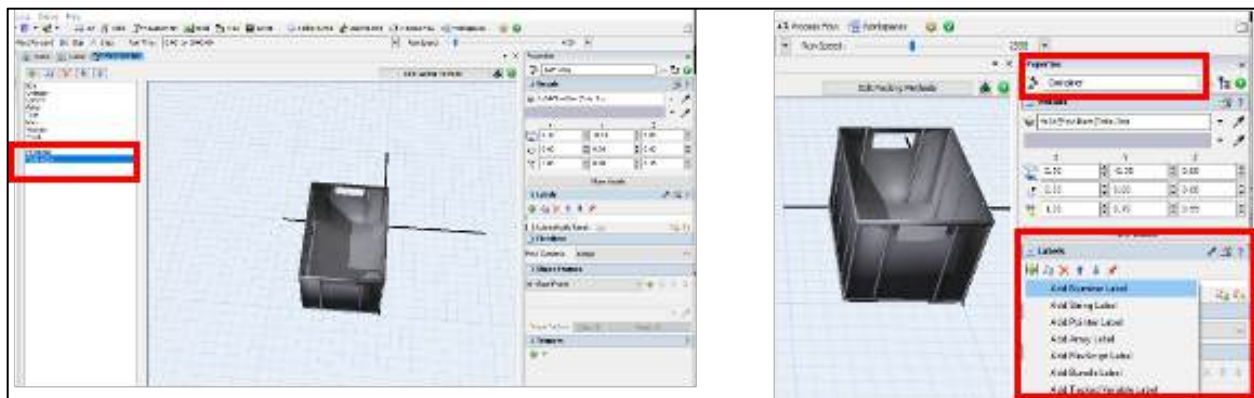
- Basic flow items are simple shapes, such as *Box*, *Cylinder*, *Sphere*, *Man*, *Women*, *Test Tube*, and *Pill Bottle*. Their visual appearances—such as size, shape, rotation, and color—can be customized in the **Properties** window of the **FlowItem Bin**.
- Container flow items – *Pallet*, *Tote*, and *Truck* – are special in that they can be packed in various ways and thus contain basic items. There are five packing methods, as shown in the popup window **Packing Methods** in the figure above. The window is opened by pressing the **Edit Packing Methods** button. The packing methods are briefly described below and may be selected and modified through the popup window. Modifying the methods is beyond the scope of this primer.
 - The *Default* packing method stacks basic items on top of the container item, filling up as much available space as possible before moving to the next level. This is similar to how items are stacked in a **Queue** by default.
 - The *Pallet Stacking* method is used by the *Pallet* container item. It is nearly identical to the *Default* method except the z-size of the pallet adjusts to accommodate items placed in a new layer.
 - The *Simple* packing method is used by the *Tote*. This method is similar to the *Default* method, except items are packed into the container instead of onto it.
 - The *Layer Stacking* method considers the uniformity of item sizes. When items have a consistent height (z direction), they are put on one layer of the container item. As soon as the height of a flow item differs from the others, it is placed on a new layer. The width (x direction) or depth (y direction) can be checked instead of the z.
 - The *Truck* packing method is used by the *Truck* class of items. Within the *Truck* item, other items are first stacked on top of each other at the forward end of the truck, and the stacking moves toward the back.
 - Each packing method can be customized using *FlexScript*, but this topic is more advanced.

It is **good practice** to create a copy of the basic class used in a model and rename it. Since the class will usually be edited or customized, the copied class is the customized one, not the basic class.

- Select the *Tote* class, then press the button to the right of the  button on the toolbar above the list of item classes, i.e., the button highlighted by the red box in the figure to the right. This creates a new class named *Tote Copy* that is a duplicate of the *Tote* class.



- Select the created *Tote Copy* class, as shown in the red box in the figure to the left below.
- In the **Properties** window, change the item class's name to *Container*, as highlighted by the red box at the top of the figure to the right below.
- Also, in the **Properties** window in the figure to the right below, add a numeric label, i.e., a user-defined property. In the **Labels** pane, as shown in the lower red box, add the label by pressing the  button and selecting *Add Number Label*.

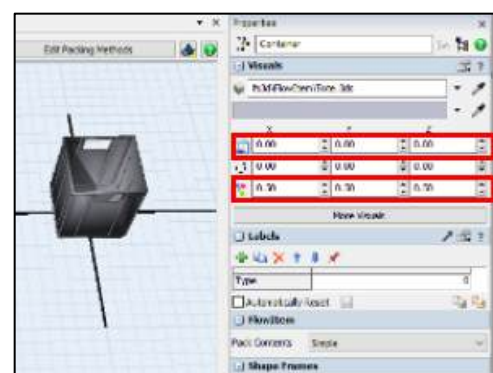


- As shown in the figure to the right, change the label's name from *label1* to *Type* and leave the default value of 0. This value will be changed to a value of 1, 2, or 3 when an item is created in the model through the **Set Label and Color** trigger described earlier in the **Source** section.



- Finally, as shown in the figure to the right, resize the container. As highlighted by the bottom red box in the figure to the right, set the *x*, *y*, and *z* dimensions to 0.5, 0.5, and 0.5; thus, the container is a cube.

As highlighted by the top red box in the figure to the right, recenter the object by setting the *x*, *y*, and *z* locations to 0.0, 0.0, and 0.0.



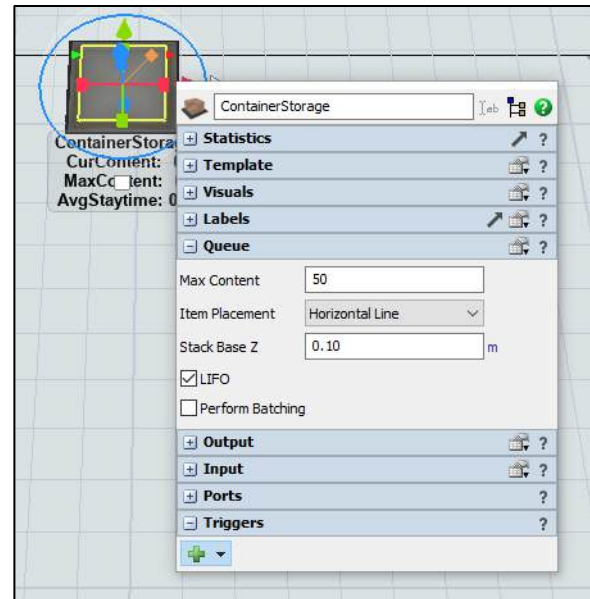
- In the **Source** object's **Source** pane, change the value of the **FlowItem Class** to *Container* using the property's dropdown menu. The **Source** now creates items from the *Container* class, not the *Tote* class.

7.4 Queue object

The **Queue** object delays an item if no downstream object is available to process an item. In this case, if a container arrives in the system and the **Processor** is busy, it waits in the **Queue**.

For this example, the **Queue** properties are changed as follows and as shown in the figure to the right.

- Name the **Queue** object *ContainerStorage*.
- Change the **Maximum Content** property from the default value of 1000 to 50. By default, the queue size is assumed to be large (1000), but in this example, it is assumed that the maximum realistic quantity is 50. This will likely change as the system is designed and the space available for storage becomes clearer.
- Change the **Item Placement** property from the default value of *Stack inside Queue* to *Horizontal Line*. By default, items are stacked in the queue by filling up as much available space as possible before moving to the next layer. The quantity in a layer depends on the physical size of the **Queue** and of the items.



7.5 Processor object

The function of a **Processor** is to invoke a planned delay on items flowing through a model.

- Name the object *FinishMach_1*; it is anticipated that more than one finishing machine will be needed.

Each finishing machine processes one container at a time, which is the default value for the **Maximum Content** property. However, note that a Processor can process multiple flowitems concurrently by changing this property's value.

As an aside, using the **Maximum Content** property is a quick and convenient way to represent multiple resources by using only a single object. This is handy when building a simulation model to obtain a rough estimate of system requirements. However, modeling multiple resources through a single object precludes modeling certain system aspects, such as downtime on individual resources, operator travel to specific resources, processing specific types of items on specific resources, etc.

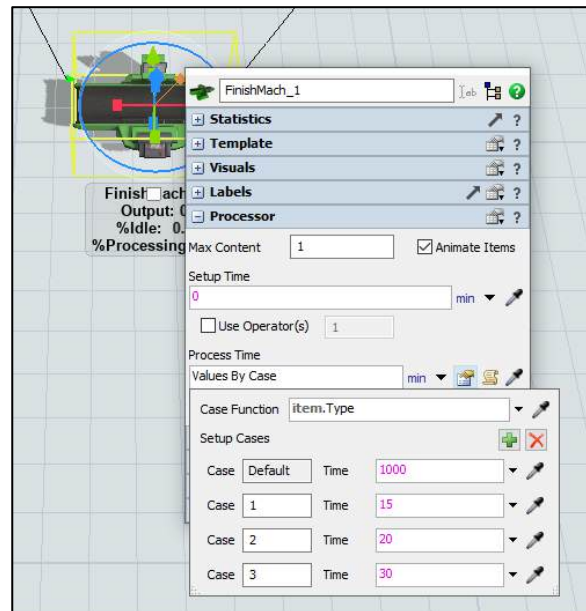
The planned delay time is specified by the **Process Time** property. The default is a constant 10 time units. In this example, the finishing time depends on the type of container; i.e., the finishing time for Type 1 is 15.0, Type 2 is 20.0, and Type 3 is 30.0. This is implemented through the **Values By Case** option. In this example, each case corresponds to a type; therefore, three cases are created.

The notion of “Case” comes from computer programming, where a Case construct is used to select a value from a set of alternatives based on prescribed conditions. It is like a series of if...then statements but is easier to read and write.

To model the situation where process time depends on product type, perform the following steps.

- Select the **Process Time** property, then select the *Values By Case* option from the drop-down menu.
- Create 3 cases, in addition to the default, by using the  button; then set the values as shown in the figure to the right.

The logic works as follows. When an item enters this **Processor** and needs a processing time, *FlexSim* checks the value of the item’s label **Type** and it becomes the case number. Therefore, if an item’s **Type** value is 2, then the time specified for Case 2, 20.0, is used for the value of the item’s process time.



The **Default** Case value is used if an item has a **Type** value other than 1, 2, or 3.

The **Default** process time in the *Values By Case* logic is set to 1000 or any large number. This helps the modeler remember to add a **Case** if a new type of container is added to the model. This way, if a fourth type is added in this example, and this property is not updated with a new **Case**, then the item will be delayed a long time at the **Processor** and things will back up in the model. This should prompt an investigation into the problem, and hopefully, the failure to update the process time is discovered and remedied. If a small value is used for the default, this oversight may not be caught, and the new container type will use an erroneously low value, thus affecting the results.

The notation in the **Case Function** property is the way *FlexSim* refers to a label. In general, it is referred to using dot notation because a dot or “.” separates parts of the statement, i.e., **object.LabelName**. In the case of the property value *item.Type*, the object is the flow item, referred to as *item*, and the name of the label being considered is *Type*.

Since the three types of containers are equally likely, the average processing time is 21.67 minutes (average of 15.0, 20.0, and 30.0). Recall the average time between arrivals from the source is 20 minutes. This means the single finishing machine will be overloaded since the planned utilization is greater than 100%, on average ($21.67/20.0 = 1.083$ or 108.3%). The situation will actually be worse since there is variability in the system due to the type of container, time between arrivals, and the finishing machine not being available 100% of the time due to downtime. One way to remedy the large queue contents is to use another finishing machine, which is done later in the primer.

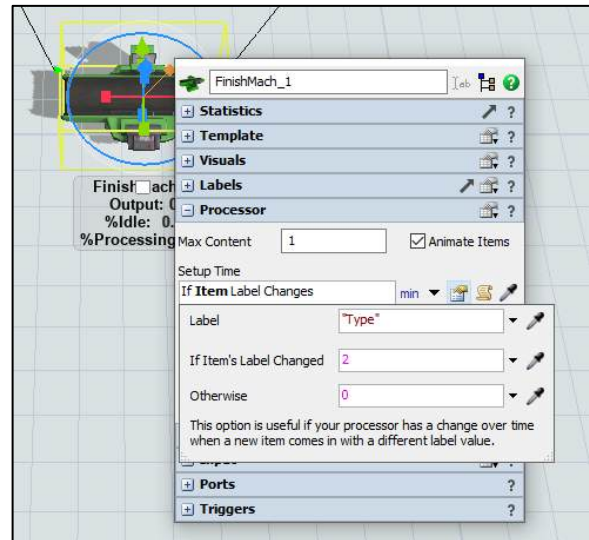
In addition to the processing time, there is a setup operation. The setup operation is separated from the process time for two reasons:

- (1) An operator will perform the setup operation, but the machine will run on its own once it is set up. This will be incorporated into the model later in the primer.
- (2) The setup operation only needs to be performed when the type of item (container) changes.

The time to perform the setup operation, when needed, is assumed to be a constant of two minutes.

The setup logic is implemented into the *FlexSim* model as follows and as shown in the figure to the right:

- Select the **Setup Time** property on the **Processor** object's **Process** pane, then select the drop-down menu option *If Item Label Changes*.
- Update the resulting menu's property values as shown in the figure to the right and as defined below.
 - **Label:** no change; use the default value "Type."
 - **If Item's Label Changed:** Set this property's value to 2. This represents 2 minutes of setup time.
 - For the property **Otherwise**, change the default value 5 to 0; i.e., if the item to be processed is the same type as the one just processed, then there is no setup time.



7.6 Sink object

The **Sink** object removes items from a model; therefore, it is the opposite of a **Source**. However, this functionality is much less complex than that of a **Source**. Removing items from a model is not as involved as creating them, which involves defining what to create, with what properties, when to create them, etc. The **Sink** is another modeling boundary, the stopping point for what is being considered in the system.

- The only change made to the **Sink** is to change its name to *NextProcess*.



If you haven't already done so, save the model. Recall that it is good practice to save often.

8 BASIC MODEL OUTPUT

Chapter 8 describes basic object statistics and creates a dashboard with time series and histogram plots of measures of performance for the Finishing Area.

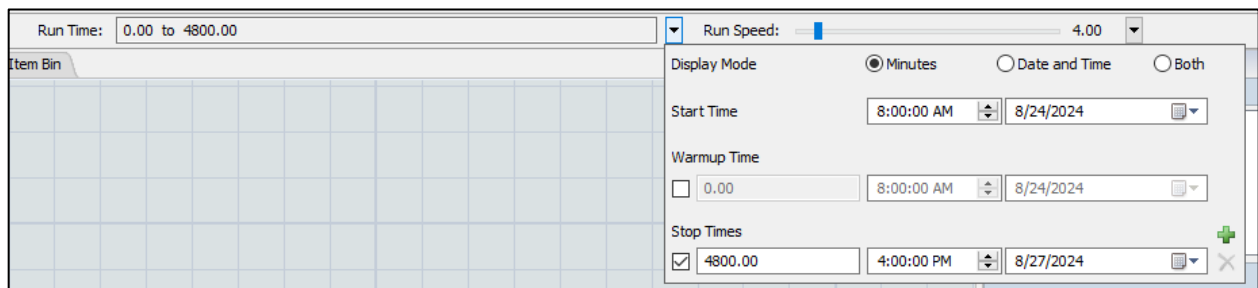
Simulation models are developed to assess the dynamic behavior of an operations system. Therefore, multiple means exist to access the results or output from a simulation model; a few simple ones are described here. Output is usually obtained from *FlexSim*'s **Experimenter**. It, among other things, is used to:

- Execute multiple experiments in a single run.
- Control experimental conditions, such as the number of replications to run and defining a warm-up period if needed. These are advanced topics discussed later in the primer.
- Obtain statistical estimates of system performance measures.
- Directly link to OptTek's *OptQuest* optimization software.

However, only a few basic output measures are discussed here for now.

The base model for the additions described here is *Primer_1* that was saved at the end of Chapter 7.

Consider an 80-hour run of the simulation model (equivalent to ten 8-hour shifts). In model terms, this is 4,800 minutes; recall, the model's time units were defined as minutes when the model was initially created. The figure below shows the model's run time settings, followed by the instructions.

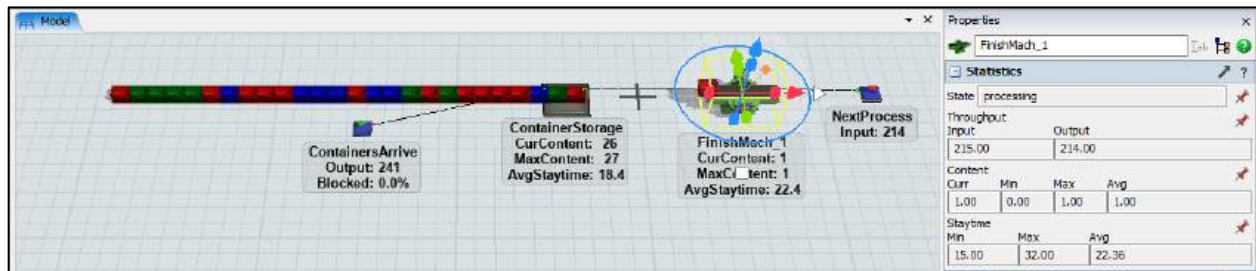


- Select the dropdown menu on the **Run Time** property that is located on the model's **Execution Toolbar**.
- Check the **Stop Time** box and enter *4800* into the text box to the right of the check box.

So that the model finishes more quickly, the **Run Speed** property can be adjusted higher than the default value of *4* using the slider interface.

8.1 Object Statistics and Properties

As shown in the figure below, a few basic output measures are provided for each object on the modeling surface. These values are continually updated as a model runs. The statistics of the selected processor are also shown in the **Statistics** pane on the **Properties** window.



At the end of the 4800-minute simulation when this screenshot was captured, the **Processor**, named *FinishMachine_1*, was processing an item (*CurContent* = 1), the most it processed at one time was 1 (*MaxContent*), and the average setup and process time is 22.4 minutes (*AvgStaytime*).

Additional statistics are shown in the **Statistics** pane on the **Properties** window, such as the object's current *State*, *processing*, total items *Input* and *Output*, 215 and 214, respectively, average contents of the object *Content-Avg* 1.00, and *Staytime-Min* and *Staytime-Max*, 15.0 and 32.0, respectively.

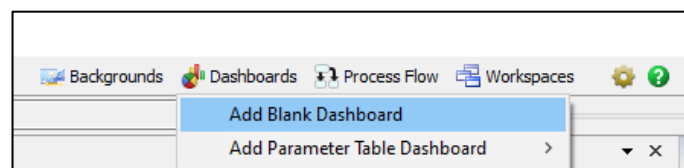
The pushpin icon is used to “pin” a statistic for that object onto a dashboard, as described in the next section.

8.2 Dashboards

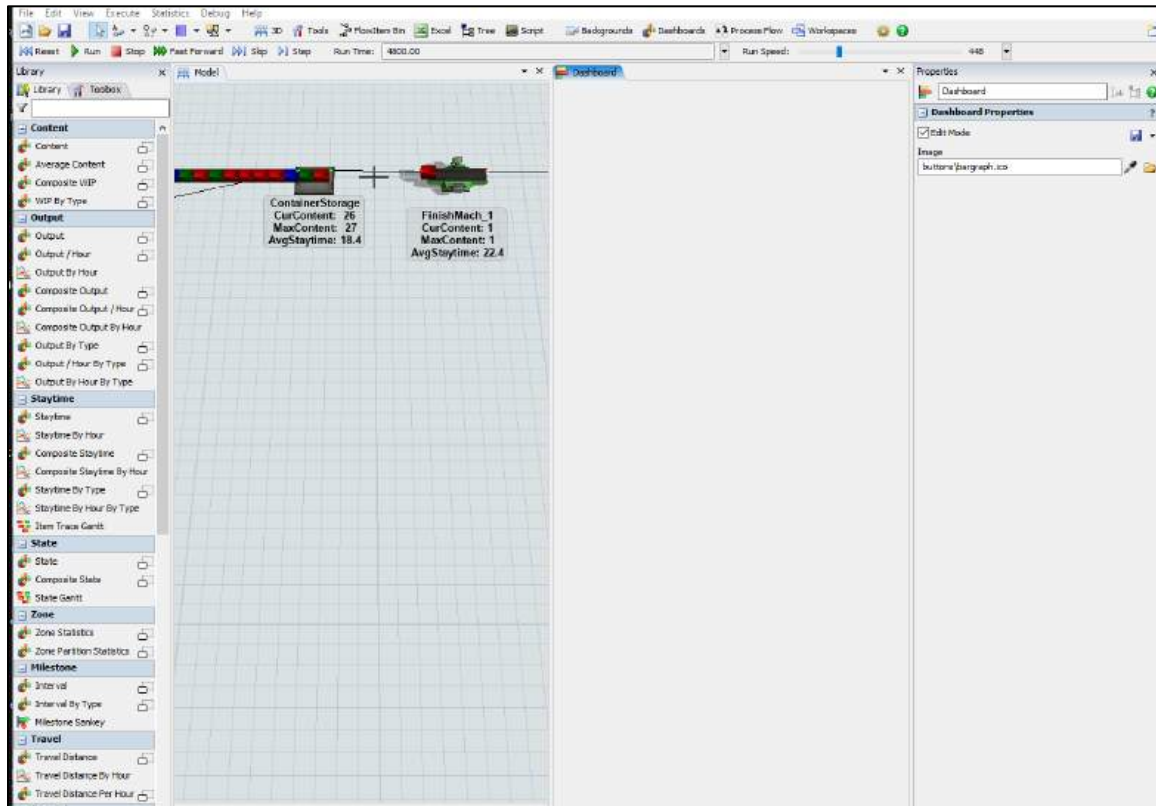
Charts and graphs are very helpful for assessing a system's dynamics compared to considering a single value at a snapshot in time, like the on-object and Properties values described in the previous section. *FlexSim* provides a wide variety of charts and graphs through the **Dashboard** tool. A Dashboard is either accessed from the **Main Toolbar** or from the **Toolbox** (Tool Library). A model may have multiple dashboards that capture various aspects of a system's behavior.

Charts and graphs are created on a Dashboard. Three example charts are described here: two types of time-series plots, where the value of a measure is plotted over time, and a histogram, which displays the frequency of occurrence of a value or range of values.

- Click the **Dashboards** button on the **Main Menu**, as shown to the right, and select *Add Blank Dashboard*. You can also add Dashboards through the  button in the **Toolbox**.



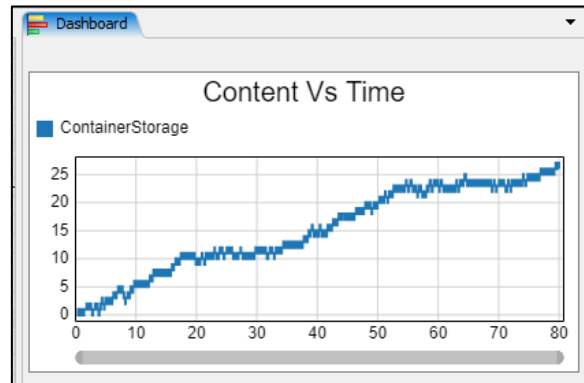
As shown in the figure below, a “blank” dashboard, or dashboard workspace, is automatically created just to the right of center on the screen. Also, a library of chart tools and templates is shown in the **Library** on the left side of the modeling environment. Note the library of 3D objects is automatically replaced by a library of chart types when a Dashboard workspace is selected. Thus, if the **3D** or **Model** view is selected, the **Object Library** is displayed, and if a **Dashboard** workspace is selected, the chart library is shown. The space to the right of the workspace, in the **Properties** window, is where the charts and graphs are defined.



The library of chart options on the left in the figure above offers many templates for commonly used charts. Of course, as is always the case in *FlexSim*, custom charts can be created. However, for now, the primer uses the predefined ones.

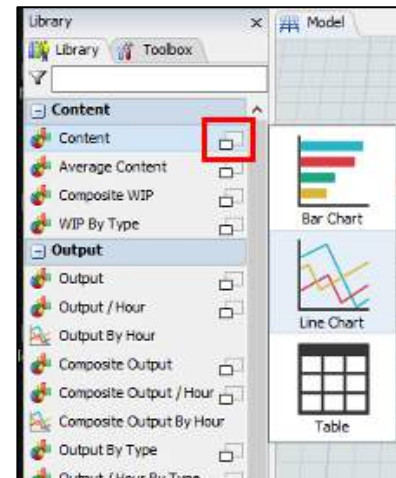
8.2.1 Time-series plot

The first type of chart considered in this example, as shown to the right, is a system-level plot of the total content of the **Queue** object named *ContainerStorage* over the simulation duration (4,800 minutes). In this case, the chart clearly depicts the dynamics of the system. It also indicates that, as expected, the queue continually grows throughout the simulation. Recall this is because the average process time is longer than the average time between arrivals. Conversely, the average service rate is less than the average arrival rate.

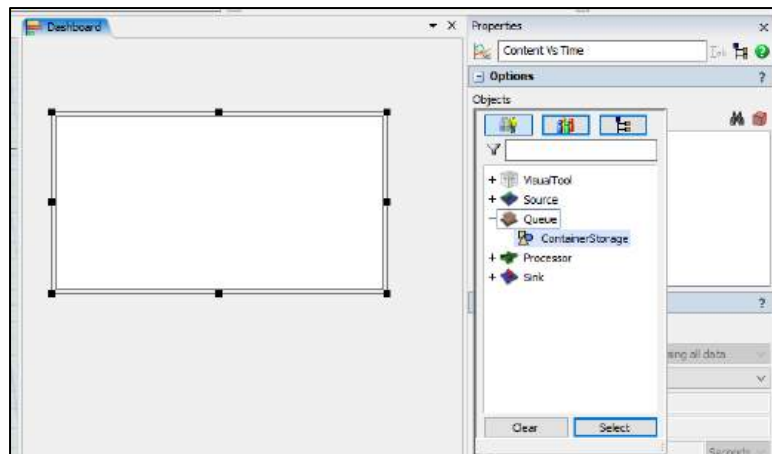


To create the time-series chart:

- As shown in the figure to the right, select the *Content* chart type in the **Library's Content** pane; then, select *Line Chart* from the pop-up menu. The icon in the red box indicates there is a choice of chart types available; in this case, *Bar Chart*, *Line Chart*, and *Table*. Selecting the chart type depends on what information is to be displayed.



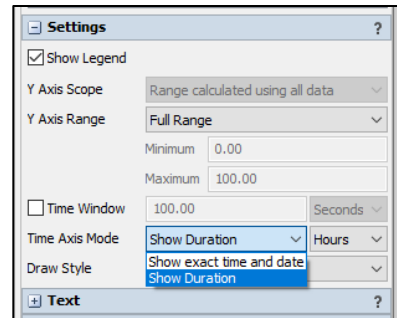
- Drag the Library item onto the **Dashboard** workspace, where it can be sized and positioned. This is a drag-and-drop operation, just like dragging objects onto the 3D view. In the figure to the right, the chart outline is shown in the workspace. The small black squares on the perimeter of the chart are resizing "handles."



- In the **Properties** window, change the name and title of the chart from *Content Vs Time* to something more meaningful, for the example, *Contents of Container Storage*.

The only required input in this case is to identify which object(s) to plot. Plots for multiple objects can be overlaid on the same chart. In this simple example, only one object is considered.

- As shown in the figure above, in the **Options** pane, click the  button in the **Objects** section, then choose *Select Objects*. This generates a list of all of the objects that are currently in the model by category (Sources, Queues, Processors, etc.).
- From the **Queues** category, select the object *ContainerStorage*, which is currently the only queue in the model. Notice that the object is highlighted. Now press the *Select* button. The **ContainerStorage** object is now displayed in the **Objects** section.
- While not required, it is suggested that the value of the **Time Axis Mode** property on the **Settings** pane be changed from the default *Show exact time and date* to *Show Duration*.
- Also, change the displayed units from *Seconds* to *Hours*. This changes the x-axis from calendar date and time to just the time, in hours, since the start of the simulation.
- Since only one value is being plotted and the chart title is descriptive, uncheck the **Show Legend** box (also on the **Settings** pane).
- The default settings for all of the other chart properties are fine for now.
- Relocate the chart on the Dashboard.
 - As noted above, when a chart is selected, it contains a double-lined frame around it with “handles.” Handles are small squares at the corners and midpoints of the sides of the charts. Also, notice that when the cursor is over the perimeter frame, it changes from the standard arrowed pointer to two crossed double arrows.
 - Left-click with the mouse anywhere on the perimeter frame of the chart to move it around and place it anywhere on the **Dashboard** window.
 - Similarly, left-click one of the handles and drag it to resize the chart. Notice that the cursor changes to arrows depicting the direction being resized—horizontal, vertical, or both (diagonal).



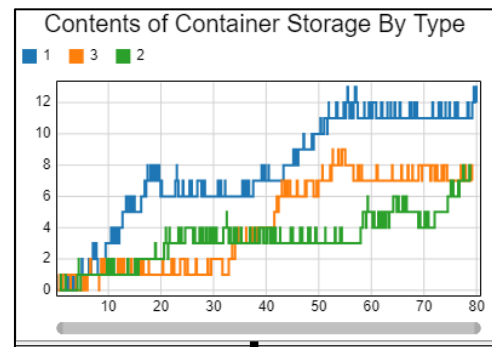
The chart type “Content” that was introduced above indicates the importance of considering how long to run a simulation. Had the model only been run for 480 minutes (8 hours), the maximum contents would have only been about five items, compared to nearly 30 when run for 4800 minutes (80 hours). Different decisions could have resulted based on the different run lengths. Discussion of this topic - how long to run a simulation - while very important, is beyond the scope of this primer.

The “Content” chart above also illustrates the dynamics of the simulation. Due to variability and interactions, the system could behave quite differently in another 4,800-minute time period, just as in the real system, behavior varies from day to day or week to week. This illustrates the importance of running a simulation model multiple times, referred to as **replications**, and combining the results, e.g., through averaging. Discussion of deciding how many replications to run, while very important, is beyond the scope of this primer.

Both of the above topics – how long to run a simulation and how many times to replicate it – are discussed in Chapter 10 in the *Applied Simulation Modeling and Analysis Using FlexSim* textbook.

8.2.2 Time-series plot by type

The second charting example is also a system-level time-series chart that shows the contents of the *ContainerStorage* Queue by container type instead of just the total of all containers. The chart is shown in the figure to the right. It again clearly depicts the system's dynamic and stochastic nature.

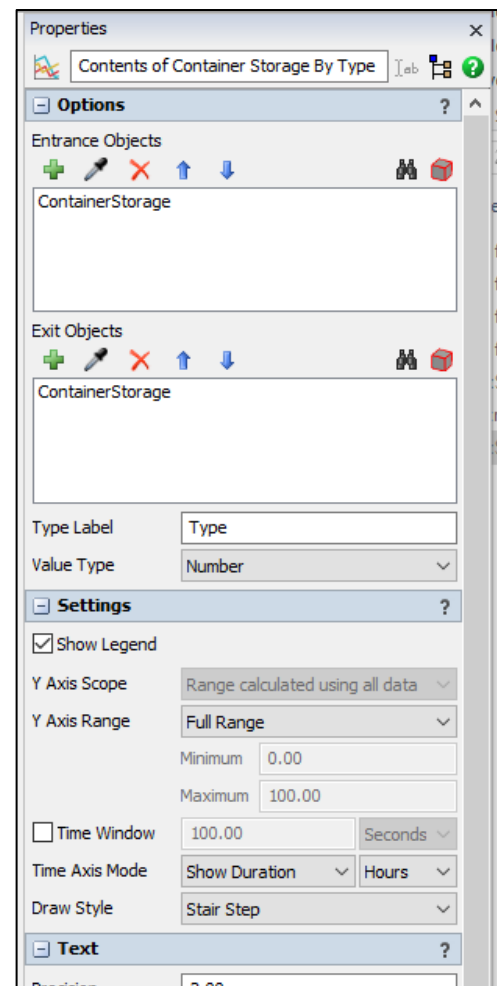


This chart is created like the Content chart discussed above.

- In the **Content** pane of the **Dashboard Library**, select the **WIP By Type** chart type, then select *Line Chart* from the pop-up menu.
- Drag the selected item onto the Dashboard workspace where it can be sized and positioned.

As shown in the figure to the right:

- Rename the chart to something meaningful, such as *Contents of Container Storage by Type*.
- On the **Options** pane, click the  button in the **Entrance Objects** section and select *Select Objects* from the drop-down menu. Again, this generates a list of all of the objects that are currently in the model by category. The *ContainerStorage* object is now displayed in the **Entrance Objects** section.
- From the **Queues** category, select the object *ContainerStorage* so that it is highlighted, and press the **Select** button. The *ContainerStorage* object is now displayed in the **Entrance Objects** section.
- Similarly, on the **Options** pane, click the  button in the **Exit Objects** section and select *Select Objects* from the drop-down menu. From the **Queues** category, select the object *ContainerStorage* so that it is highlighted, and press the **Select** button. The *ContainerStorage* object is now displayed in the **Exit Objects** section.
- As with the Content chart, the only other parameter that should be changed for now is the **Time Axis Mode** property on the **Settings** pane. Change the value from the default *Show exact time and date* to *Show Duration*. Also, change the displayed units from *Seconds* to *Hours*.



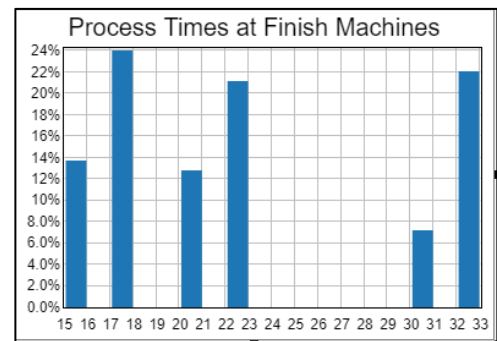
- Since there are multiple values being plotted on only one chart, leave the **Show Legend** box checked (again, on the **Settings** pane).
- Position the chart on the **Dashboard**.
- The default settings for all of the other chart parameters are fine for now.

Note that the chart could track the contents of multiple objects over time; e.g., to track the number of items in the **Queue** and the **Processor**, then the **Exit Objects** would be changed to the **Processor**, *FinishMachine_1*.

Also, note that the **Type Label** value in the **Options** pane can be any label value on the flow item. In this example, the **Type** label is used, which is quite common.

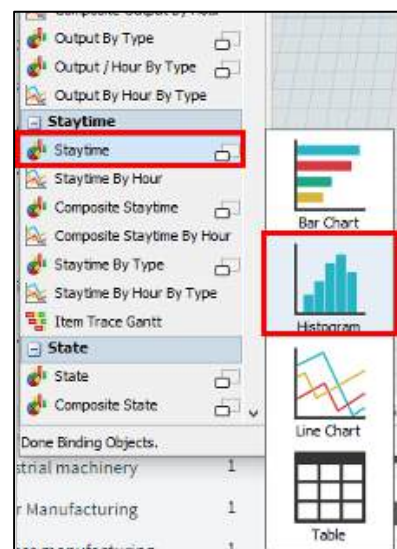
8.2.3 Histogram plot

The third charting example is also a system-level chart, and, as shown in the figure to the right, it considers the time an item spends in the **Processor** (at the Finishing machine). The time an item stays in an object is called “Staytime.” Thus, in this example, the staytime includes the three constant process times (15, 20, and 30 minutes) and the 2-minute setup times when needed, i.e., when the item type changes.



This chart is created like the Content charts described above.

- As shown by the red highlight boxes in the figure to the right, select the **Staytime** chart type in the **Staytime** pane section of the Dashboard Library. Then, select the **Histogram** chart type from the pop-up menu.
- Drag the selected chart onto the **Dashboard** workspace, where it can be sized and positioned.

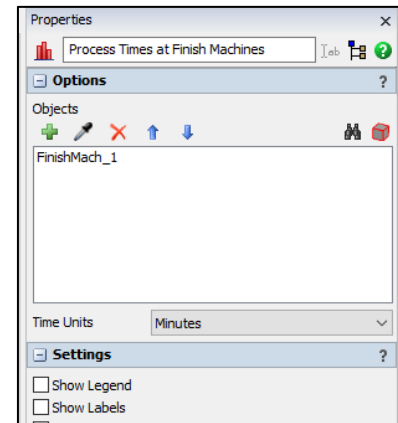


As shown in the figure to the right:

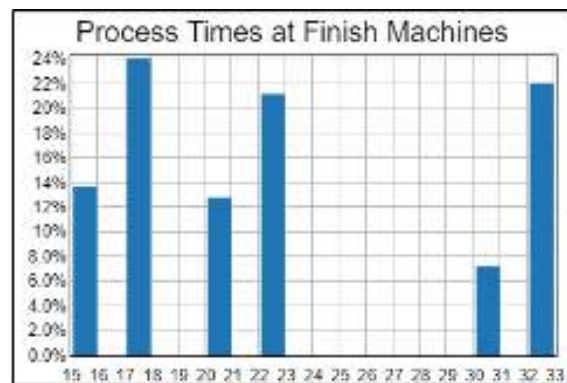
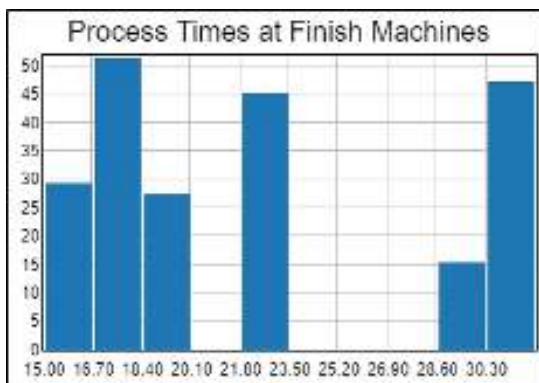
- Rename the chart to something meaningful, such as *Process Times at Finish Machines*.

The only change from the default values is to select which object(s) will be the subject of the plot. Thus, as shown in the figure to the right:

- Click the  button in the **Objects** section of the **Options** pane in the **Properties** window. Then, select the *Select Objects* option.
- Then, select *FinishMachine_1* from the **Processor** section and press the **Select** button at the bottom of the interface.
- Since the chart is only for one object, uncheck the *Show Legend* box in the **Settings** pane.



The resulting chart, when run for 4,800 minutes, should look similar to the one to the left in the figure below. While useful, the histogram can be reformatted to be more readable. Of course, the underlying data are not changed; it is just the way the data are summarized in the chart.



The following two additional steps make the chart correspond to the one to the right in the figure above. The x and y axes are customized for readability and interpretation.

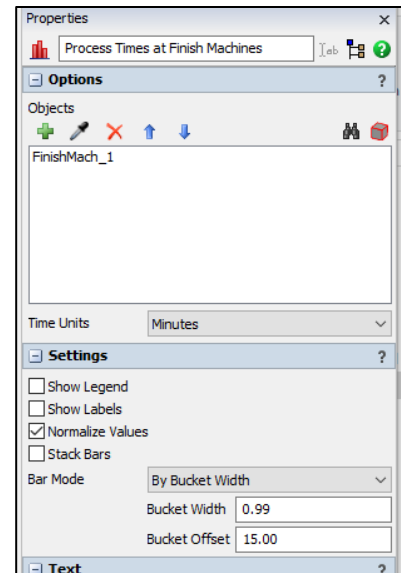
The changes are shown in the figure to the right and described below.

- Change the scale of the y-axis to display the frequency of the histogram as percentages (relative values) rather than the default, the number of occurrences. To do this, check the box **Normalize Values** on the **Settings** pane.
- Change the definition of the bars, the x-axis, from a fixed number of bars (default is 10) to a fixed bucket width.

To do this, again on the **Settings** pane, change the **Bar Mode** from *By Number of Buckets* to *By Bucket Width*. For this example, change the Bucket Width from the default of 50.0 to 1 (representing 1 minute) and the **Bucket Offset**, the starting point of the chart, from the default of 0.0 to 15 (the shortest processing time is 15 minutes with no setup).

Setting the number of bars and bar width for histograms is often done by trial and error. However, there are some methodologies to help with this; three such methodologies are available in *FlexSim*: Sturges, Scott, and Freedman-Diaconis rules. They are available when selecting the **By Bucket Width** property. Discussion of these methodologies is beyond the scope of the primer.

- On the **Text** pane, change the **Precision** property value from the default of 2 (2 decimal places) to 0.



The final dashboard should resemble the figure to the right.

The dashboard has been renamed *Contents and Process Times* since other dashboards will be created later, and the descriptive name helps for reference.

➤ Thus, change the dashboard's name in the text box at the top of the **Properties** window.

Note the following about the charts on the **Dashboard**.

- In the first two plots, the content of the container storage area is growing over time. This is an obvious indicator that the system is unsustainable and not feasible since, in practice, queues cannot continue to grow in size. One solution would be to add another finishing machine, which will be done later in the primer.
- The third plot, the histogram of staytimes, verifies the three deterministic process times by container type (15, 20, and 30 minutes), with approximate frequencies of occurrence of 14%, 13%, and 7%, respectively. The values of 17, 22, and 32 are the process times by type, including setup times; their frequencies are approximately 24%, 22%, and 22%, respectively. Obviously, since the container types are equally likely and not batched, the setup operation is required more often than not.

Also note that the total percentage of Type 1 is 38% (14%+24%), Type 2 is 35% (13% + 22%), and Type 3 is 29%. (7% + 22%). This is close to the assumption that the products are equally likely to occur. Of course, they will not be exactly equal to 33.33% for any run due to sampling error. However, over the long run, the product mix should be equally likely.

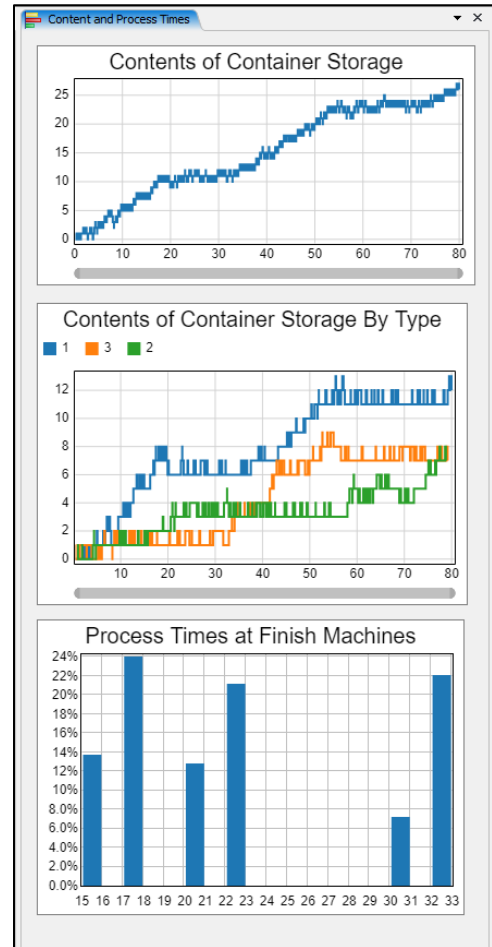
This brief example demonstrates that charts provide a very valuable means for verifying and validating simulation models.



If you haven't already done so, save the model. Recall that it is good practice to save often.



Use the **Save Model As** option in the **File** menu to make a copy of the existing model so that it can be customized beginning in the next section. Again, you can use any file name, but in the primer, the next model is referred to as *Primer_2*.



9 TASK EXECUTERS

Chapter 9 introduces mobile resources, referred to as Task Executors in *FlexSim*, and adds a Finishing Operator to the model.

Task executers (TEs) are mobile or dynamic resources – they move about in a model, typically moving items between objects. A section in the **Object Library**, right below *Fixed Resources*, contains different types of TEs, including **Operator**, **Transporter**, **Crane**, **Robot**, etc. All of these have the same functionality and differ in how they move; e.g., an **Operator** can move in any direction (x, y, and z), whereas an **Elevator** can only move vertically (z-direction).

TEs can represent complex behaviors, but only the most basic properties and a single type of TE, the **Operator**, are covered here. Before discussing how to implement TEs in a *FlexSim* model, a few general concepts are introduced.

The base model for the additions described in this chapter is **Primer_1** that was saved at the end of Chapter 7. However, a copy of that file was saved as **Primer_2**; thus we begin with that file.

9.1 Basic task executer concepts

TEs have speeds (and accelerations); therefore, the location of model objects and the distances between them now become very important.

As the name indicates, TEs execute tasks. Sets of tasks for performing a specific operation, called a “task sequence,” are passed to TEs, typically from fixed resources (**Sources**, **Processors**, etc.). *FlexSim* includes default task sequences, but as in most things in *FlexSim*, they can be customized. For now, only default task sequences are considered. For transporting items, a TE must move items from one object (FromObj) to another object (ToObj). To do so, by default, a TE executes the following task sequence sent to the TE from the FromObj.

Travel from current location to FromObj
Load an item from FromObj
Travel from FromObj to ToObj
Unload the item to ToObj

Tasks other than travel, load, and unload are available in *FlexSim*. However, developing custom task sequences is an advanced topic discussed later in this primer.

Fixed resource objects communicate with TEs via special connections, referred to either as a center-port connection or an S-connection. It is called a center-port connection because it connects the center ports of two objects. It is also called an S-connection because the connection is made by holding down the S key while dragging the mouse between the two objects.

Objects have center ports in addition to the input and output ports that were introduced earlier. Center ports are not for item flow but for communications between objects. Communications can be bidirectional between the objects; this is in contrast to the unidirectional flow between input and output port connections (A-connections).

A TE will often receive more requests to perform task sequences than can be met at that time. For example, a TE may receive a request while performing a task sequence. Therefore, TEs can queue requests to perform task sequences and can use different means to process requests in the task queue, e.g., first-in, first-out, or with priorities.

TEs not only receive tasks but can also send them to other TEs. Thus, TEs are like working managers – they will carry out a task-sequence request unless they are busy when they receive the request. If they are busy when they receive the request, they can send it to an available TE. If all associated TEs are busy, the receiving TE puts the task-sequence request into its work queue. To implement this, decide which TE is the working manager and connect it to all associated TEs via A-connections (from the working manager TE to the associated TEs).

If all TEs are considered the same and no one acts as a manager, then a **Dispatcher** object is used. The only function of the **Dispatcher** is to allocate work and maintain the work queue for all of the associated TEs. In this case, the fixed objects communicate with the **Dispatcher** and not directly with the individual TEs. Therefore, the fixed objects are connected to the **Dispatcher** with S-connections through their center ports, and the **Dispatcher** is connected to each TE with an A-connection (from the Dispatcher to the TE).

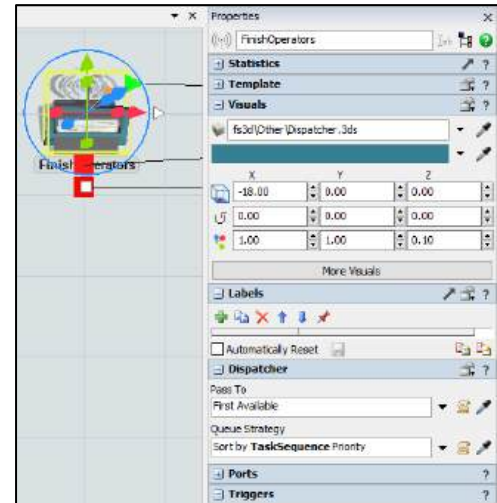
By default, a TE travels between objects in a straight-line path, traveling the shortest distance between two objects. A TE's default travel path does not consider other objects in its path. However, there are several alternative means to control the travel path of a TE – one uses path networks, and another uses the A* algorithm. Both approaches are discussed later in the primer.

9.2 Adding a finishing operator to the model

Begin with the basic model from the previous chapter, named `Primer_2`.

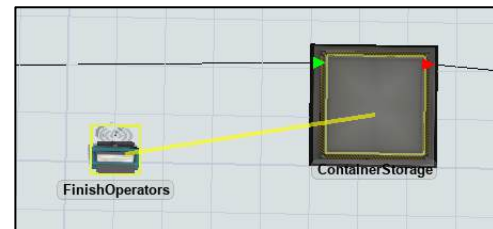
Since more than one finishing operator may be needed, use both the **Dispatcher** and **Operator** objects.

- Drag out a **Dispatcher** object from the *Task Executors* pane of the **Object Library** to the modeling surface. As shown in the figure to the right:
 - Name the object *FinishOperators*.
 - The object will be more precisely located later, but for now, set $(x = -9.0, y = 0.0, z = 0.0)$. This object can be placed anywhere, but this is a convenient location for now. It also illustrates another means to position an object in the workspace instead of dragging it to a location.
 - Resize the object using the size settings $x = 1.0, y = 1.0$, and $z = 0.1$. This minimizes the object's presence since the **Dispatcher** object is not an actual entity in the system.



The **Queue** makes requests of the **Dispatcher**, i.e., it sends task sequences to move items from the **Queue** to the **Processor**. The **Dispatcher** then sends the request (task sequence) to an **Operator** that it manages. Implement this aspect in the model as described below.

- As shown in the figure to the right, make an S-connection between the **Dispatcher** and the *ContainerStorage Queue* by holding down the S key, selecting the **Dispatcher** so that it is highlighted by a yellow box, dragging the mouse to the **Queue**, and releasing once the **Queue** is highlighted with a yellow box.

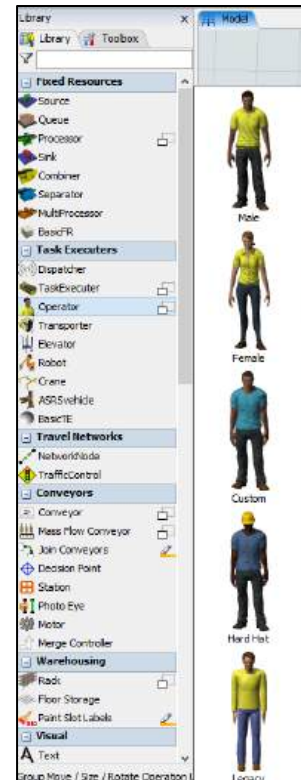
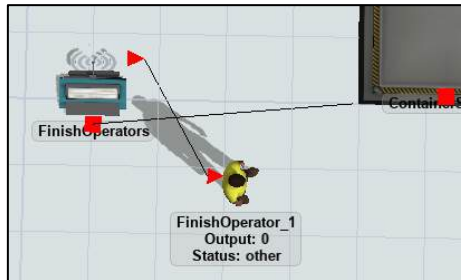


There should be a connection between the two objects' center ports, as shown in the figure to the right. When making center-port connections, the order of connections is not important since it involves bi-directional communications; i.e., the connection can be made from the TE to the fixed object or from the fixed object to the TE.



- Select an **Operator** object from the *Task Executors* pane of the **Object Library** . Note there are various types of **Operators** to choose from - they are functionally the same and only differ by appearance. The appearances can be customized, but that level of detail is not considered in this example. Therefore, any **Operator** type is fine, but the primer will use the *Male*.
- Drag the *Male* image to the modeling surface. At the moment, it can be placed anywhere.
- Name the object *FinishOperator_1*.
- Make an A-connection from **Dispatcher** to *FinishOperator_1* by holding down the A key, selecting the **Dispatcher** so that it is highlighted by a yellow box, dragging the mouse to the finishing operator, and releasing it once the **Operator** is highlighted with a yellow box.

As shown in the figure to the right, there should be a connection between the **Dispatcher**'s output port and the **Operator**'s input port. Ensure the connection is from the **Dispatcher** to the **Operator** since it is a one-way flow of information.



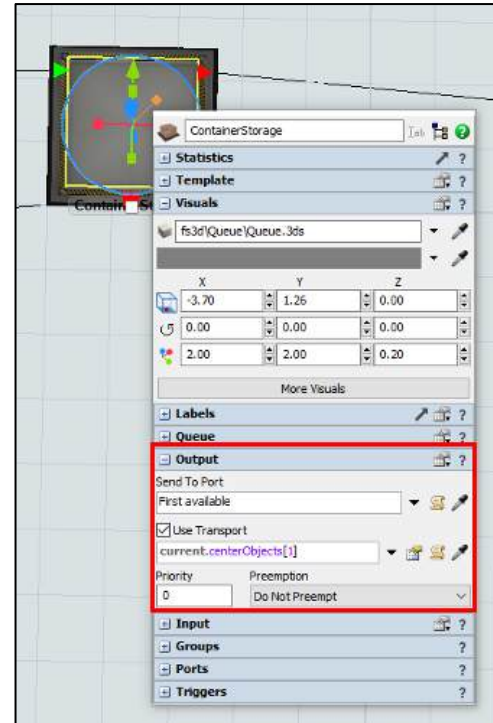
Now that all of the objects can communicate, the **Queue** needs to be able to send a task sequence to the **Dispatcher**, telling it what to do. In this case, the **Queue** requests the TE to transport an item from the **Queue** to the **Processor** (*FinishMachine_1*). This is accomplished as described below and shown in the figure.

- On the **Output** pane of the **Queue** object (*ContainerStorage*), check the **Use Transport** box as shown in the red box in the figure to the right.

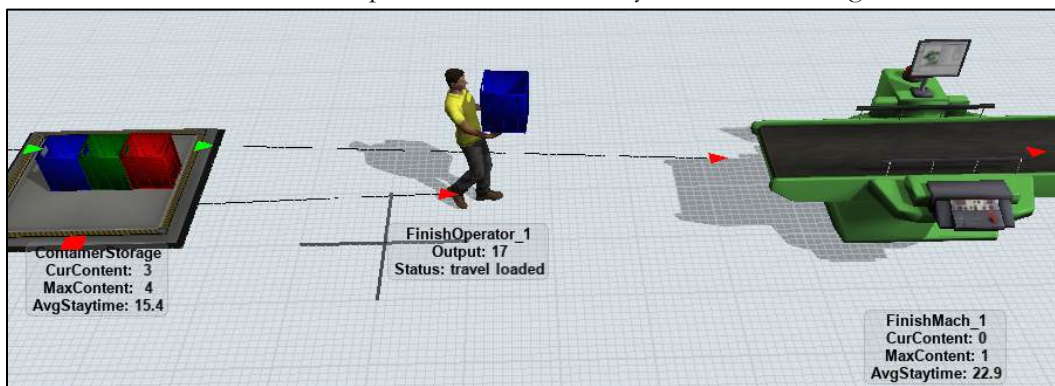
The default routing logic, the **Send To Port** property's value *First Available*, is fine since now there is only one object downstream from the **Queue**, which is the **Processor**.

By default, the task sequence goes to whatever object is connected to the **Queue**'s center port; in this case, that is the **Dispatcher**. This action is accomplished by the default **Use Transport** property value, which is the *FlexSim* command `current.centerObjects[1]`. This command is interpreted as the object connected to the current object's first center port. In this case, the current object is the **Queue**, and the object connected to its first center port is the **Dispatcher**.

Once the **Dispatcher** receives the task sequence, it sends it to the **Operator** when the **Operator** is available.



- Verify that all the connections and settings are correct by pressing the **Reset** and **Run** buttons and observing that the **Operator** picks up a container in the **Queue** and transports it to the **Processor**. This call for transport occurs when the **Queue**'s downstream object, the **Processor**, becomes idle and is available to receive the next item to process. Such an activity is shown in the figure below.



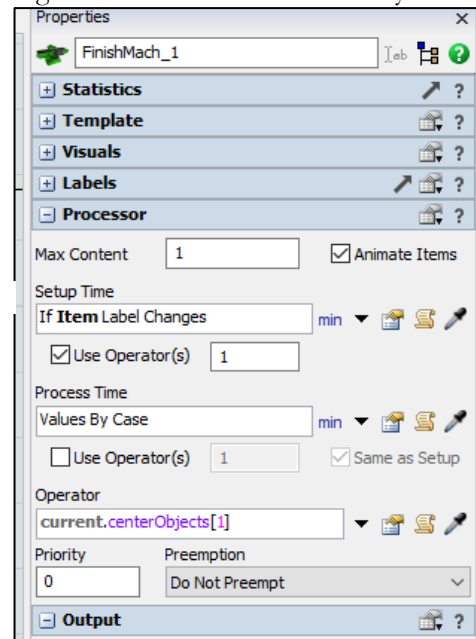
You may need to slow down the model's speed to see this (e.g., Run Speed = 0.1) or stop the model as a transport occurs.

- Observe the **Operator**'s behavior – after unloading the container onto the finishing machine, the **Operator** waits at the **Processor** until it receives the next task from the **Queue**, which currently is its only source of tasks. The **Operator** waits at the **Processor** because the default task sequence (Travel-Load-Travel-Unload) discussed earlier is being followed. Once the container is unloaded, if the **Operator** has no more task sequences waiting, it becomes idle where it is, at the **Processor**.

If the **Operator**'s only task is to move items between the **Queue** and **Processor**, it might make sense to change the model's logic so that the **Operator** returns to the **Queue** once a container is loaded onto the **Processor**. This is easy to do but may not be what happens in the real system. In this case, the **Operator** has other tasks.

The **Operator** also needs to perform the setup operation at the finishing machine. This is done similarly to the process just described. As shown in the figure to the right and as described below, modify the **Processor** as follows.

- On the **Processor** pane, check the *Use Operator(s)* box below the *Setup Time* property.
- The value of the *Operator* property, near the bottom of the **Processor** pane, remains the default. `current.centerObjects[1]`
The **Processor** will use the object connected to its first Center port to do the setup operation, which will be the Dispatcher.
- Make an S- or Center connection between the **Processor** and the **Dispatcher**.

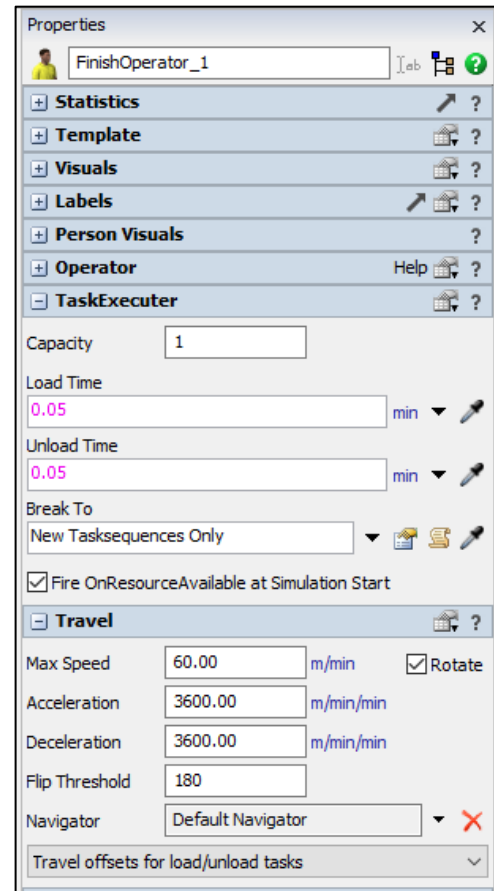


Since the performance of a system depends on speeds and distances, the **Operator** and other **Task Executer** properties are important. The default value for an Operator's speed is 120 meters/minute. This seems a bit fast for this operation since the average walking speed is about 84 meters/minute. Therefore, use a value somewhat slower than the average walking speed, 60 meters/minute.

- As shown in the figure to the right, change the **Max Speed** value on the **Travel** pane of *FinishOperator_1* from the default of 120 to 60.

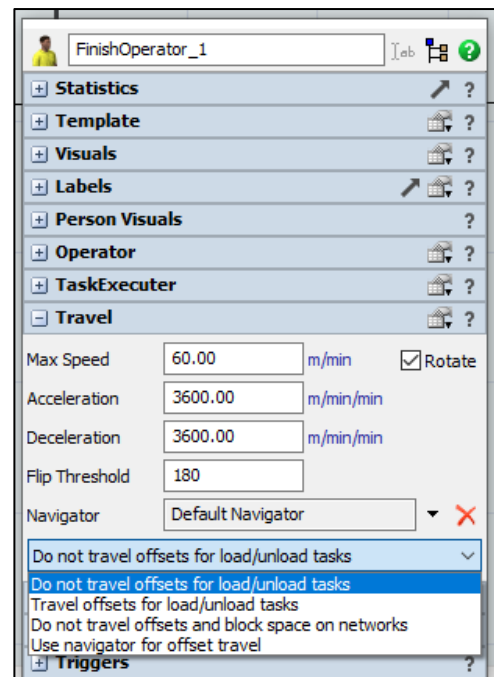
Also, consider the time it takes to load a flow item at the originating object and the time to unload an item at the destination object for a transport task sequence. The default time is 0.0.

- In the **TaskExecuter** pane, change the **Load Time** and **Unload Time** property values to 0.05 minutes (3 seconds).



By default, **Task Executors** travel “offsets” when loading or unloading items. This means the TE travels inside the object that contains the item in order to load it. In this example, we have the **Operator** travel to the end of the object and not go inside.

- As shown in the figure to the right, use the dropdown menu to change the last property in the **Travel** pane to *Do not travel offsets for load/unload tasks*.



Since the **Operator** object is mobile, it can be anywhere on the modeling surface when a running model is stopped. This location will be where the **Operator** starts when the model is run again.

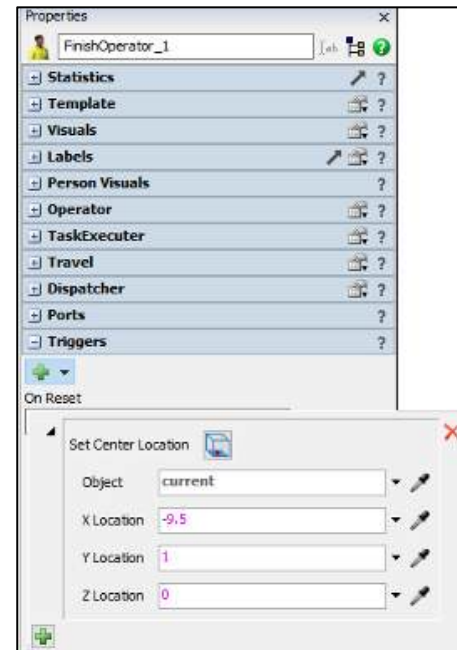
For consistency in comparisons and analyses, having the task executor start at the same location each time a model is reset and run is good practice.

Therefore, in this case, set the location of the *FinishOperator_1* near the Finishing Machine and the container **Queue**.

- As shown in the figure to the right, use the  button on the **Operator** object's **Triggers** pane to add an **On Reset** trigger, then select the **Set Center Location** option.
- Change the default location values **X Location** from 0.0 to -9.5 and **Y Location** from 0.0 to 1. **Z Location** remains 0.0.

Now, the **Operator** moves to this location whenever the **Reset** button is pushed. To verify this:

- **Reset** and **Run** the model. Note that the **Operator** returns to the same location when the model is reset.



If you haven't already done so, save the model. Recall that it is good practice to save often.

We now have a basic model of the unfinished containers arriving at and being processed at the Finishing Area.



Use the **Save Model As** option in the **File** menu to make a copy of the existing model so that it can be customized beginning in the next section. Again, you can use any file name, but in the primer, the next model is referred to as *Primer_3*.

10 BASIC LOGIC WITHIN THE PROCESSOR OBJECT

Chapter 10 examines the Processor's operation, which is one of the basic 3D objects. It is meant to help the reader better understand what is happening behind the scenes in the basic 3D objects. Thus, this chapter does not add features and capabilities to the primer model.

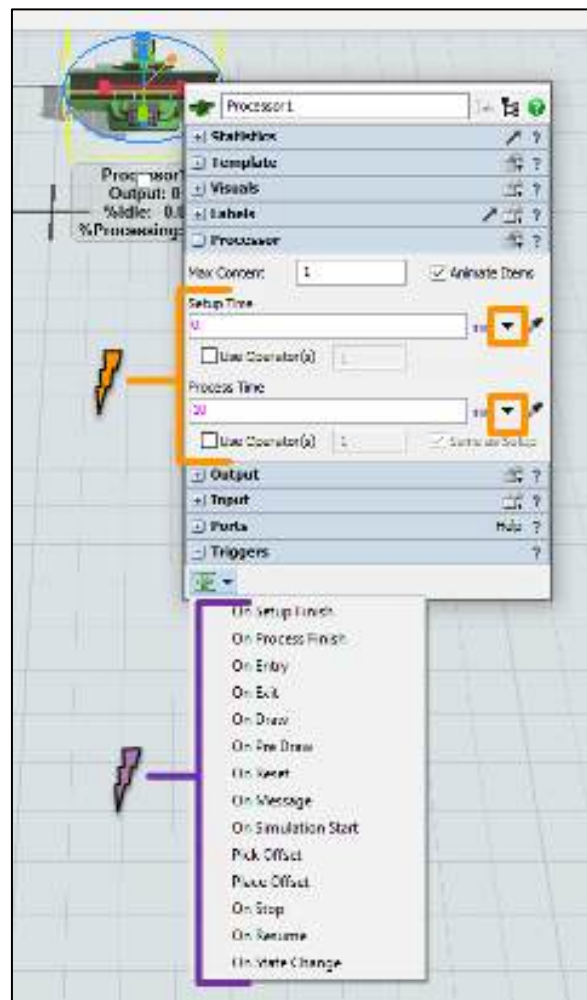
Before discussing additional types of objects, this chapter provides a glimpse of what happens inside an object from the time a flow item enters it until it leaves. **Note that this section is not required to build and analyze models in FlexSim; therefore, it may be skipped and revisited later.** This chapter is for the users who want to peek into what is happening “under the hood.”

Each *FlexSim* object contains the logic to perform tasks corresponding to a real system's operational functions. The **Processor** object is used to illustrate what happens to a flow item and an object as an item passes through the object. Recall that the primary purpose of a **Processor** is to invoke a planned delay on items flowing through a model.

Every item flowing through a **Processor** is delayed by the value specified by the required **Process Time** property and the optional **Setup Time** property. The default times are 10 and 0, respectively. Specifying how the values of these delays are determined is set via the dropdown menus highlighted by the small orange boxes in the figure to the right. The orange lightning symbols will represent these two properties in the next figure. Checking the **Use Operator(s)** property box may increase the time an item spends at the **Processor**, especially if the required operator is a resource shared with other objects and thus must travel to the **Processor** when needed.

FlexSim objects contain the means to specify optional actions that can be performed on items as they flow through an object, and a specified set of conditions occurs in a simulation. These optional actions are called “triggers.” The triggers on a **Processor** object are shown in the figure to the right. The purple lightning symbols will represent these actions in the next figure.

While each object type is unique, they do share common actions. For example, a modeler may want to change the color or shape of an item when it exits an object; in this case, the **On Exit** trigger would contain the information needed to cause an item to change when it leaves an object.



Many triggers are common across most objects, such as *On Entry*, *On Reset*, etc. However, the *On Setup Finish* and *On Process Finish* triggers are unique to the **Processor** object.

The figure to the right shows when various triggers “fire” in an object as an item flows from left to right over time. To keep the explanation simple, not all possible triggers are discussed here.

The first trigger to fire when an object becomes empty and idle is *Pull from Port*. This is used to decide what item to process next in a “pull” system, which will be discussed later in the primer.

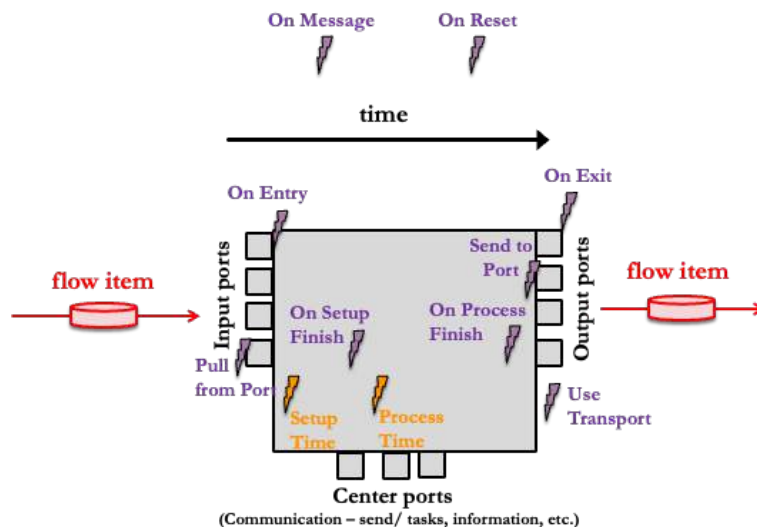
Therefore, from our current perspective, the first trigger is *On Entry*, and the last trigger to fire is *On Exit*. As the names imply, the user can take action when an item enters and leaves the object. Example actions could be changing the item’s or object’s color or getting or setting a label value. Of course, the do-nothing option is the default.

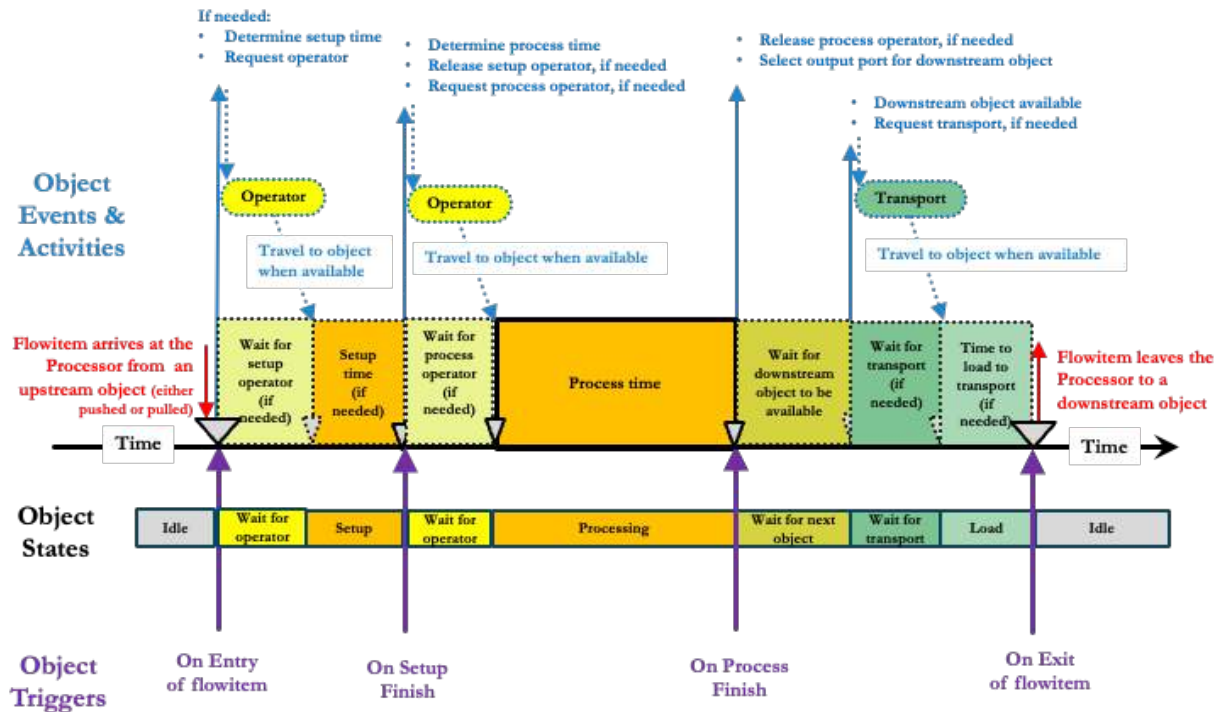
The orange lightning symbols in the figure above represent property values, the *Setup Time* and *Process Time*, that are set as an item moves through an object.

The purple lightning symbols above the timeline, *On Message* and *On Reset*, are triggers originating from an external source, not initiated by a flow item. One way for objects to communicate with each other is via messaging, so the *On Message* triggers are the actions performed when an object receives an external message. Messaging is a more advanced topic and is not discussed here. Another external trigger is *On Reset*, which fires when a model is reset. In this case, an action might be to return an object’s color to its default. If an object is normally green but changes color to red when it is the breakdown state, then it needs to be set back to green if the simulation ends with an object in the down state.

The following figure illustrates the most common activities and delays in a **Processor** object and the order in which they are incurred. In this case, the **Processor**’s assumed maximum content is one. A **Processor** can process multiple items simultaneously, but the default is one item at a time. The black horizontal line in the middle of the figure indicates the passage of simulated time. The red arrows on the figure’s left and right show a flow item’s entrance and exit, respectively.

The optional triggers that cause custom activities to occur, such as *On Entry* and *On Setup Finish*, are shown in purple at the bottom of the figure. The triggers spawn the actions shown in blue at the top of the figure.



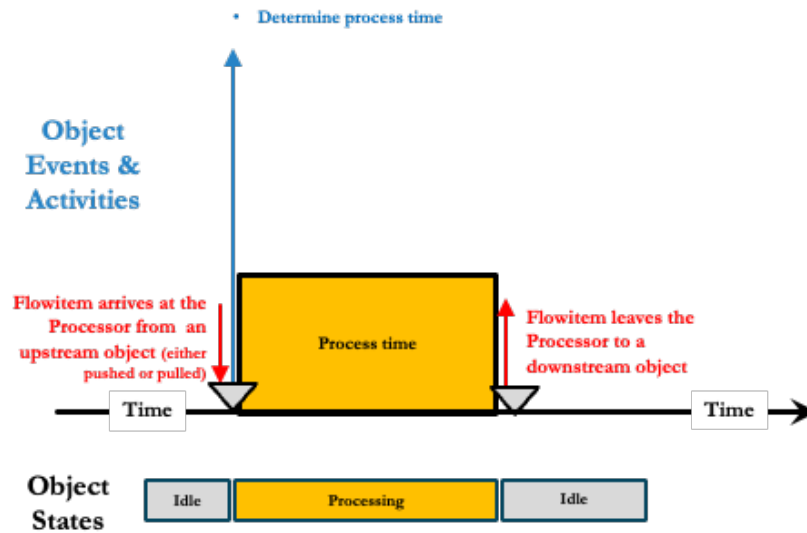


A quick narrative of the figure is provided. When a flow item enters an object, it is determined whether a setup is needed. If it is, a setup time is determined. Next, it is decided whether an external resource, such as an operator, is needed to perform the setup. If the setup requires an external resource, the flow item is delayed until the resource is acquired and travels to the object. The item is then delayed for the duration of the setup time. When the setup is complete, the **On Setup Finish** trigger is available to perform other actions if the modeler needs it.

Once the setup activity is complete, the processing time is determined, e.g., it may be a constant value or a sample from a probability distribution. Next, it is decided whether an external resource, such as an operator, is needed to do the processing. If the process requires an external resource, the flow item is delayed until the resource is acquired and travels to the object. The item is then delayed for the duration of the processing time. When the processing is complete, the **On Process Finish** trigger is available to perform other actions if the modeler needs it.

Before an item can be released from the **Processor**, its downstream object must be available. Thus, if the downstream object is not available, the item waits in the **Processor** until the downstream object is available. Once available, it is determined if the item needs to be transported to the downstream object. If so, a transporter is called, and the item waits until the transporter arrives at the object. The item usually must be loaded onto the transporter, which is another delay in the **Processor**. Once loaded, the item exits the **Processor**, and the **On Exit** trigger can be used to perform additional actions. Once an item has exited, the **Processor** can accept another item if one is waiting to enter. If no item is waiting to be processed, the **Processor** goes into the idle state.

The figure above illustrates how a flow item can be delayed for a much longer time in a **Processor** than just the duration of the *Process Time*. The case where an item is only delayed by the process time is shown in the figure below. In this case, no setup is required, no other resource is needed for processing, the downstream object is available, and no transport to the downstream object is required.



PART III – FURTHER DEVELOPMENT OF THE FINISHING AREA

The model of the Finishing Area is enhanced by changing object graphics and routings and introducing some of *FlexSim*'s many modeling support tools.

- Chapter 11 changes the Finishing Machine and Container Queue graphics and employs new item routing rules. It also introduces model views.
- Chapter 12 uses the Empirical Distribution tool to define the product mix of containers entering the Finishing Area. It also uses the Empirical Distribution tool to fit system data to a probability distribution that best describes the arrival process of containers to the Finishing Area.
- Chapter 13 introduces two types of data tables that help organize information used in *FlexSim* models: Model Parameter Tables and Global Tables.
- Chapter 14 adds several types of downtime to the model. Break and lunch times are added for the Finishing Operator, i.e., when the operator is unavailable to do system work. Two types of downtimes are considered for the Finishing Machines. The first is a quality check that occurs on a fixed clock-based schedule; the second is randomly occurring breakdowns that are based on the object's state. Machine breakdowns require a Finishing Operator for repairs; the quality checks do not require an operator. Pie charts are introduced to summarize the utilization and percentage of time the Finishing Machines are in various states. The charts are added to a new dashboard.

11 CUSTOMIZE OBJECTS TO REPRESENT SYSTEM BEING MODELED

Chapter 11 changes the Finishing Machine and Container Queue graphics and employs new item routing rules. It also introduces model views.

This section discusses various topics and changes that further customize the model to better represent the planned production system. These include:

- Change the finishing machine graphic to make it look more like the machine in the real system.
- Create a second finishing machine.
- Change the shape of the queue from upstream processes.
- Implement an initial decision rule for routing items to finishing machines. This will later be changed to a multi-criteria decision – the shortest processing time but with a wait time threshold.
- Implement a decision rule for routing items based on current conditions, e.g., if a buffer is full, then where should an item be sent?
- Change to a more general distribution for the mix of containers by using an Empirical probability distribution rather than the Discrete Uniform.
- Set views and toggle between Working and Presentation views.

The base model for the additions described in this chapter is *Primer_2* that was saved at the end of Chapter 9. However, a copy of that file was saved as *Primer_3*; thus, we begin with that file.

11.1 Changing object graphics

While a 3D shape can be created to represent any system object, many such shapes are readily available. For example, 3D Warehouse (<https://3dwarehouse.sketchup.com>) contains millions of 3D objects that have been created in the 3D modeling software *SketchUp*.

Here are a few comments about *SketchUp* and using shapes developed in *SketchUp* in *FlexSim* simulation models. Of course, *FlexSim* can use a variety of 3D file formats. For the latest list and more information on importing 3D graphics into *FlexSim*, see the “Creating and Importing Custom 3D Objects” section of the ***FlexSim User Manual***.

- Shapes from 3D Warehouse can be downloaded for free and used in *FlexSim*. Unless the user modifies the shapes, *SketchUp* itself is not required, just the .skp file.
- Since file formats change often, it is best not to use the latest version of the format since it may not be compatible with *FlexSim*. Therefore, downloads should be in *SketchUp* file format 2022 or earlier.
- *SketchUp* is now a web-based application, but older versions are available at the SketchUp Help Center (<https://help.sketchup.com/en/article/60107>).
- The size of the *SketchUp* model, especially the number of polygons, affects the *FlexSim* model size and its run time. Therefore, very complex graphics should be avoided unless they are really needed, and the model is run on a powerful computer.

11.2 Changing the Finishing Machine's (Processor) graphic

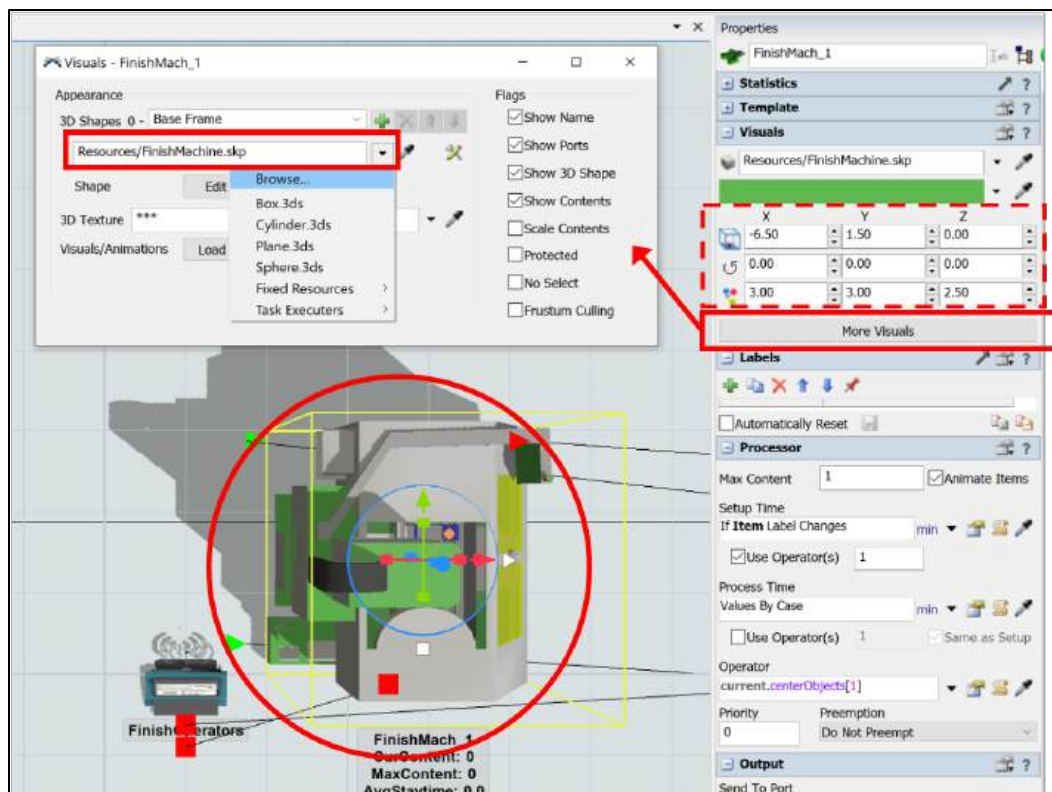
The shape of the Finishing Machine is changed from the default *FlexSim* graphic to the graphic in the *SketchUp* file named *FinishMachine.skp*. The process for doing so as described below and shown in the following figure.

- In the **Processor's Properties** window and **Visual** pane, select the *More Visuals* button, which is highlighted by a red box in the righthand portion of the figure. This opens the **Visuals** window, as indicated by the arrow in the figure.
- Select the second dropdown menu button below **Appearance**; it is to the right of the textbox containing *fs3d/Processor.3ds*, which is the path and file name for the default **Processor** shape.
- Select the top option, *Browse...*, in the dropdown box.
- Use the *Browse* option to search your computer to find the *SketchUp* file. In the example in the figure below, the path *Resources/FinishMachine.skp* indicates the *SketchUp* file *FinishMachine.skp* is in the folder *Resources*, which is in the same location as the *FlexSim* model.

By default, in *FlexSim*, the graphics media are embedded within a model so that the model can be copied and shared without having to provide the external graphics files.

As shown in the dashed-line red box in the figure below,

- Change the location properties to $x = -6.5, y = 1.5, z = 0$. The object could have been placed in the correct location by moving it manually on the layout.
- Change the object size properties to $x = 3, y = 3$, and $z = 2.5$. All values are in meters; i.e., each machine is three meters square and 2.5 meters high.



The **Processor** should now look like the image in the circle in the figure above.

Another Finishing Machine is needed since the average input rate is equal to the average processing rate; therefore, it is a good time to add it now.

- Copy *FinishMach_1* using Cntl-C shortcut key combination, then click on the layout near where the machine will be located and paste using the Cntl-V keyboard shortcut. This is a common way to copy and paste in most *Windows* applications. Alternatively, an object can be selected, right-clicked (click the right mouse button), then select **Edit**, and then either *Copy* or *Paste*.

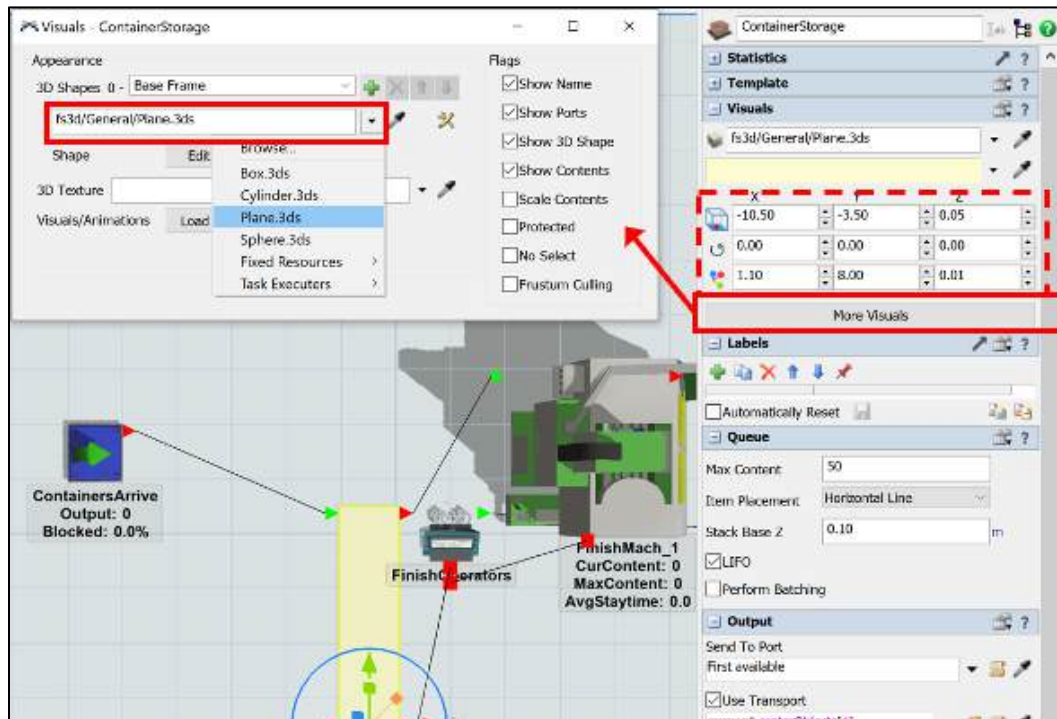
When copying and pasting objects, it is best to copy (Cntl-C), then click on the modeling surface approximately where the new object is to be located, then paste the copied object (Cntl-V). If you don't click on the modeling surface, the pasted object may become a subset of the copied object. This is a good feature in some cases, especially where you want to build sub-models, but that is a more advanced topic.

- Rename the copied object as *FinishMach_2* and place it in the proper location on the layout ($x = -6.50$, $y = -2.40$, and $z = 0.0$).
- Make the flow and communications connections:
 - A-connect from the **Queue** named *ContainerStorage* to the Finishing Machine.
 - A-connect from the Finishing Machine to the **Sink** named *NextProcess*.
 - S-connect between the **Dispatcher** (*FinishMachines*) and *FinishMachine_2*.
- On both finishing machines, uncheck the **AnimateItems** box at the top of the **Processor** pane next to the **Max Content** property. By default, items move across a **Processor** as they are being processed; however, in this case, containers are processed at a fixed location in a Finishing Machine.

Items can remain static on a **Processor** while being processed by unchecking **Animate Items** on the **Processor** pane. Checking or unchecking the box is for visual purposes only and does not affect the model's behavior; i.e., the processing time is the same whether or not the item is conveyed across the object.

11.3 Changing the Container Storage's (Queue) graphic

Next, the **Queue** or *ContainerStorage* object is customized. Its shape is changed to a *Plane*, which is a basic 3D shape that comes with *FlexSim*. The process is similar to changing the graphic on the **Processor** in the previous section. Refer to the figure below and the instructions that follow.



- In the **Queue's Properties** window and **Visual** pane, select the *More Visuals* button, which is highlighted by a red box in the righthand portion of the figure above. This opens the **Visuals** window, as indicated by the arrow in the figure.
- In the **Visuals** pane, select the second dropdown menu button below **Appearance**; it is to the right of the textbox containing *fs3d/Queue/Queue.3ds*, which is the path and file name for the default **Queue** shape.
- Select the fourth option, *Plane.3ds*, in the dropdown box.

As shown in the dashed-line red box in the figure above,

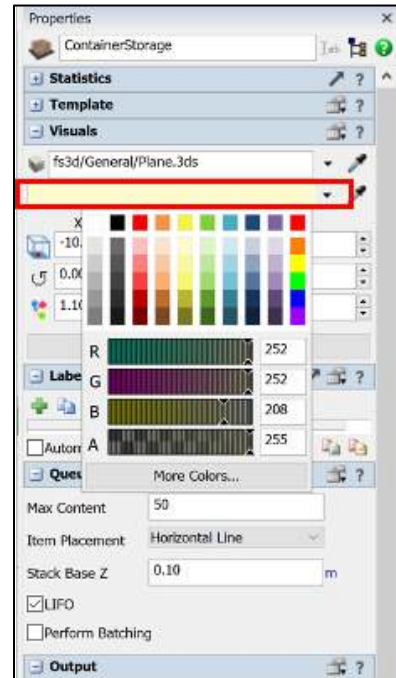
- Change the location properties to $x = -10.5$, $y = 3.5$, $z = 0.05$. Again, the object could have been placed in the correct location by moving it manually in the 3D workspace.
- Set the object's size to $x = 1.1$, $y = 8$, and $z = 0.01$. Note that the x value should be set slightly larger than the container size so that the container will fit.

If a **Queue's** size is less than or equal to the size of an item that is trying to fit into the **Queue**, it will not fit; as a result, *FlexSim* will position it outside of the **Queue**. The logic will not be affected, but it will look strange.

Change the object's color, in this case, to light yellow.

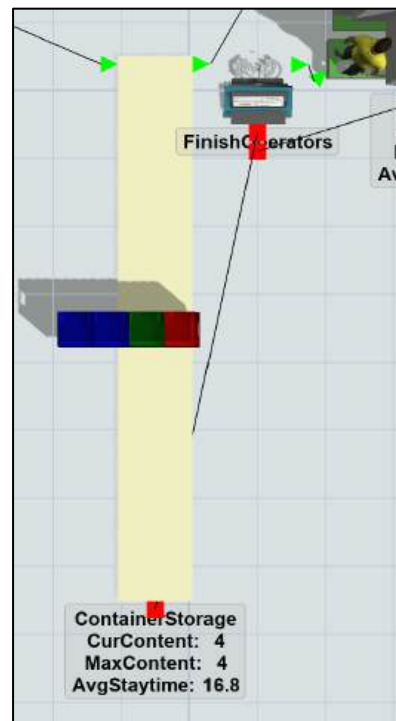
- Select the object's color dropdown box in the **Visuals** plane, as highlighted by the red box in the figure to the right. This opens many options for color – selecting one of the thumbnails, setting the RGB values, or more options through the *More Colors* button. In this case, select the lightest yellow color from the thumbnails.

Of course, the color does not affect the system's behavior or performance, but it may help with validation. For example, if it is standard practice to paint floor storage areas yellow, then this change helps stakeholders identify with the model.



- **Reset** and **Run** the model and observe the queuing process.

The containers likely queue up, as shown in the figure to the right, and thus do not conform to the shape of the storage area. This is because only the general shape of the object has been changed; logically, items still queue in a horizontal line from the front of the object to the back of the object and beyond. Basically, items line up from right to left.



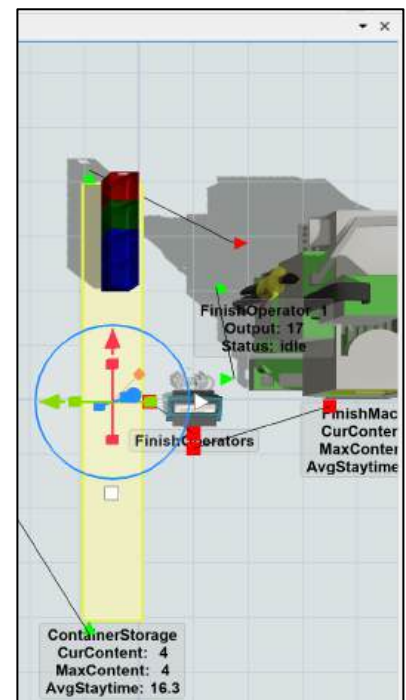
To correct this, make the following changes on the **Queue's Visuals** pane. The revised properties are shown in the dashed red box in the figure to the right.

- Rotate the **Queue** 90 degrees about the z-axis so that it is vertically oriented and not horizontal. Set the α rotation to 90.0 .
- Since the item is now rotated, reverse the sizes in the x and y directions, i.e., $x = 8.00$ and $y = 1.10$.
- Since the size and rotation have changed, the object needs to be repositioned; i.e., set the locations to $x = -10.5$ and $y = 0.00$.

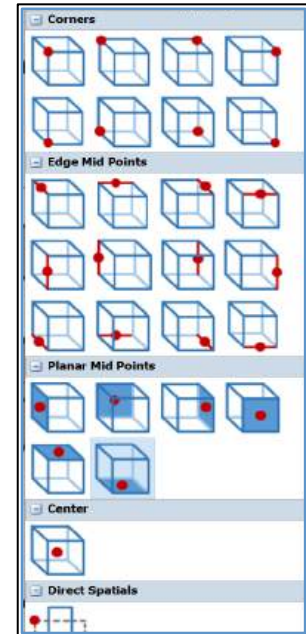


- **Reset** and **Run** the model and observe the queueing process. It should resemble the figure to the right where the containers now line up as expected.

This change may appear to be only for appearances, but in this case, the queue orientation will affect estimated system performance. When an operator moves containers from the storage area to the machines, the location of the containers to be moved will affect the operator's travel time and, thus, performance.



Notice a small cubic icon to the left of location parameters in the **Visuals** pane of an object's properties. Clicking on the cubic icon will display the window shown to the right. *FlexSim* provides the different possible object reference points shown in the figure. The default reference is the midpoint of the object's bottom surface. For example, if the location parameters are specified as $x = 10.0$ and $y = 2.0$, then the center point of the object's bottom surface will be located at (10, 2). The reference point for the object can be changed by clicking on the icon of the desired reference point.



The **Source** (*Containers Arrive*) and **Sink** (*Next Process*) shapes and sizes are not changed. Since they are just boundaries and not physical objects in the real system, they can be moved anywhere in the model view, or they can be minimized.

11.4 Item routings

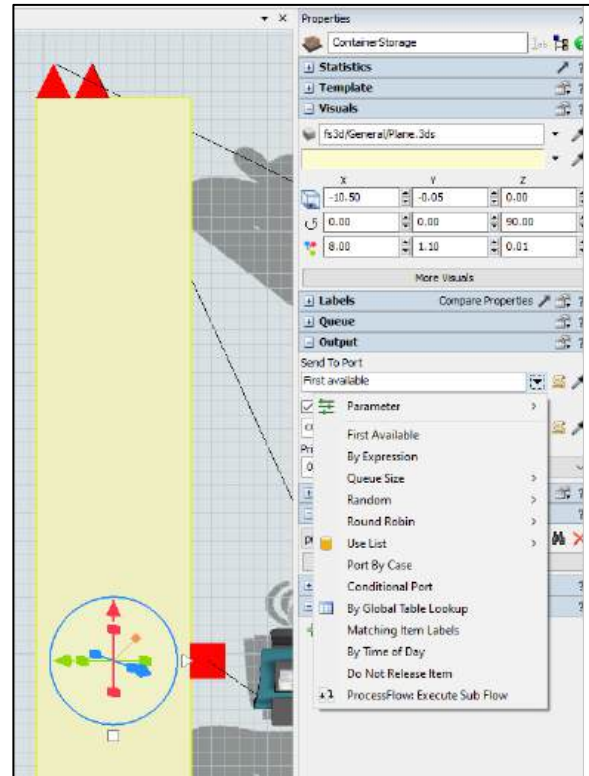
Routings are a key part of modeling process flows. As items flow through a system, they may follow different paths due to the current condition or state of a system. For example, different paths may be followed depending on the type of object, the availability of a resource, the perceived waiting time to secure a resource, etc. Paths may also change randomly, such as a customer choosing between two available servers.

Earlier in the simpler model, all items (containers) followed the same routing path – they all flowed through the four basic objects in sequence: **Source** to **Queue** to **Processor** to **Sink**. But now there are two **Processors** (finishing machines) in parallel; i.e., an item can go to either one. Thus, there is a routing decision in the **Queue**. Also, since the capacity of the **Queue** is limited, there needs to be a routing decision at the **Source** – either go to the **Queue** or somewhere else if the **Queue** currently has no remaining capacity. Both of these behaviors are considered in the next two sections.

11.5 Choosing a routing based on availability

The **Queue** for storing containers awaiting finishing has the option of sending a container to either finishing machine 1 or 2. For now, the default setting, *First Available*, is used. The routing logic is controlled at the **Send To Port** trigger on the **Output** pane, as shown in the figure to the right. Note in the figure that the dropdown menu offers many routing options to move items from the **Queue** to one of the machines.

The *First Available* option works as follows. When an item is ready to leave the **Queue**, it checks to see if the downstream object connected to Port 1 is available to receive the item. If it is, then the item is sent – in this case, the item leaves the **Queue** and arrives at the first machine in zero simulated time. Recall that by default, it takes no time to move from object to object in a model. If the first machine is not available, then the **Queue** checks to see if the downstream object connected to Port 2 is available to receive the item. If it is, then the item is sent – in this case, the item leaves the **Queue** and arrives at the second machine in zero simulated time. If both machines cannot accept the item, then the item waits in the **Queue** until one of the downstream machines becomes available.



Note that the port connections between objects drive routing decisions since they set the order for checking downstream objects for availability.

Using the *First Available* routing logic usually results in the object connected to the first port receiving more items since it is always checked first. Similarly, the object connected to the last output port will receive the fewest items.

11.6 Choosing a routing based on current system conditions

The container storage area has a limited capacity (*Maximum Content's* property value on the **Queue** is 50). Therefore, provisions must be made for the condition when the **Queue** is at capacity and cannot accept any more containers.

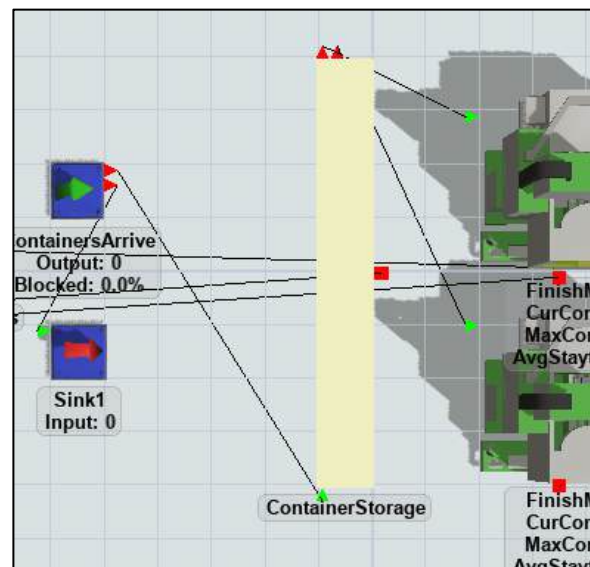
If nothing is done, items will be backed up in the **Source** and are not released until there is space in the **Queue**. Other than the **Source** statistic *Blocked*, this hides the fact that there is insufficient capacity in the **Queue**. In the real system, if there is no place to put arriving containers, they must be dealt with. Thus, exceeding capacity needs to be addressed, even if just by tracking how often this occurs.

To provide a means to track this condition, divert containers to a second **Sink**, named *ReDirectedContainers*, if the **Queue** is full. Thus, no action is taken if capacity is full, but it will be noted when this occurs. Implement this in the model as described below.

As shown in the figure to the right:

- Drag a **Sink** object from the **Object Library** onto the modeling surface and name it *ReDirectedContainers*.
- A-connect the **Source** to the new **Sink**.

As shown in the figure to the right, there are now two output port connections from the **Source**, one to the **Queue** and one to the second **Sink** that was just created. As described earlier, the default decision rule for routing, *FirstAvailable*, is used so that items go out Port 1 to the **Queue** if there is available capacity and go out Port 2 to the **Sink** if the **Queue** is full.



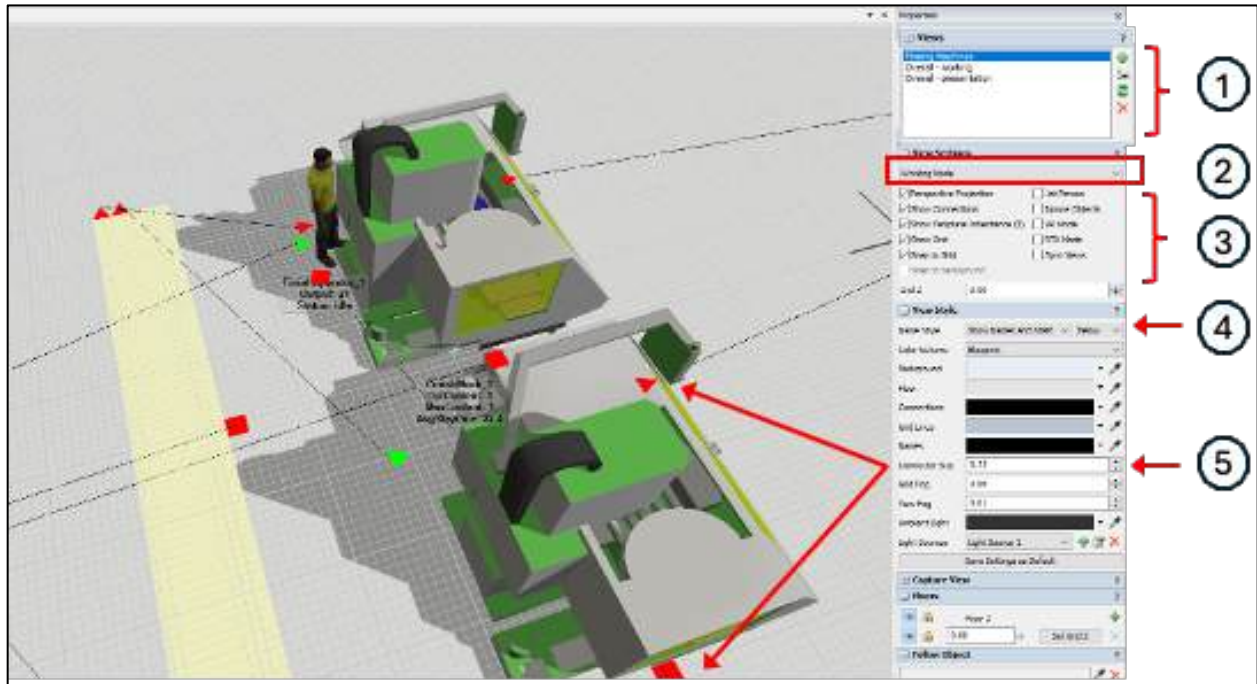
Note that if the connections are reversed, the model would behave differently. If the **Source** and **Sink** are connected through Port 1 and the **Source** and **Queue** through Port 2, then all items would go to the **Sink**, and none to the **Queue** since a **Sink** has no capacity limits. Therefore, it is important to be careful about the order in which objects are connected.



If you haven't already done so, save the model. Recall that it is good practice to save often.

11.7 Creating model views

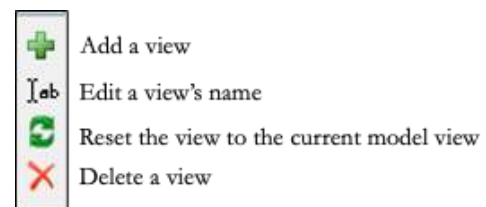
When the modeling surface is clicked and not an object, the **Properties** window controls what is displayed on it. The figure below highlights a few controls associated with the model view. The numbers in the figure refer to the sections below where the item is discussed.



1. This area of the figure shows three model views that have been created. The first one, the one shown as selected, is the view in the figure above – it is a closeup of the area near the finishing machines.

Clicking on a View easily switches the model's display to predefined views, such as an overall view of the model, a close view of a particular object, etc.

The buttons shown in the figure to the right, along with their definition, are used to create and edit the model views.



2. This dropdown menu switches from *Working Mode* to *Presentation Mode*. The former is what is shown in the figure above since it shows all of the ports, connections, the grid, etc. The latter removes the ports, connections, grid, etc., making the view less cluttered and more amenable to presentation. An example is shown in the figure below.
3. These items allow the user to toggle various aspects of the view on and off by checking or unchecking the item, such as perspective projection, connections, grid, etc.

4. Three **Name Style** display options are available: *Show Names and Stats*, *Show Names*, and *Show Nothing*. These options control what information is displayed for each object in the model — name and a few statistics, name only, or nothing.
5. The only other **View Style** option that is highlighted in the figure above is **Connector Size**. This controls the size of the port connections.
 - The current value of 0.25 is a bit large, so set it to 0.15.

The figure below is the *Overall presentation View*. It shows the overall model with all the connectors, ports, grid, etc. removed.



If you haven't already done so, save the model. Recall that it is good practice to save often.



Use the **Save Model As** option in the **File** menu to make a copy of the existing model so that it can be customized beginning in the next section. Again, you can use any file name, but in the primer, the next model is referred to as **Primer_4**.

12 EMPIRICAL DISTRIBUTION AND DISTRIBUTION FITTING

Chapter 12 uses the Empirical Distribution tool to define the product mix of containers entering the Finishing Area. It also uses the Empirical Distribution tool to fit system data to a probability distribution that best describes the arrival process of containers to the Finishing Area.

The base model for the additions described in this chapter is **Primer_3** that was saved at the end of Chapter 11. However, a copy of that file was saved as **Primer_4**; thus, we begin with that file.

The primer now switches from *FlexSim* modeling *objects* to *FlexSim* modeling *tools*. As the name suggests, tools are means to support various aspects of simulation modeling and analysis.

Tools are accessed through the **Toolbox** Library. As shown by the red boxes in the figure to the right, the **Toolbox** Library is accessed either through

- (1) the tab next to the **Objects** Library or
- (2) the **Tools** button on the **Main Menu** bar.



Also, shown in the figure to the right, tools are added to a model through the  button in the upper left portion of the interface.

The first of many *FlexSim* tools discussed in the primer is the **Empirical Distribution**, which has several roles. The first is specifying an empirical distribution, one that represents sample data from a system and not a theoretical probability distribution such as the normal, exponential, binomial, etc. The second role is as a distribution fitter – using sample data from a system or a simulation model to represent that data more generally as a theoretical probability distribution. Each is explored in the sections below.

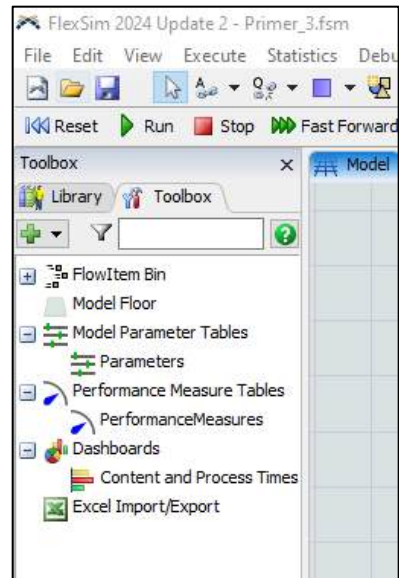
12.1 Using an Empirical Distribution for product mix

The **Empirical Distribution** tool can be used to specify continuous or discrete distributions, and the data values can be weighted or not.

As assumed so far, the product mix of containers is unlikely to be uniformly distributed, i.e., each type is equally likely. Therefore, we'll use the **Empirical Distribution** tool to specify the planned product mix.

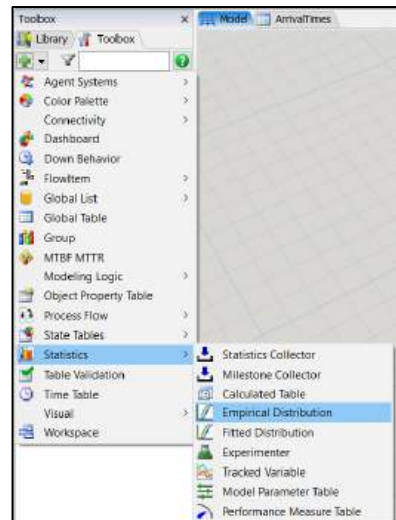
Since this is the first time the **Toolbox** has been used, the figure to the right shows the default contents of the **Toolbox** tab.

- To access the **Empirical Distribution**, first select the  button that is located below the **Library** tab, as shown in the figure to the right.



Selecting the  button results in the dropdown menu shown in the figure to the right.

- Select the *Statistics* option, then select the *Empirical Distribution* option, as shown in the figure to the right.



Update the default Empirical Distribution Properties to match the figure to the right and based on the instructions below.

Empirical Distribution Properties

ProductMix

Data

Data Type: Number ☒ Weighted

Rows: 3

	Data	Weight
1	1.00	0.50
2	2.00	0.30
3	3.00	0.20

Sample Generation

Distribution Type: Discrete Empirical

Generate Samples

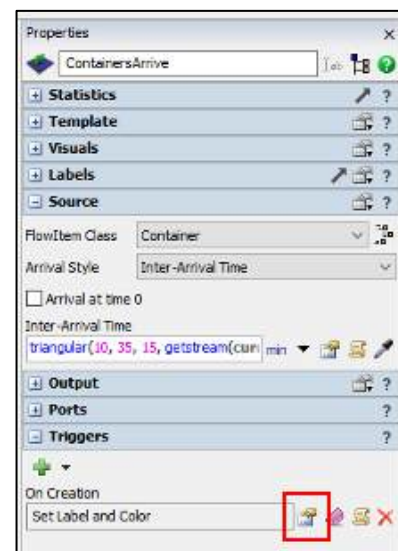
- Change the name from *EmpiricalDistribution1* to something more meaningful, such as *ProductMix*.
- Select the **Weighted** checkbox to create a second column in the data table called **Weight**.
- Increase the **Rows** to 3.

- Change the **Distribution Type** from *Continuous Empirical* to *Discrete Empirical*.
- Enter the data in the table as shown in the figure above. The data is interpreted as follows. When needed at the **Source**, the product type is randomly selected from this discrete distribution. The type will take on one of three values – 1, 2, 3 – with probabilities of 0.5, 0.3, 0.2, respectively.

As more product types are added to the model, this table needs to be updated — the number of rows increased, and the new product types and their probabilities need to be specified. The existing probabilities need to be updated since **the total of the Weight column must always be exactly 1.00**.

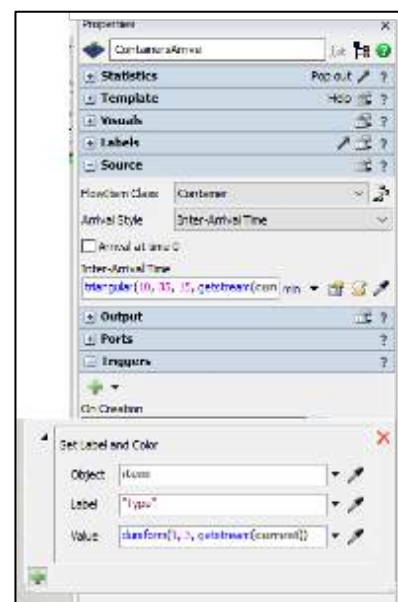
The **Source** now needs to be updated so that it uses the **Empirical Distribution** and not the **Discrete Uniform Distribution** to randomly assign product types. This is done as follows.

- In the **Source** object (*ContainersArrive*), open the **Triggers** pane and press the view button, as highlighted by the red box in the figure to the right, for the *On Creation – Set Label and Color* trigger.



The resulting interface is shown in the figure to the right.

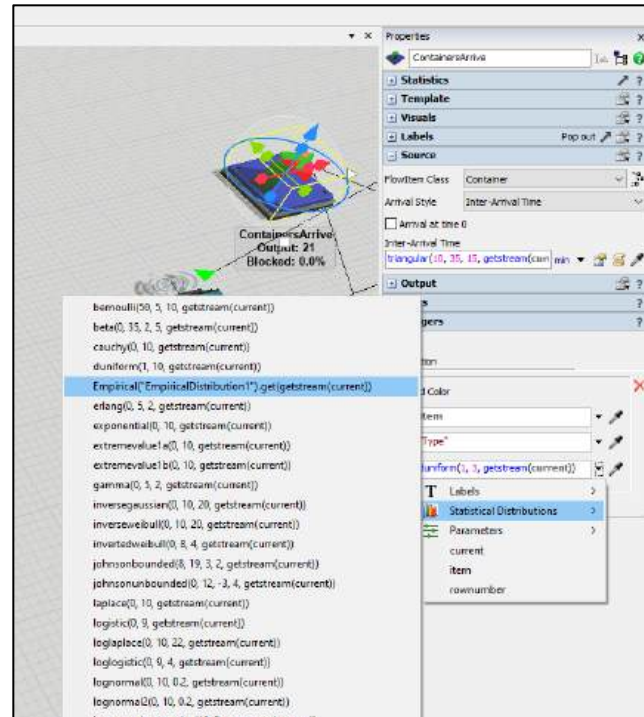
- Select the dropdown menu for the **Value** property.



The resulting interface is shown in the figure to the right.

- Select the **Statistical Distributions** option from the dropdown list, then select *Empirical*("EmpiricalDistribution1")
.get(getstream(current))
- In the **Value** field, change *EmpiricalDistribution1* to *ProductMix*, the name assigned to the empirical distribution. **Be sure to edit the name correctly and keep the quotation marks.** The **Value** field should read as

Empirical("ProductMix").get(getstream(current))

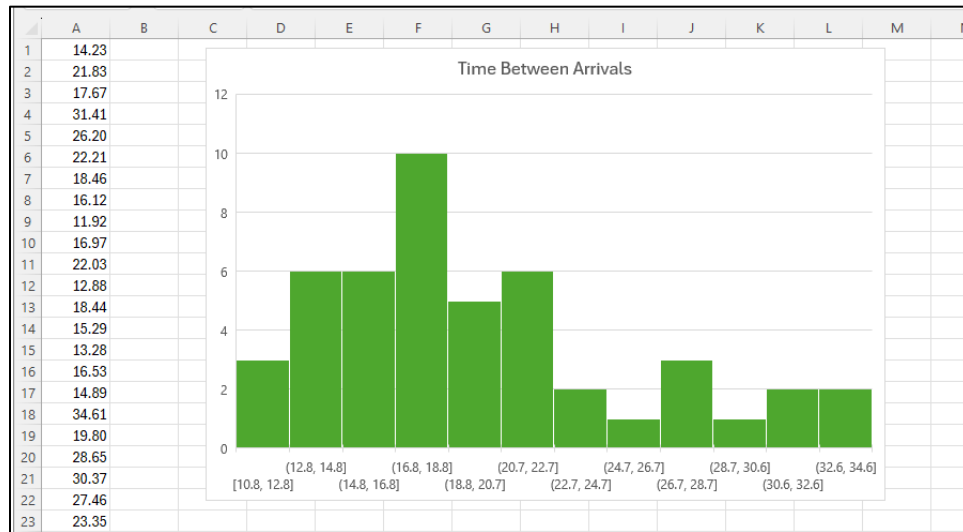


12.2 Using an Empirical Distribution for distribution fitting

The **Empirical Distribution** tool is also used to help select the probability distribution to use to represent system data in a simulation model.

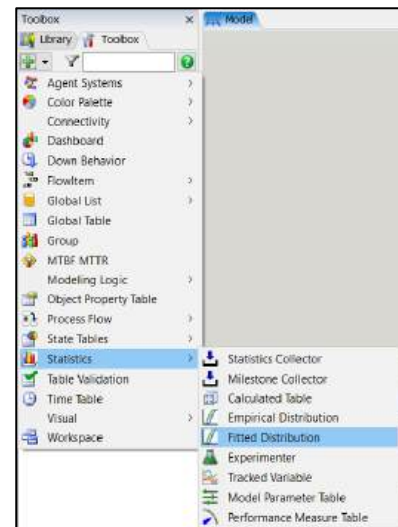
Another DPL facility ran a small batch of test containers. They were produced at the planned production rate that would support two Finishing Machines. During the test, they collected data on the time between arrivals to the finishing area.

They produced 125 containers but discarded the data for the first 25 arrivals to any remove startup effects. The remaining 100 times between arrivals are provided in the *Microsoft Excel* file named *TimeBetweenArrivals*, which is provided in the Resources folder. The file also contains a histogram of the data. A portion of the file is shown in the figure below.

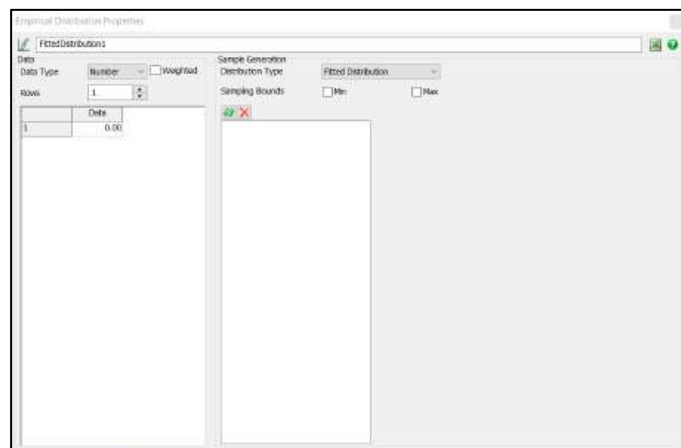


To fit this data to various probability distributions and decide which distribution best fits the data, create an instance of the **Empirical Distribution** tool starting in the same manner as in the previous section.

- Select the  button located below the **Toolbox** tab. Select the *Statistics* option from the dropdown menu, and then select the *Fitted Distribution* option, as shown in the figure to the right.



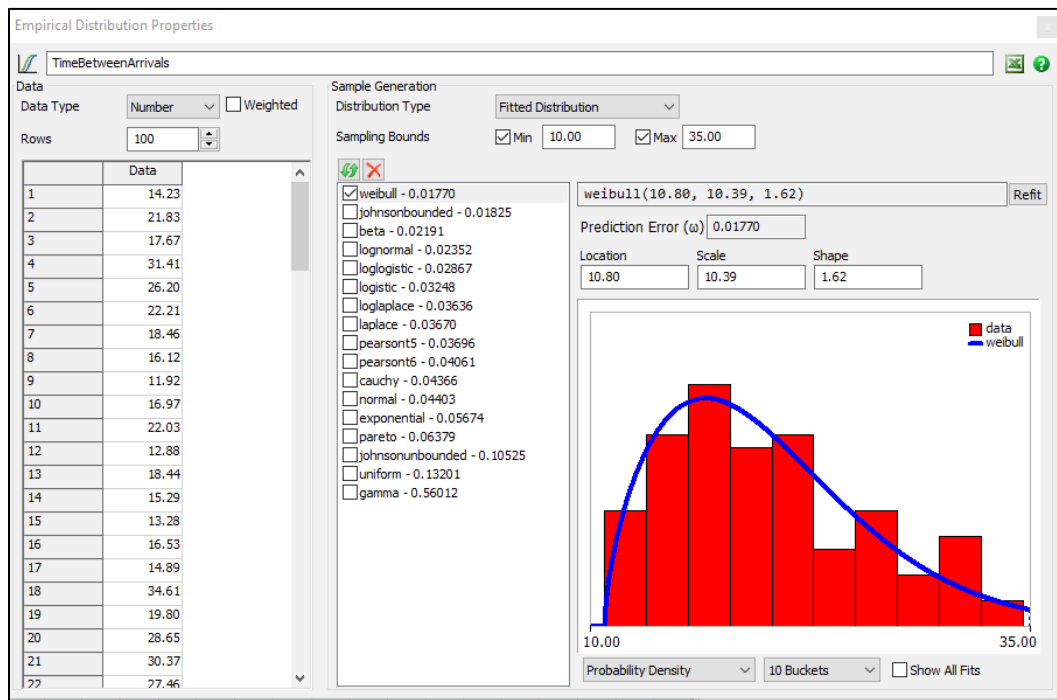
The resulting Empirical Distribution interface is shown in the figure to the right.



Update this interface based on the instructions below.

- Change the name from *FittedDistribution1* to something more meaningful, such as *TimeBetweenArrivals*.
- Increase the **Rows** to 100.
- Select and copy (Ctrl-C) the 100 values from the *Excel* file.
- Select the 100 cells in the *Data* column of the **Empirical Distribution** table.
- Paste (Ctrl-V) the data into the *Data* column.

- Under the **Sample Generation** section, press the  button to fit the distributions. This change should result in the interface shown in the figure below.



The best-fitted distribution is considered to be the Weibull distribution. It is at the top of the rank-ordered list of distributions and has the minimum Prediction Error (0.01770). The probability density function for the Weibull is plotted as the blue line that overlays the histogram of collected data. The next-best distribution in terms of Prediction Error is the Johnson Bounded. There is very little difference between the Prediction Error for this and the Weibull, so either could be used.

The following are a few comments on modifying the display above.

- To display a distribution and its fit to the data, check the box that precedes its name in the list.
- To overlay multiple distribution fits, check the **Show All Fits** box below the plot.
- To delete any distribution from considerations, check the box to the left of each distribution name and then press the  button. For example, if you want to overlay plots of the top three distributions, delete all distributions except the top three, then check the **Show All Fits** box below the plot.
- If any changes are made, use the  button to refresh list.

The checked distribution is the one that gets implemented as the empirical distribution in the model. The selected function and its parameters are shown above the figure and next to the **Refit** button.

Recall that the engineers and domain experts assumed the arrival process for the containers from the upstream operation would be triangularly distributed with a minimum of 10 minutes, a maximum of 35 minutes, and a most likely value of 15 minutes. Data from the sample production run is very close to these estimates. Therefore, the Triangular distribution could be used as well. Depending on the stakeholders in the project, one

advantage of using the Triangular distribution is that it might be easier to explain to a layperson than the Weibull. However, the Weibull is a commonly used probability distribution in many engineering disciplines.

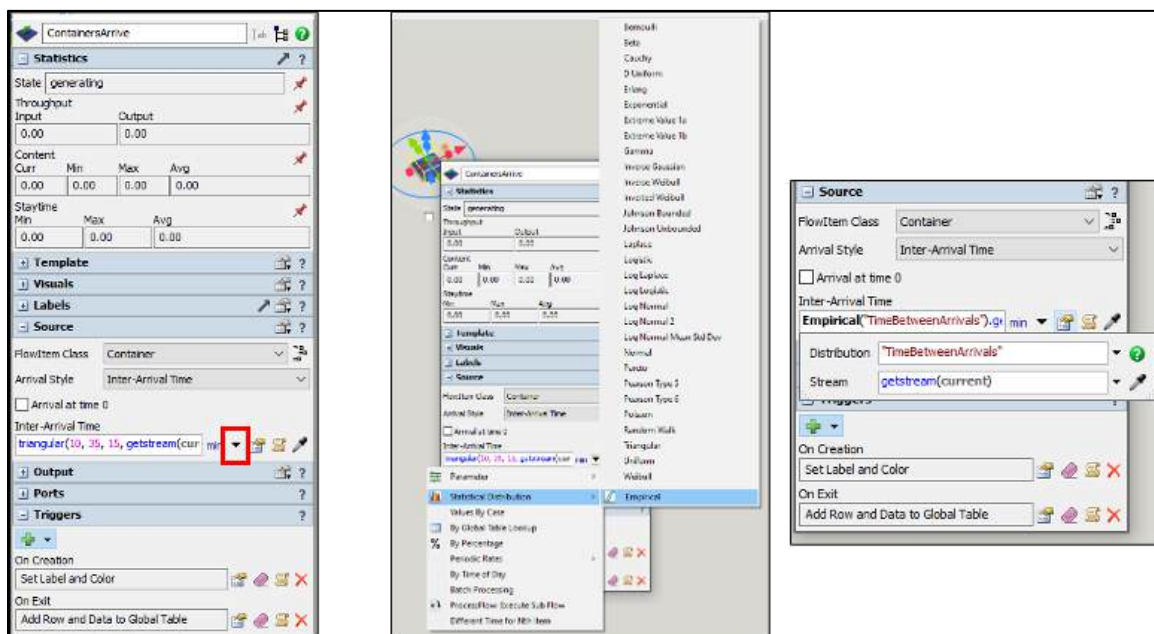
- Since the Weibull can be unbounded, and the minimum and maximum interarrival times were considered to be 10 and 35, respectively, check the *Min* and *Max* boxes to the right of the **Sampling Bounds**, as shown in the figure above.

By checking these boxes and specifying the extreme values, *FlexSim* will sample from the specified Weibull distribution, and if a sampled value is outside of the designated range, *FlexSim* will resample. Therefore, by using the Empirical Distribution, we are sure all of the interarrival times will be between 10 and 35.

Now, the **Source** needs to be updated so that it samples the containers' interarrival times from the specified Weibull distribution and not the triangular as first assumed.

In the **Source** *ContainersArrive*,

- Select the dropdown menu button to the right of the **Inter-Arrival Time** textbox, as highlighted by the red box in the first pane in the figure below.
- Select **Statistical Distribution**, then select **Empirical** from the resulting dropdown menu button, as shown in the middle pane in the figure below.
- Finally, select the *TimeBetweenArrivals* **Empirical Distribution**, as shown in the right panel in the figure.



If you haven't already done so, save the model. Recall that it is good practice to save often.



Use the **Save Model As** option in the **File** menu to make a copy of the existing model so that it can be customized beginning in the next section. Again, you can use any file name, but in the primer, the next model is referred to as Primer_4A.

13 MODEL PARAMETER & GLOBAL TABLES

Chapter 13 introduces two types of data tables that help organize information used in *FlexSim* models: Model Parameter Tables and Global Tables.

We now consider a few more modeling tools that are available in *FlexSim*. Both are tables, which provide convenient means for storing information. During a simulation, an aspect of the model can reference a table cell to obtain a parameter value.

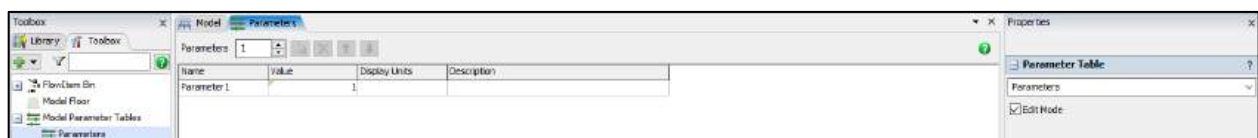
It is good modeling practice to separate data (property values) from objects, especially if the values are expected to be changed, such as when conducting what-if analyses and experimentation.

Parameter Tables contain values that are likely to be changed during experimentation with the **Experimenter** or in an optimization (*FlexSim* links with *OptQuest* to search for an optimal solution.) **Global Tables** are more general, resemble spreadsheets, and can be used to store input information or output from a simulation. *FlexSim* provides a convenient link to *MS Excel* so that data can be easily exchanged (imported or exported) between a model and *MS Excel* via Global Tables.

The base model for the additions described in this chapter is **Primer_4** that was saved at the end of Chapter 11. However, a copy of that file was saved as **Primer_4A**; thus, we begin with that file.

13.1 Model Parameter Tables

As shown in the figure below, a **Model Parameter Table** named *Parameters* is created by default in the **Model Parameter Tables** section of the **Toolbox**. For brevity, these tables are referred to as **Parameters Tables**; a model may include multiple **Parameter Tables**.



Select the *Parameters* table in the **Toolbox** and edit it as described below. It will contain the average process times at the Finishing Machines for each container type.

- In the text box in the **Properties** window, rename the table from *Parameters* to *FinishTimes*.
- Increase the number of parameters to 3.
- Rename the parameters from *Parameter1*, *Parameter2*, and *Parameter3* to *FinishTime_1*, *FinishTime_2*, and *FinishTime_3*, respectively.

- Click on the **Value** cell for *FinishTime_1*. The default settings are shown in the figure to the right.

Name	Value	Display Units	Description
FinishTime_1	1		
FinishTime_2			
FinishTime_3			

Type	Continuous
Lower Bound	1
Upper Bound	10
Reference	None
On Set	

- The options for the data type of the value are shown in the figure to the right. Note the other types that may be declared, but leave the default value of *Continuous* since the process times are continuous values, i.e., not integer (1, 2, 3, ..), binary (0,1), etc.

Value	Display Units	Description
1		

Type	Continuous
Lower Bound	
Upper Bound	
Reference	
On Set	

- Continuous
- Integer
- Discrete
- Binary
- Option
- Sequence
- Expression
- Pass-through
- Custom

- As shown in the figure to the right, for *FinishTime_1*, enter the **Lower Bound** as 10 and the **Upper Bound** as 20.
- Set the value of **Value** to be 15.

The finishing machine's process time for container type 1 will be 15 minutes, but it could be set as low as 10 and as high as 20. If a value outside this range is entered, an error message will indicate the specified value is out of bounds.

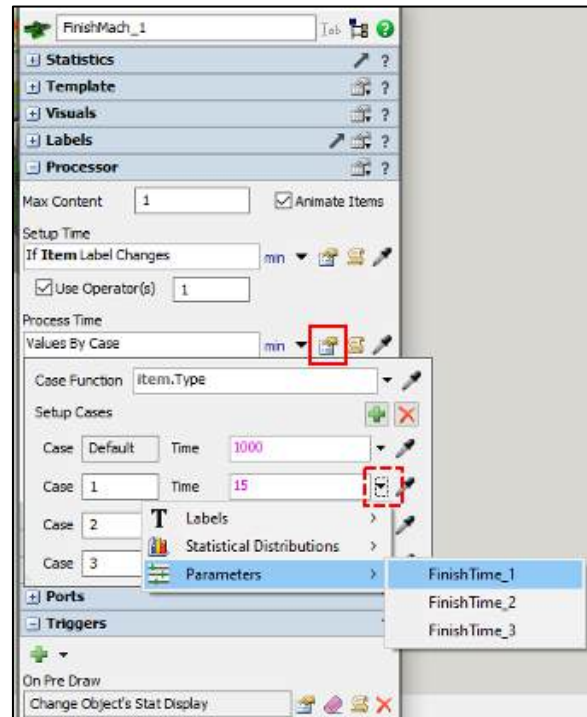
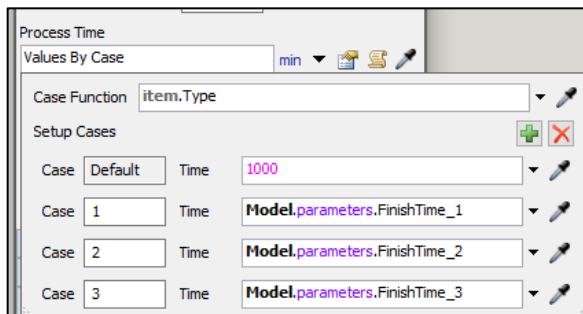
Name	Value	Display Units	Description
FinishTime_1	15		
FinishTime_2			
FinishTime_3			

Type	Continuous
Lower Bound	10
Upper Bound	20
Reference	None
On Set	

- For *FinishTime_2*, enter the **Lower Bound** as 15 and the **Upper Bound** as 25.
- Set the value of **Value** to be 20.
- For *FinishTime_3*, enter the **Lower Bound** as 20 and the **Upper Bound** as 40.
- Set the value of **Value** to be 30.

Link the parameters to the process times in the Processors. On **each** Processor:

- As shown in the figure to the right, in the **Processor** pane, select the edit button next to the *Values By Case* option of the **Process Time** property. A red square in the figure highlights the edit button.
- For **each** case, select the correct **Parameters** value, e.g., *FinishTime_1* for **Case 1**. The completed interface should look like the figure below.



Again, repeat this *FinishMachine_2*.

Model.parameters.FinishTime_1 is a *FlexSim* command that references the current value of the *FinishTime_1* **Model Parameter**.

13.2 Global Table to store arrival times

Global tables are used to store many types of information. In this case, a table will be used to store the time and type of each container as it arrives to the Finishing Area. This data may be used to verify the arrival process.

The table begins blank, i.e., it contains no information, but when an arrival occurs, a new row is added, and its time and type are written in the cells under the appropriate columns.

- Click the  button on the **Toolbox** and select **Global Table** from the dropdown list.

Change the table's **Properties** to those shown in the figure below.

- Change the name from *GlobalTable1* to *ArrivalTimes*.
- Ensure the value of **Rows** is 0 and **Columns** is 2.
- Click the edit button for the **On Reset** property and select *Delete All Rows*. The data in the table will be removed whenever the model is reset.

- Change shaded “headers” for the columns from the default *Col 1* and *Col 2* to more descriptive names, e.g., *ArrivalTime* and *Type*, respectively.

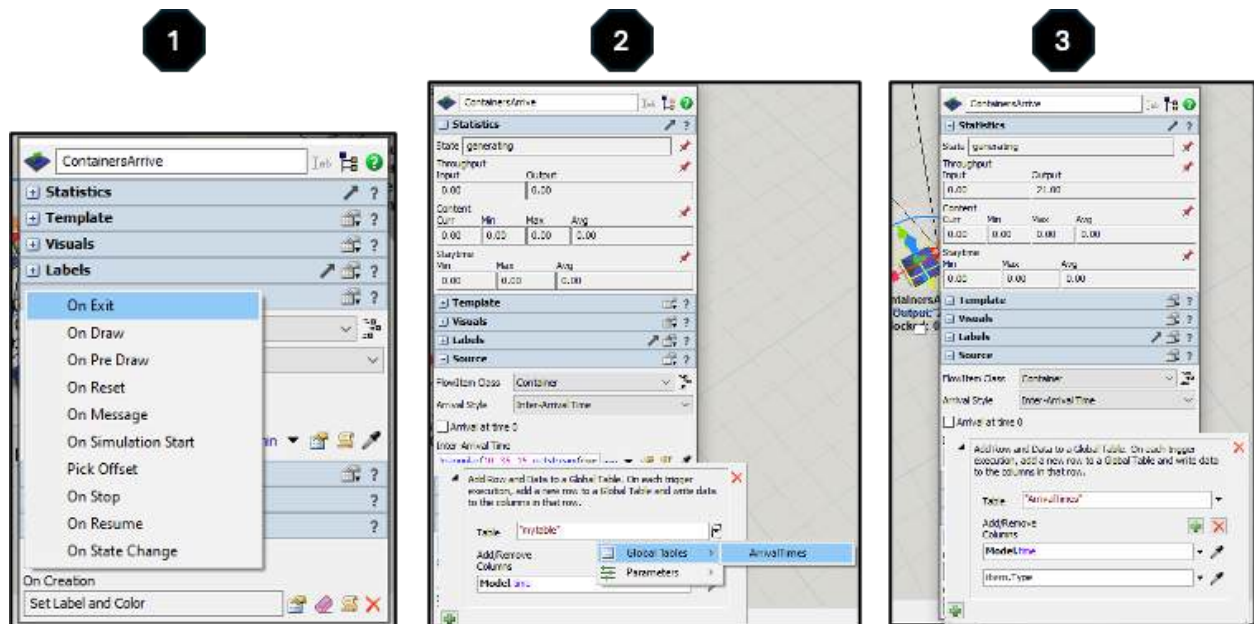
Note that the headings are just text for information purposes and are not table values. However, they are very helpful in understanding what data tables contain.

Also, the row heights and column widths can be changed by dragging the lines dividing the cells.



The figure above shows that tables open in a new tabbed window, like the model. The view may be closed by pressing the **X** in the upper-right corner of the interface. The window can be reopened by double-clicking the table in the **Toolbox**.

- As shown in Panel 1 in the figure below, use the  button in the **Triggers** pane of the **Source** object *Containers.Arrive* to create an **On Exit** trigger.



- Using the  button next to the text box of the **On Exit** trigger, select **Data**, then select *Add Row and Data to Global Table*. The interface should look like Panel 2 in the figure above.

- Also, as shown in Panel 2, for the **Table** property, use the dropdown button next to “mytable” to select **Global Tables** then *ArrivalTimes*, the table just created above

Note that if you create a table before using it in the model, the table will appear in a dropdown menu list, and thus, you do not have to type in the table name. The name must be typed exactly as it is in the **Global Table** – names are case-sensitive in *FlexSim*. Therefore, it is better to select the name from the dropdown menu.

The first column will contain the arrival time; it is determined by the *FlexSim* command **Model.time**.

- As shown in Panel 3 in the figure above, add the data to be written in Column 2. Use the  button to the right of **Add/Remove Columns**, which creates a new column with a default value of 10.
- Use the dropdown button to select *item*, then, as shown in Panel 3 in the figure above, change the entry to *item.Type*. This means the value of the current item’s label named **Type** will be displayed in the column cell.

- **Reset** and **Run** the model. Verify that each container's arrival time and type are written to the Global Table *ArrivalTimes*.

The figure to the right shows a partial list of each container’s arrival time and its type.



If you haven’t already done so, save the model. Recall that it is good practice to save often.



Use the **Save Model As** option in the **File** menu to make a copy of the existing model so that it can be customized beginning in the next section. Again, you can use any file name, but in the primer, the next model is referred to as *Primer_5*.

Model		ArrivalTimes	
	ArrivalTime	Type	
	11.83	1	
	23.43	3	
	45.16	1	
	61.76	2	
	82.15	3	
	111.50	2	
	127.37	1	
	145.41	1	
	171.09	3	
	195.76	1	
	212.53	1	
	235.42	1	
	248.95	1	
	262.40	2	
	284.95	1	
	297.63	1	
	311.17	3	
	335.75	3	
	361.81	2	
	377.52	1	
	393.01	2	

14 DOWNTIME

Chapter 14 adds several types of downtime to the model. Break and lunch times are added for the Finishing Operator, i.e., when the operator is unavailable to do system work. Two types of downtimes are considered for the Finishing Machines. The first is a quality check that occurs on a fixed clock-based schedule; the second is randomly occurring breakdowns that are based on the object's state. Machine breakdowns require a Finishing Operator for repairs; the quality checks do not require an operator. Pie charts are introduced to summarize the utilization and percentage of time the Finishing Machines are in various states. The charts are added to a new dashboard.

In *FlexSim*, there are two means for making resources unavailable, i.e., incurring downtime – the **Time Table** tool and the **MTBF/MTTR** tool. Both are accessed through the **Toolbox** Library. A powerful feature of *FlexSim* is the management of downtimes, especially multiple types of downtimes. As such:

- Any object can be subjected to downtime.
- Any object can be subjected to multiple types of downtimes, oftentimes referred to as *competing downtimes*.
- Multiple objects can follow the same downtime process.

The base model for the additions described in this chapter is **Primer_4A** that was saved at the end of Chapter 13. However, a copy of that file was saved as **Primer_5**; thus, we begin with that file.

14.1 Time Tables

Planned downtimes - ones that occur on a known and recurring basis, such as breaks and shift schedules - are modeled using Time Tables.

In this example, the Finishing Operator takes a 15-minute break two hours into an 8-hour shift, a 30-minute lunch break four hours into the shift, and another 15-minute break six hours into the shift. This is implemented via the **Time Table** tool and its three tabs – *Members*, *Functions*, and *Table*. The property settings for this tool are explained below.

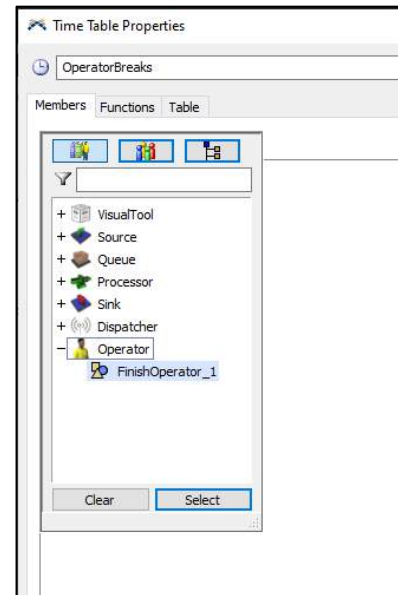
- In the **Toolbox**, press the  button and select **Time Table** from the drop-down menu.

As shown in the figure to the right, on the **Time Time** interface:

- Change the name from *TimeTable1* to *OperatorBreaks*.

The **Members** tab is used to assign this downtime pattern to one or more objects.

- Using the  button, select *FinishOperator_1* from the **Operator** object category and press the **Select** button. If you choose a category, such as **Operators**, the timetable will apply to all objects in that category.

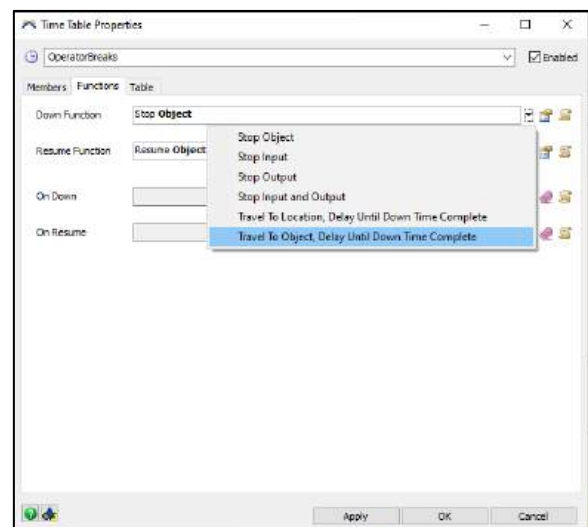


The **Functions** tab controls what happens when a resource goes down, i.e., becomes unavailable and then resumes. By default, *FlexSim* stops the object at the appropriate time in the simulation, and the object remains down for the prescribed time. In this example, the following behavior is added to the default.

On the **Functions** tab, the default **Down Function** value, *Stop Object*, stops the resource wherever it is and whatever it is doing, and it remains at that location for the prescribed duration. However, in this example, the **Operator** takes a break at a specified location, at the **Dispatcher** (*FinishOperators*) object. The prescribed delay starts when the **Operator** reaches the specified location.

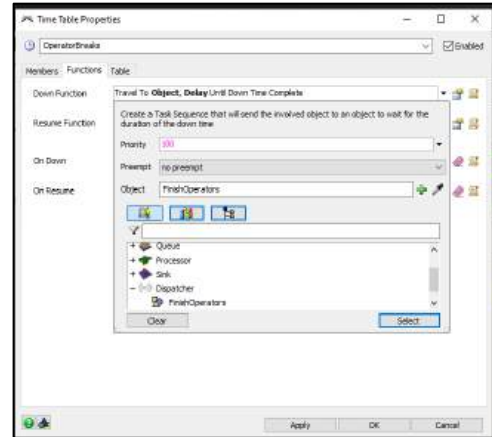
To implement this, complete the following steps.

- As shown in the figure to the right, change the **Down Function** property from *Stop Object* to *Travel To Object, Delay Until Down Time Complete* by selecting the option from the drop-down menu.



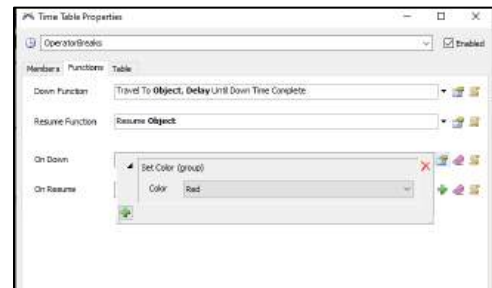
- In the resulting dialog, set the **Object** property to *FinishOperators* using the  button and select the object in the **Dispatcher** category.

Retain the default values for the **Priority** and **Preempt** properties.



To provide a visual cue that the operator is not available, change the object's color during the break period.

- As shown in the figure to the right, change the **On Down** option to **Set Color (Group)** and the **Color** property value to **Red** (the default). This changes the operator's color to red when on break.

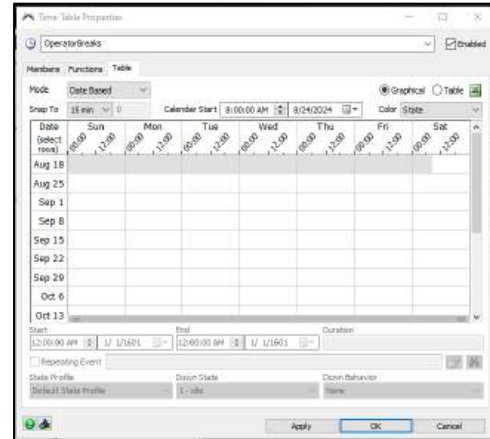


- Similarly, reset the operator's color after the downtime is concluded. Change the **On Resume** property to **Set Color (Group)** and then the **Color** property to **Yellow** from the drop-down list of colors. This will restore the operator's color to its default color, yellow, when the operator is available for work.

The **Table** tab is used to enter the downtime information. The figure to the right is the default interface, referred to as the *Graphical* interface.

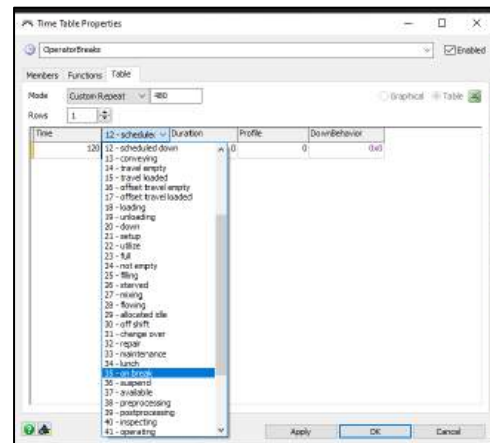
However, we will use the simpler *Table* interface.

- On the upper right portion of the interface, change the selection from Graphical to Table.



The Table interface is shown in the figure to the right.

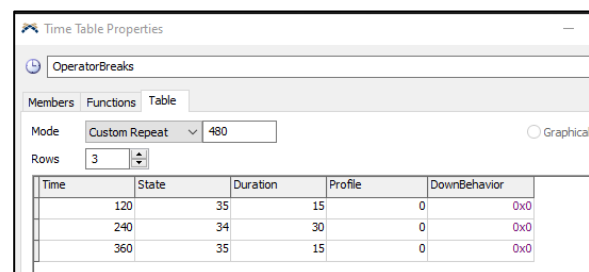
- Change the **Mode** from *Daily Repeat* to *Custom Repeat*.
 - Change the repeat value, which is just to the right of the **Mode** value, to 480. This sets the break pattern defined below to repeat every shift or every eight hours (480 minutes).
- This is handy when simulating multiple shifts, as the downtimes for each shift do not have to be explicitly included in the timetable; the provided pattern just repeats for as long as the simulation runs.



- The first row will be for the first break, two hours into the shift. Therefore, enter 120 in the **Time** column.
- The next column defines the **State** the resource is in when on break. Use the dropdown menu as shown in the figure above to select state 35 – on break. There are 50 possible states defined in *FlexSim*. The **State** property is used to calculate the resource’s utilization.
- Set the **Duration** of the break to 15.
- Keep the default values for the **Profile** and **DownBehavior** columns.
- Increase the number of **Rows** from the default 1 to 3 so the other breaks can be entered.

As shown in the figure to the right, enter the information for the other two break periods.

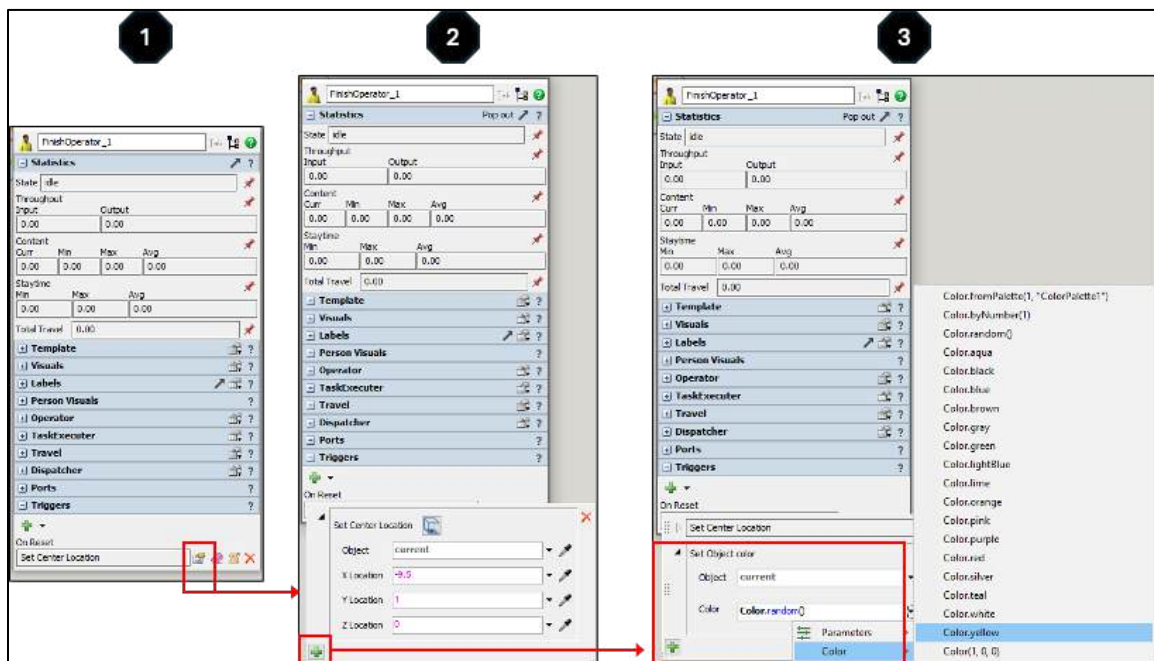
- The second downtime is a 30-minute lunch break that occurs four hours into the shift; therefore, set **Time** to 240 minutes and **Duration** to 30 for the second row. Set the **State** to 34 – lunch.
- The third downtime is another 15-minute break that occurs six hours into the shift; therefore, set **Time** to 360 minutes and **Duration** to 15 for the third row. Set the **State** to 35 – on break.



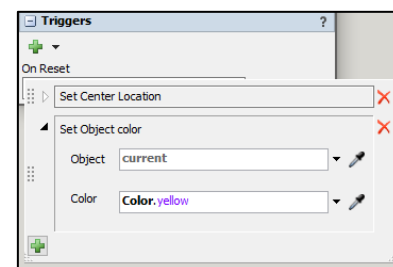
Since the **Operator**'s color is set to red when on a break, its color is not set back to its normal color if the simulation ends while the **Operator** is on break. To avoid this, add an **OnReset** trigger to the **Operator** that sets its color to yellow whenever the model resets.

Recall the **Operator**, *FinishOperator_1*, already has an **On Reset** trigger that places the object at a set location when a simulation begins. Therefore, we need to add another **On Reset** trigger.

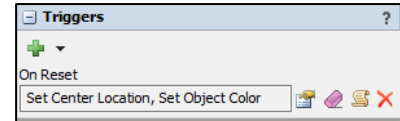
- As shown in the first panel in the figure below and highlighted by a red box, select the **Edit** button to the right of the **On Reset** trigger.
- This opens the interface for defining the *Set Center Location* **On Reset** trigger, as shown in the second panel in the figure below. Click the  button to open a list of possible triggers; select *Set Object Color*.
- This opens the interface to define the *Set Object Color* trigger, as shown in the third panel in the figure below.
- Change the **Object** value from *item* to *current*. *Current* refers to the current object we are working on, which in this case is the **Operator**.
- Change the Color from *Color.random()* to *Color.yellow* using the picklist, as shown in the last panel of the figure below.



The interface should now look like the one in the figure to the right.



Similarly, the **Triggers** pane on the **Properties** window should look like the figure to the right. Note that it indicates there are two **On Reset** triggers, one to set the object's location and one to set the object's color.



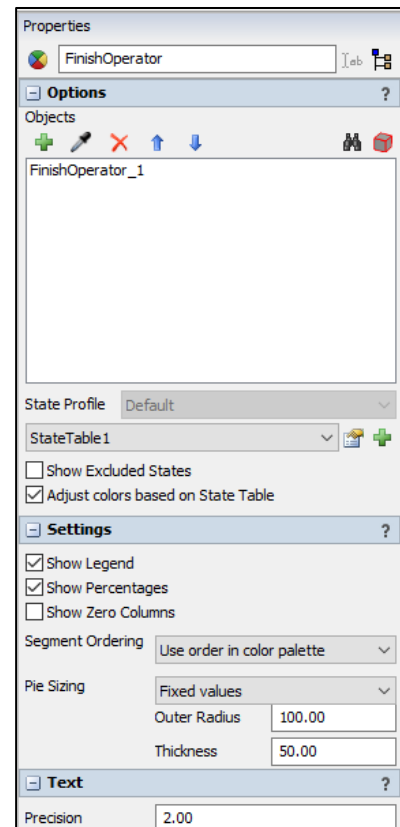
14.2 Chart of Operator utilization and states

To see the effects of downtime, create a new dashboard that shows the states of the **Operator**.

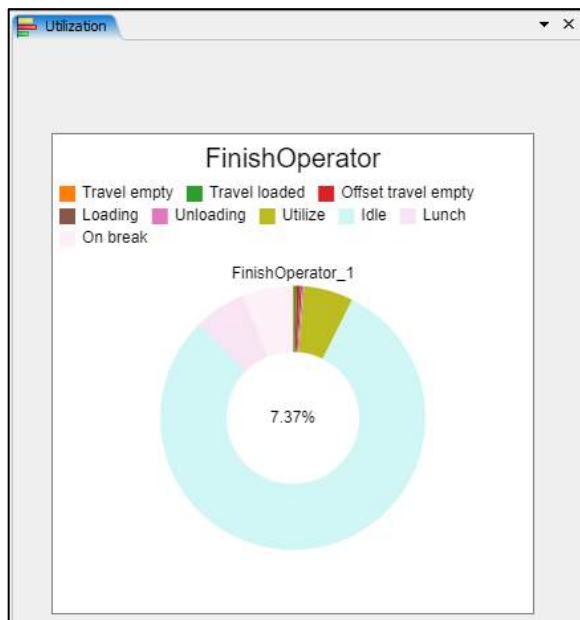
- Create a new dashboard by clicking the **Dashboards** button on the **Main Menu** or the  button in the **Toolbox**.
- Change the name from *Dashboard* to *Utilization*.
- Drag a *Pie Chart* from *State* option in the **State** pane in the **Dashboard** library to the Utilization workspace.

As shown in the figure to the right

- Change the name of the chart from *State* to *FinishOperator*.
- Use the  button in the **Objects** section to select the Operator.
- In the Settings pane, change the size of the pie chart
 - Set **Outer Radius** to 100.0
 - Set **Thickness** to 50.0.



If the model is reset and run, the chart should resemble the one below.



Notice all of the states the operator is in during the simulation – *Travel empty*, *Travel loaded*, etc. The **Operator** spends most of the time in the *Idle* state since it is utilized only 7.37% of the time. After *Idle*, the three next most occupied states are *On break*, *Lunch*, and *Utilize* (performing the setup operation on the **Processors**).

The states that contribute to the utilization calculation are shown in a **State Table**. The default table in *FlexSim* is shown in the figure to the right. Custom tables can be created if needed. Percent utilization is calculated as follows.

$$\text{Utilization} = 100 * (\text{total time in utilized states}) / (\text{total time})$$

The Analysis column in the table shows how each state contributes to an object's utilization. All states with the green *Utilize* indicator contribute to the numerator in the formula above. All states with the green *Utilize* indicator plus those with a blank in the Analysis column, i.e., no indicator, contribute to the denominator in the formula above. The utilization calculation does not count any states with a red *Excluded* indicator.

The Analysis value for each state can be toggled between *Utilize*, blank, and *Excluded*.

State Tables can be accessed and created via the **Toolbox**.

- **Reset** and **Run** the model. Verify that the **Operator** travels to the **Dispatcher** at the prescribed times (120, 240, 360, 600, etc.), that its shirt turns red while on break, and that it reverts to yellow when not on break.

In the figure below, the operator is on a 30-minute lunch break at the **Dispatcher's** location. Note that an easy way to stop a model at predetermined times is to set multiple stop times, as shown in the figure below.

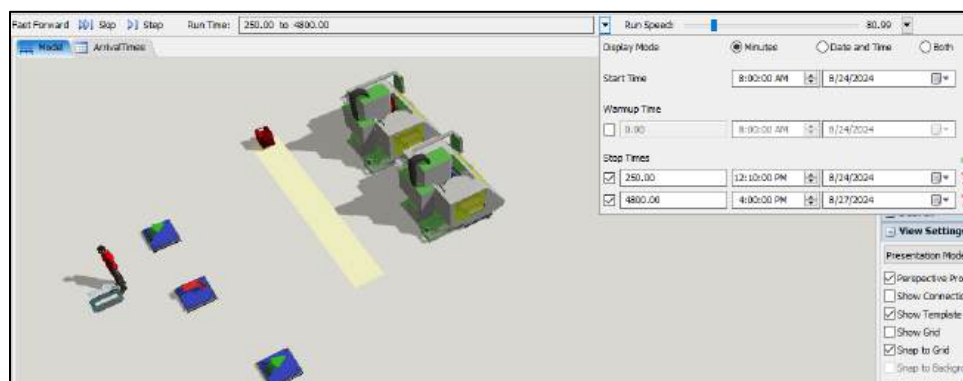
Stop times are added by clicking the  button, removed by clicking the  button, and are made temporarily inactive by unchecking the box in front of the time text box.

State Table Properties

StateTable1

Auto-fill from state profile

State	Display Name	Analysis
1 Idle		
2 Processing		Utilized
3 Busy		Utilized
4 Blocked		
5 Generating		Utilized
6 Empty		
7 Collecting		Utilized
8 Releasing		Utilized
9 Waiting for operator		
10 Waiting for transporter		
11 Breakdown		
12 Scheduled down		Excluded
13 Conveying		Utilized
14 Travel empty		Utilized
15 Travel loaded		Utilized
16 Offset travel empty		Utilized
17 Offset travel loaded		Utilized
18 Loading		Utilized
19 Unloading		Utilized
20 Down		
21 Setup		Utilized
22 Utilize		Utilized
23 Full		Utilized
24 Not empty		Utilized
25 Filling		Utilized
26 Starved		
27 Mixing		Utilized
28 Flowing		Utilized
29 Allocated idle		Utilized
30 Off shift		Excluded
31 Change over		Utilized
32 Repair		
33 Maintenance		
34 Lunch		
35 On break		
36 Suspend		
37 Available		
38 Preprocessing		Utilized
39 Postprocessing		Utilized
40 Inspecting		Utilized
41 Operating		Utilized
42 Standby		
43 Purging		Utilized
44 Cleaning		Utilized
45 Accelerating		Utilized
46 Maxspeed		Utilized
47 Decelerating		Utilized
48 Stopped		
49 Waiting		
50 Accumulating		Utilized





If you haven't already done so, save the model. Recall that it is good practice to save often.



Use the **Save Model As** option in the **File** menu to make a copy of the existing model so that it can be customized beginning in the next section. Again, you can use any file name, but the next model is referred to as **Primer_6** in the primer.

The base model for the additions described in this part of the chapter is **Primer_5** that was saved above. However, a copy of that file was saved as **Primer_6**; thus, we begin with that file.

14.3 Reliability

Reliability refers to an object being able to perform its required functions for a specified time. It is an important contributor to overall system performance. Two key factors are used to specify reliability: *Mean Time Between Failures* (MTBF) and *Mean Time To Repair* (MTTR). The former is the uptime or time an object operates, expressed as an average or mean value; the latter is the time it takes to get a down object back to its operational state, again expressed as an average or mean value.

A discussion of reliability is beyond the scope of this primer. However, the topic is introduced in the *Applied Simulation Modeling and Analysis Using FlexSim* textbook in Sections 7.1-7.5.

In this example, each Finishing Machine is subject to two types of downtimes. Thus, the model has competing downtimes that must be managed, which *FlexSim* does very well. For example, in most cases, if a resource is already down or not available, it cannot incur another type of downtime.

14.4 Constant MTTF/MTTR; MTBF based on clock time; no resource for repair

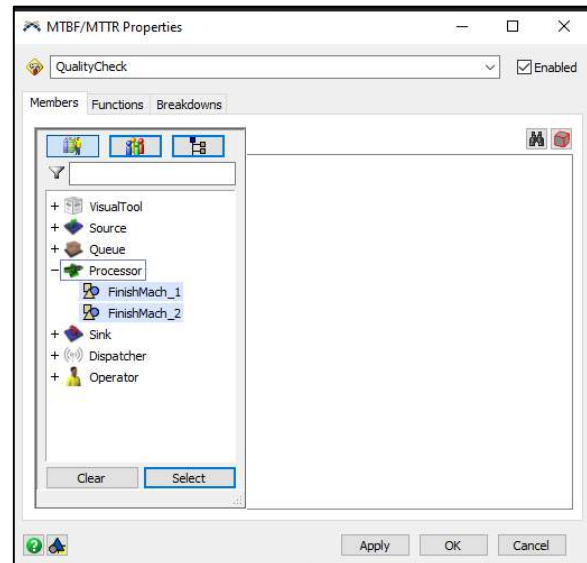
The first type of downtime on the Finishing Machines is a quality check that occurs every 10 minutes for 15 seconds; i.e., MTBF is a constant 10 minutes, and MTTR is a constant 15 seconds. The downtime occurs regardless of the machine's current state, and how much the machine has been used; i.e., MTBF depends on clock time only. No other resource is needed; i.e., the machine is just down for the duration of the self-check.

- In the **Toolbox**, press the  button and select the **MTBF MTTR** tool from the drop-down menu.
- Change the tool's name from *MTBFMTTR1* to *QualityCheck*.

The MTBF MTTR tool contains three tabs – **Members**, **Functions**, and **Breakdowns**. The settings for the quality check downtime are explained below.

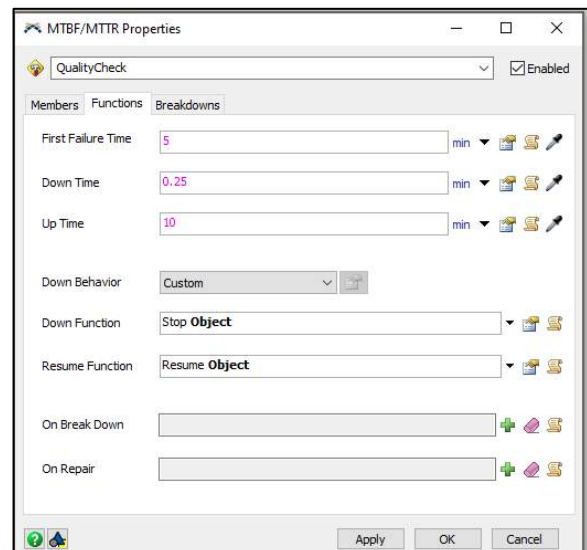
The **Members** tab assigns this downtime pattern to one or more objects; i.e., the downtime behavior defined on the **Functions** and **Breakdowns** tabs are applied to all objects in the **Members** tab.

- Use the  button to get the list of model objects, as shown in the figure to the right.
- Select both finishing machines, *FinishMach_1* and *FinishMach_2*, from the **Processor** category. Alternatively, if the category **Processor** is selected, all **Processors** are chosen.



The **Functions** tab defines how often downtimes occur and their durations. In this case, all values are deterministic or constant; thus, the default probability distributions will be replaced by constants, as shown in the figure to the right.

- Change the **First Failure Time** property from the exponential probability distribution to a constant 5 minutes. Since a quality check occurs every 10 minutes, this assumes the simulation starts halfway between downtimes.
- Change the **Down Time** property from the uniform probability distribution to a constant 15 seconds, 0.25 minutes.
- Change the **Up Time** property from the exponential probability distribution to a constant 10 minutes. This is the time between failures or between down states,



The default values are used for the **Down Behavior**, **Down Function**, and **Resume Function**. When an object experiences downtime, it is stopped for the duration of the downtime and then restarted. No resource is needed during the downtime; the machine processes the downtime itself.

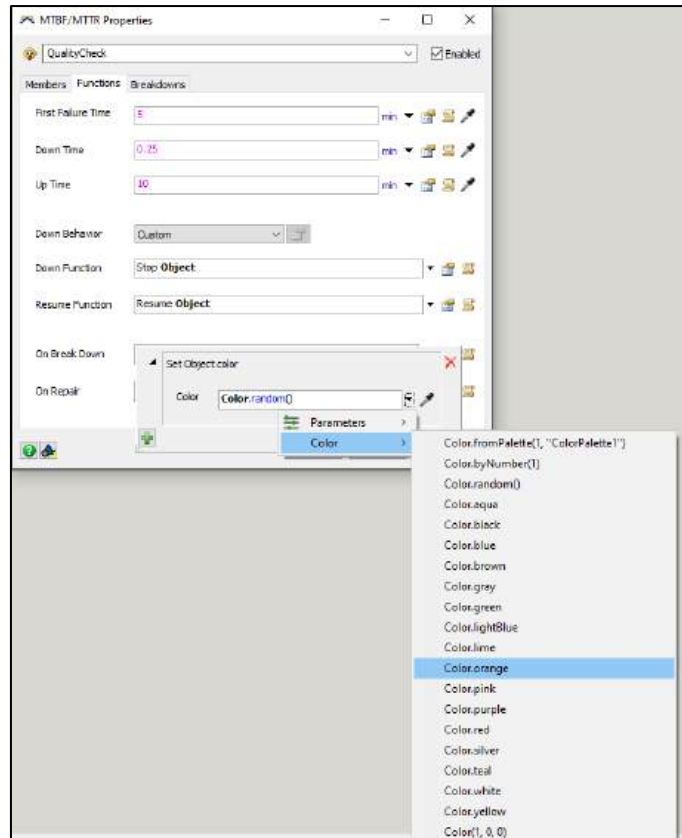
The **On Break Down** and **On Repair** triggers are used to change the object's color when it is down. In this case, the object is orange when it is down for a quality check. It is set back to green when it is not being checked.

As shown in the figure to the right:

- Use the  button on the **On Break Down** trigger to show a list of actions. Then, select the *Set Color (individual)* option from the drop-down menu.
- Use the **Color** dropdown list to change the **Color** value from *Color.random()* to *Color.orange*.

Similarly, the **On Repair** trigger sets the object's color back to green, indicating it is not down.

- Using the  button on the **On Repair** trigger, select the *Set Color (individual)* option from the drop-down menu.
- Use the **Color** dropdown list to change the Color value from *Color.random()* to *Color.green*.



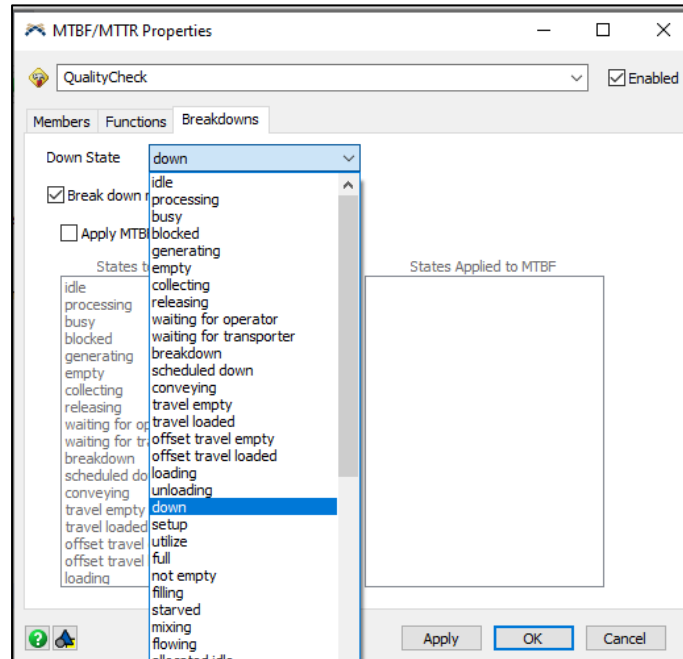
Now consider the third tab, **Breakdowns**.

For clarity, it is best to differentiate the downtime state for failures, which is a “breakdown,” as described in the next section, and the planned downtime for the quality checks.

For the check quality case, the time between failures (downtimes) is based on the simulation clock and not on any of the object's states; i.e., the machine will be unavailable for 15 seconds every 10 minutes, regardless of whether it is processing or not, so that it can upload data.

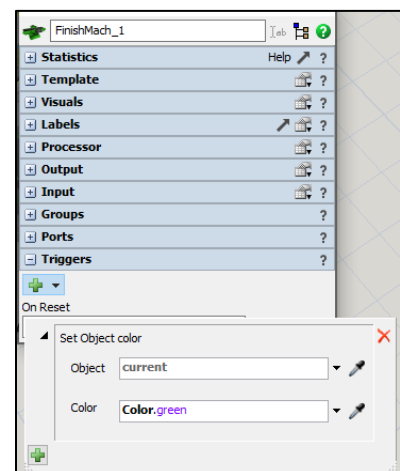
The default for breakdowns is to base the downtime on the simulation clock only and not on any object state. The default is used for the quality check. The default is when the *Apply MTBF to a set of states* box is unchecked. This will change when the downtime is due to a failure, which is described in the next section.

- On the **Breakdowns** tab, as shown in the figure to the right, change the **Down State** from the default, *breakdown*, to *down*, using the list of states on the menu. This state is more descriptive of the behavior since the quality check is not a breakdown.



Since the finishing machines' color is set to orange when undergoing a quality check, its color is not set back to its normal color if the simulation ends while the **Processor** is performing a quality check. To avoid this, as described below, add an **OnReset** trigger to **both Processors** that set their color to green whenever the model resets.

- Use the  button on the **Processor's Triggers** pane to add an **On Reset** trigger.
- Use the  button to the right of the trigger's text box to select the action, **Set Object Color**.
- As shown in the figure to the right, use the dropdown menu for **Object** values to change from the default, which is *item*, to *current*. The color change pertains to the current object, the **Processor**, and not an item being processed.
- Similarly, use the dropdown menu for **Color** values to change from the default value *Color.random()* to *Color.green*.
- Remember to define the trigger on **both Processors**.

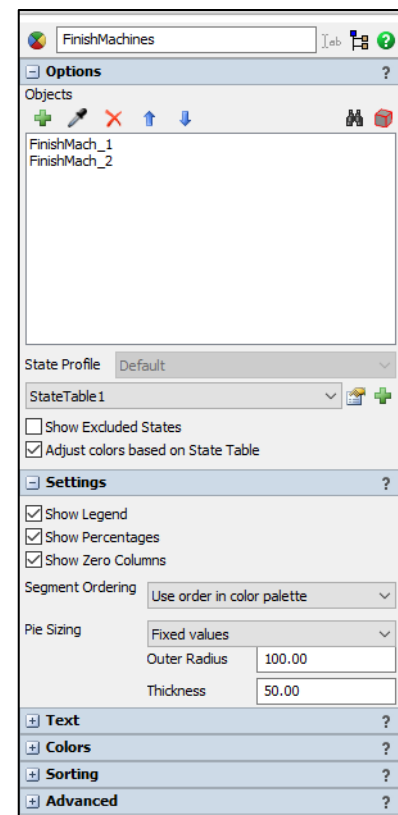
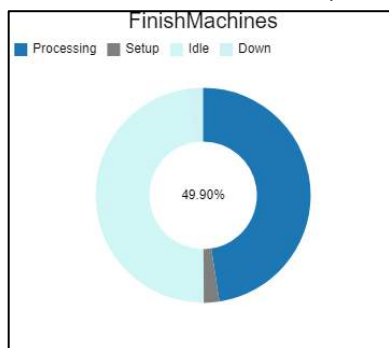


14.5 Composite state chart

To see the effects of downtime, add a chart to the **Dashboard Utilization**.

- Open **Dashboard Utilization** either through the **Toolbox** or the **Dashboard** button on the **Main Menu**.
- Select the *Composite State* diagram from the **State** section of the **Dashboard** library and place the diagram under the one created for the Finishing Operator. The Composite State diagram will show the combined utilization of both finishing machines.
- Change the name to *Finish Machines*.
- Use the  button in the **Objects** section to add the two **Processors**, as shown in the figure to the right.
- In the **Settings** pane, change the size of the pie chart
 - Set **Outer Radius** to 100.0
 - Set **Thickness** to 50.0.

If the model is reset and run, the chart should resemble the one below.

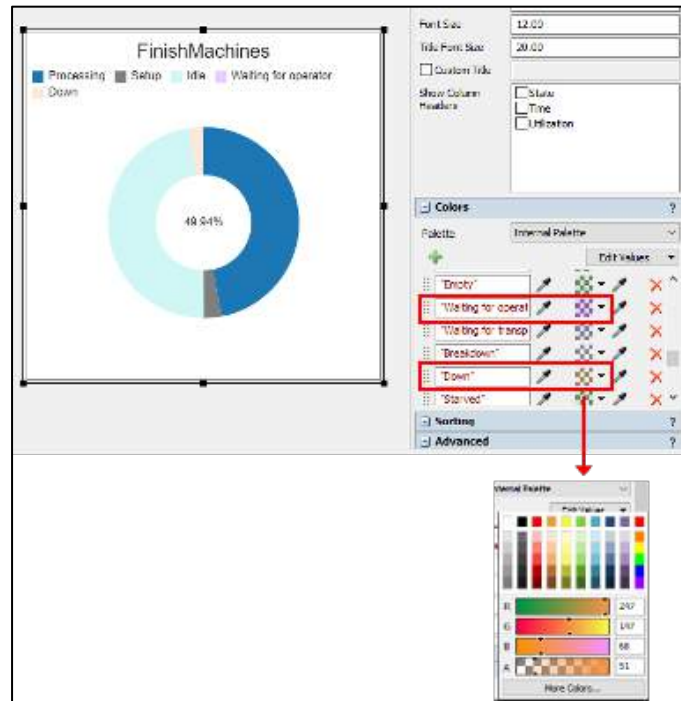


Notice all of the states the two finishing machines go through during the simulation – *Processing*, *Setup*, *Idle*, and *Down*. The **Processors** spend about half their time in the *Processing* state and half in the *Idle* state.

Also, notice the colors for *Idle* and *Down* states are very similar and hard to distinguish. We'll modify the *Down* color to orange.

As shown in the figure to the right,

- Select the chart so that its **Properties** appear in the right frame.
- In the **Colors** pane, scroll down to the “*Waiting for operator*” and “*Down*” states, as highlighted in the red boxes in the figure. We’ll change both states here because when *Down* is changed to orange, it is very close to the color of the *Waiting for operator* state.
- Select the dropdown menu to the right of each state. This will result in displaying the color palette indicated by the arrow in the figure.
 - Select orange for the *Down* state.
 - Select purple for the *Waiting for operator* state.



If the color seems transparent, change the A color factor (below the R, G, B factors) to 255. This will make the color opaque or have no degree of transparency.

The chart should now look like the one in the figure above. While the colors in the chart still need improvement, they are sufficient. Since there are many more topics to explore, the interested reader can explore customizing color palettes further in the *FlexSim User Manual*.



If you haven’t already done so, save the model. Recall that it is good practice to save often.



Use the **Save Model As** option in the **File** menu to make a copy of the existing model so that it can be customized beginning in the next section. Again, you can use any file name, but the next model is referred to as *Primer_7* in the primer.

14.6 Random MTTF/MTTR; MTBF based on system states; resource for repair

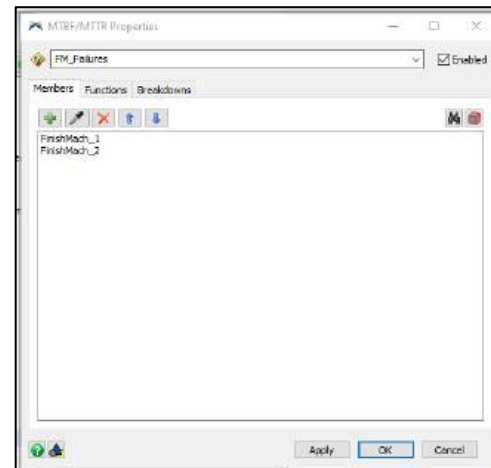
The second type of downtime on the finishing machines is a random machine failure. The times between failures and the times to repair are both considered random variables or probabilistic and are based on probability distributions. For this type of downtime, a resource – the finishing operator – is needed to perform the repair.

The base model for the additions described in the remainder of the chapter is **Primer_6** that was saved above. However, a copy of that file was saved as **Primer_7**; thus, we begin with that file.

- In the Toolbox, press the  button and again select the **MTBF MTTR** tool from the drop-down menu.

Modify the Members tab as follows and as shown in the figure to the right.

- Change the name of the downtime from *MTBFMTTR1* to *FM_Failure*.
- On the Members tab, just as was done with the QualityCheck downtime described above, use the  button and select both finishing machines, *FinishMach_1* and *FinishMach_2*, from the **Processors** category.



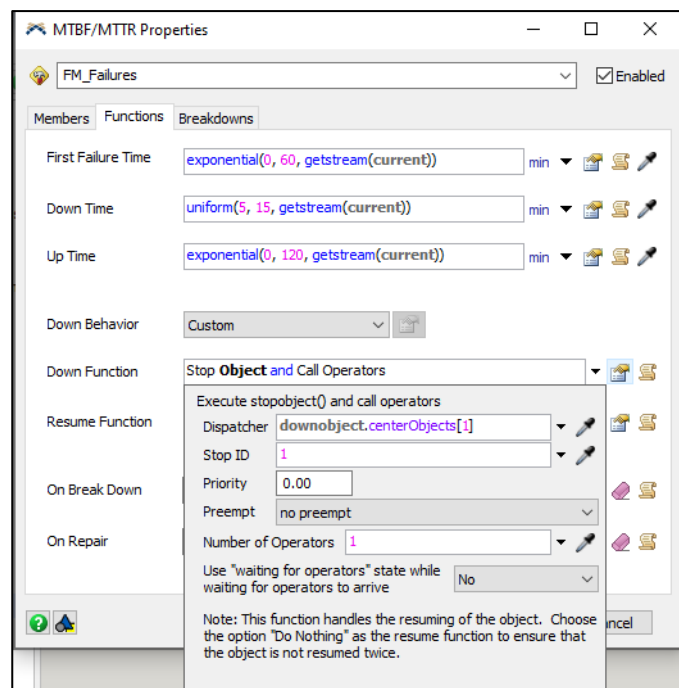
The **Functions** tab defines how often downtimes occur and their duration. It also defines what actions occur in the simulation whenever a downtime occurs.

The **Functions** tab for this example is shown in the figure to the right and explained below.

In this case, all of the times are obtained by sampling from probability distributions. For this example, the default distributions are used.

The exponential distribution is the default for the times between failures (**First Failure Time** and **Up Time**). The only parameter required for this distribution is its mean value.

The default distribution for the repair times is uniform. Two parameters are required for the uniform distribution: the lowest and highest values, i.e., the shortest and longest possible repair times. All repair times are assumed to be equally likely between these two limits.



- Change the **First Failure Time's** second parameter in the exponential distribution from the default value of 1000 to 60 minutes. This is one-half of the distribution's mean value, and it assumes the simulation started halfway between downtimes.
- Change the **Down Time's** parameters from the default values of 50 and 100 to 5 and 15. This assumes downtime is uniformly distributed between 5 and 15 minutes.
- The **Up Time** property is the time between failures or time between down states. It is assumed to be exponentially distributed with a mean of two hours (120 minutes). Therefore, change its second parameter, the mean, from the default value of 1000 to 120.
- Change the **Down Function** from the default *Stop Object* to *Stop object and call operators* via the property's drop-down menu. The resulting interface is shown in the figure above. All default values are used. This selection stops the object during downtime and calls a finishing operator to perform the repair. The default calls the object connected to its center port; in this case, it is the **Dispatcher**.
- According to the note at the bottom of the interface for the **Down Function** property, the **Resume Function** value is to be set to *Do Nothing* via the drop-down menu. This is because the logic for resuming is managed by the **Down Function**.

As with the *QualityCheck* downtime, the **On Break Down** and **On Repair** triggers are used to change the color of the object when it is down.

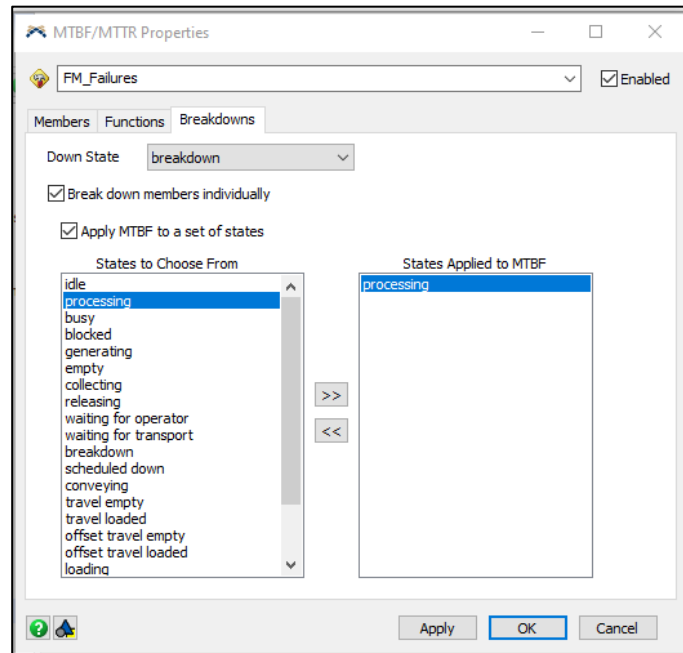
- For the **On Break Down** trigger, as with the *QualityCheck* downtime, select **Set Color (individual)** option. However, for failures the object is colored red when it is down; therefore, choose the *Color.red* option.
- For the **On Repair** trigger, as with the *QualityCheck* downtime, select **Set Color (individual)** option and then set the color back to green using the *Color.green* option.

As stated previously, using the correct probability distributions in a model is important since the choice can significantly impact system performance and simulation results. *FlexSim* includes many types of probability distributions. A full discussion of how to select distributions is beyond the scope of this introductory primer.

In this example, the time between failures (downtimes) is based on the state of the object and not the simulation clock. The finishing machines only accrue downtime when they are running or processing. For example, if the time to the next failure on a machine is two hours and the machine operates only 50% of the time, then the next downtime will occur in four hours of simulation time. Also, a failure will occur only when the object is in one of the selected states. For example, a failure will occur only when a machine is processing and not when it is idle.

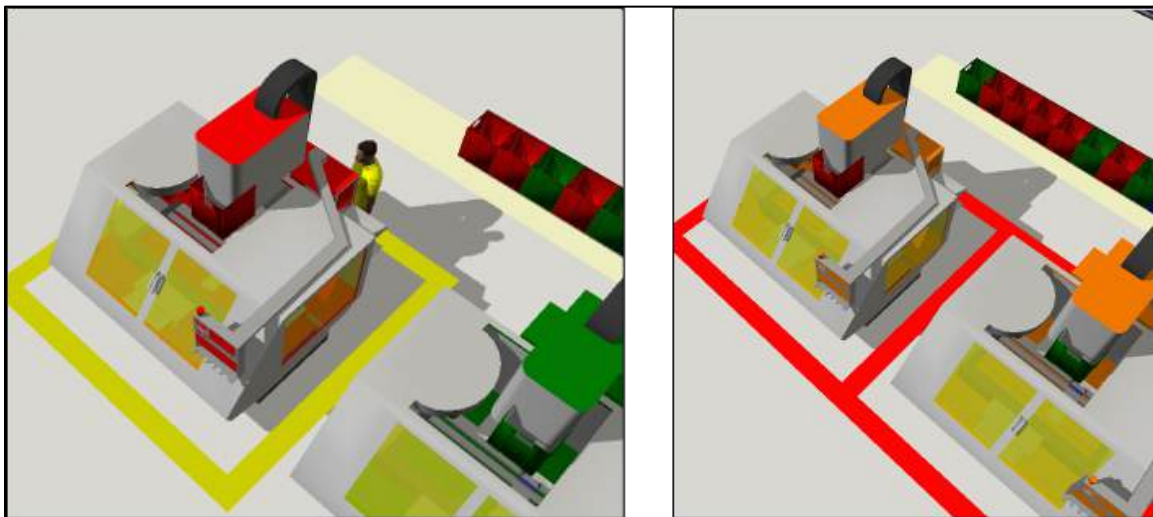
The above situation is defined on the **Breakdowns** tab and is shown in the figure to the right.

- Check the box *Apply MTBF to a set of states*.
- Move states from the **States to Choose From** list to **States Applied to MTBF** using the >> move button. Select the *processing* state, then click the >> button so the *processing* state moves to the **States Applied to MTBF** pane. In this case, only the *processing* state is selected.



- **Reset** and **Run** the model and observe the activities at the Finishing Machines.

In the figure below, the screenshot on the left shows one Finishing Machine in the breakdown state, and the Finishing Operator is at the machine performing the repair. The down machine is colored red, and the up machine is colored green. The screenshot on the right shows both Finishing Machines in the down state (colored orange) for a quality check and the Finishing Operator is not involved.

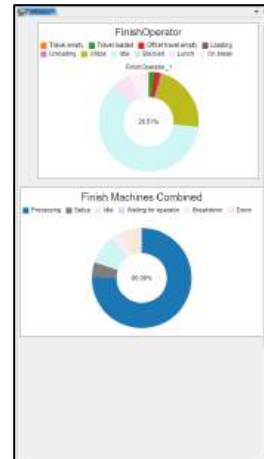


If you haven't already done so, save the model. Recall that it is good practice to save often.

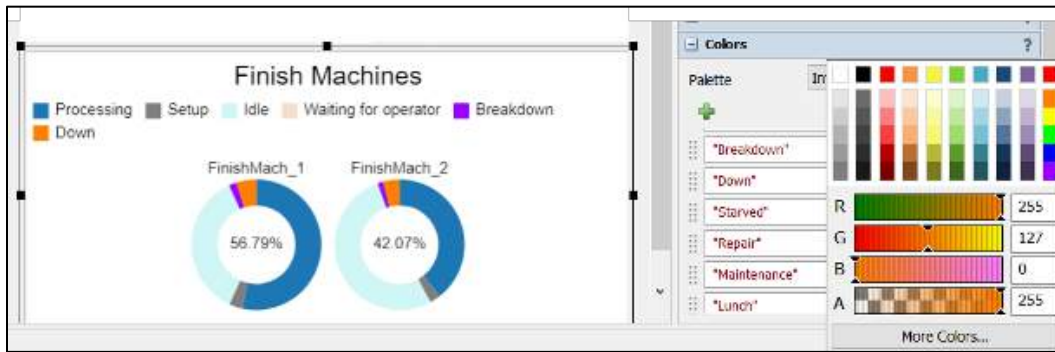
14.7 State chart for each finishing machine

Previously, a state chart was created that showed the combined utilization of the finishing machines. We'll add a chart that shows the utilization of each machine.

- As shown in the figure to the right, resize the charts on the *Utilization Dashboard* to make room for a third chart.
- The size of the pie charts can be changed through the **Pie Sizing** property in the chart's **Settings** pane, e.g., set the **Outer Radius** to 75.0 (**Thickness** = 50.0).
- Note the new state, *Breakdown*, is shown in the composite state chart for the finishing machines.
- Before creating the new chart, change the name of the *FillingMachines* dashboard to *Filling Machines Combined*.



- In the **Dashboard Library**, select the *State* chart type in the **State** section, then select *Pie Chart* from the pop-up menu.
- Drag the selection to the dashboard workspace and resize the chart to fit in the remaining space on the dashboard.
- Name the chart *Finish Machines*.
- In the **Objects** section of the **Options** tab, click the  button and choose the **Select Objects** option. Then, select the category **Processors** and press *Select*. As a result, both *FinishMach_1* and *FinishMach_2* should appear in the **Objects** section of the interface.
- In the **Settings** pane, set the **Outer Radius** to 50.0 and **Thickness** property to 20.
- Since the default color of the pie slices for *Idle* and *Down* states are quite similar, they may be hard to differentiate on the chart. Therefore, change the *Down* state color as was done earlier and as shown in the figure below.
 - In the **Colors** pane, scroll down to the “*Down*” states.
 - Select the dropdown menu to the right of the state. This will result in displaying the color palette.
 - Select orange for the *Down* state and adjust the color's A factor to 255.

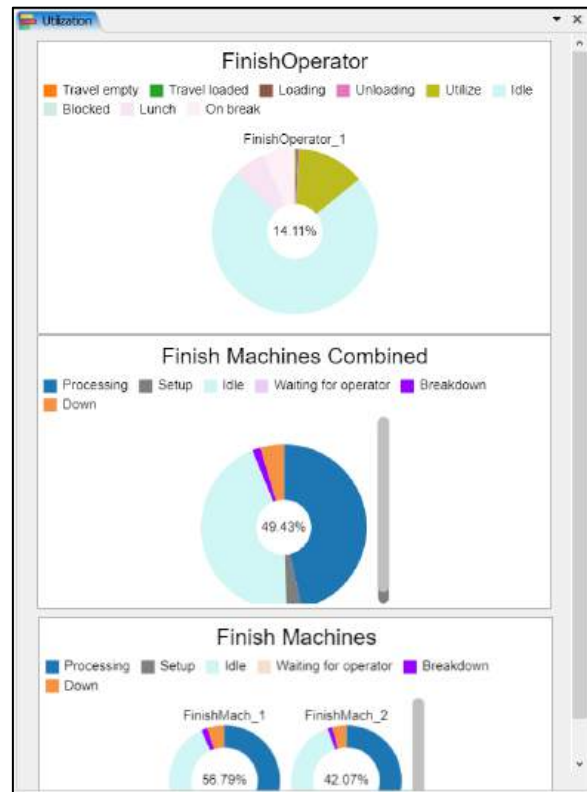


- **Reset** and **Run** the model run for 80 hours (4800 minutes). The results should be similar to that shown in the figure to the right.

Based on this simulation run, the two Finishing Machines are each busy about 57% and 42% of the time, respectively, or about 49% overall. These percentages include the times the **Processors** are in the processing and setup states (overall about 46.5% and 2.9%, respectively). These values can be found by hovering over the pie segments in the *Finish Machines Composite* chart.

The pie chart also shows the small percentage of the time that they were in the breakdown (about 1.6%) and down for quality check (about 4.6%) states. The remainder of the time, the idle state, is when the machines were ready and available but had nothing to process; in this case, that is about 44.4% overall. Only a negligible amount of time, were the Finishing Machines waiting for the Finishing Operator.

The utilization of the Finishing Operator is the percentage of time the operator is busy, which in this case is about 14.1%. This includes traveling both loaded with containers and unloaded, loading and unloading containers, performing setup operations (part of the Utilize state on the chart), and repairing the Finishing Machines when they break down (the other part of the Utilize state). In addition, the operator is in the On-break state about 6.25% of the time. The remaining 73% of the time, the operator is in the idle state, ready and available, but has no tasks to perform.



If you haven't already done so, save the model. Recall that it is good practice to save often.



Use the **Save Model As** option in the **File** menu to make a copy of the existing model so that it can be customized beginning in the next section. Again, you can use any file name, but the next model is referred to as *Primer_8* in the primer.

PART IV – MODELING THE PACKING AREA

This section involves modeling the packing of finished containers with components.

- Chapter 15 first provides an overview of how the packing area works. Subsequently, component items are defined, such as how and when they are created (in batches). A data table is created to store information on the operation. Storage areas for the components are added to the model.
- Chapter 16 uses the Separator object to unpack batches of components when they arrive. The unpacking task requires the Finishing Operator. The Combiner object is used to model the packing of containers with components. A robot places components in containers. The Packing Robot is subject to state-based downtime like the Finishing Machines, and the Finishing Operator performs the repairs. A time series plot is added to show the inventory levels of the components.

15 MODELING THE PACKING AREA – PART 1

Chapter 15 first provides an overview of how the packing area works. Subsequently, component items are defined, such as how and when they are created (in batches). A data table is created to store information on the operation. Storage areas for the components are added to the model.

The model from the previous section is extended to include a packing area. The following is an overview of the approach to modeling the Packing Area, which involves twelve steps divided into two parts.

Part 1 – batches to components

1. Description of the packing area and modeling approach.
2. Flow items for components – represent each component as a separate type of flow item.
3. Store operation data in tables – use Parameters Tables to store each type of component's batch frequency (time between batches) and batch size. Storing values in data tables rather than in the object facilitates conducting what-if analyses.
4. Create batches of components, i.e., components arrive at the Packing Area in specified quantities and frequencies.
5. Provide a queue for batches to await processing, i.e., each batch waits in a queue/buffer for the Finishing Operator to unpack the components.
6. Provide storage areas for components, i.e., each type of component has its place to be stored in the Packing Area.

Part 2 – packing operation (the next chapter)

1. Provide a means to unpack components, i.e., each batch is unpacked into its storage area.
2. Define Finishing Operator tasks – the Finish Operator moves each batch from a queue to the component's storage area, unpacks the batch, and loads the components into storage for the packing operation.
3. Containers arrive by conveyor, i.e., containers are transported from the Finishing Area to the Packing Area via a conveyor; thus, they are processed in a first-in, first-out manner.
4. Pack containers based on their type – a mix of components is packed into each container based on the container type. Component flow items are combined or packed into containers.
5. Pack containers by robot – a robot moves components from their storage area to containers. The robot also removes containers from the incoming conveyor and moves containers to the outgoing conveyor. Robots incur downtimes that the Finishing Operators address.
6. Check component inventory levels – use graphs to assess the effectiveness of the components' replenishment parameters.

15.1 Description of the packing area and modeling approach

In the packing area, finished containers are packed by a robot with components produced elsewhere. Again, the model being developed in this primer is limited in scale since the focus is on the system's main elements. Later, not a part of the primer, the main elements can be scaled up to represent the entire system. Therefore, for the primer, only two types of components, referred to as A and B, are considered to be packed in the three types of containers. Component A is a purple boxed-shaped item, and Component B is a white cylinder-shaped item.

As mentioned before, a very important concept that bears repeating is that it is good modeling practice to build smaller models first to test, validate, and verify the approach and methods, then scale up to represent the real system.

For now, it is assumed that the components arrive at the packing area on a schedule and in batches, i.e., three Component As arrive every hour, and five Component Bs arrive every half hour. In a later section in the primer, the model is modified so that the components arrive based on current inventory levels and specified reorder points.

Each component item will contain some information about itself – its type and batch size. This information is stored in labels on the components. **Labels** are user-defined data that can reside in objects and items. They consist of a label name and a label value. The value can be of various data types, such as numeric, string, array, pointer, etc. In this case, for simplicity, only numeric values are used. The two labels that will be created on each component are:

- For component type, the label's name will be *CompType*, and its values will be 1 for A, 2 for B, etc. Since we refer to the components as A, B, etc., we could store each item's type as a string data type and thus have values A, B, etc. However, for simplicity, component types are defined as numeric.
- For the component's batch size, the label's name will be *BatchSize*, which will have numeric values such as 3 and 5.

An arriving batch waits for a Finishing Operator to travel to it. Depending on the workload of the Finishing Operator and the frequency of batches, multiple batches might arrive and await the Operator. Therefore, a small batch queue is included in the model. The Finishing Operator moves each batch to its component storage area and unloads the components into the storage area.

At the packing station, a robot loads each container type with its mix of components.

One packing station should be enough to support two Finishing Machines.

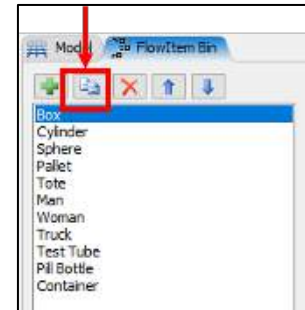
The base model for the additions described in this chapter is **Primer_7** that was saved at the end of Chapter 14. However, a copy of that file was saved as **Primer_8**; thus, we begin with that file.

15.2 Flow items for components

The first step is to create two new flow items to represent components A and B. This is done via the **FlowItem Bin**, which is accessed by the button on the **Main Menu** bar (to the right of the **Tools** button) or through the **Toolbox**.

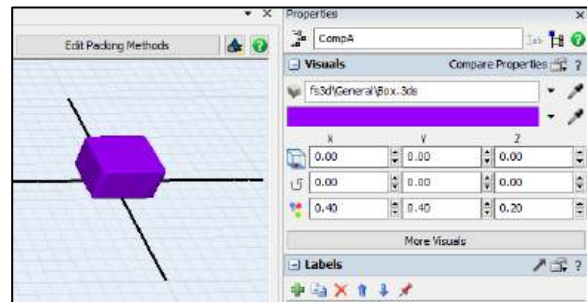
Create the first component as follows.

- Duplicate the **Box** item by highlighting it in the list of flow items (Box, Cylinder, Sphere, ...) and pressing the **Duplicate the selected flow item** button (to the right of the  button) and highlighted by the red box in the figure to the right. The result is a new flow item at the bottom of the list called "Box copy."



Select the new item *Box copy* and in the **Properties** interface, make the following changes. The result should be as shown in the figure to the right.

- Change the name of the item from *Box copy* to *CompA*.
- Change its color from *brown* to *purple* using the color palette dropdown menu.
- Change its size from the default values ($x=0.61$, $y=0.61$, $z=0.3$) to $x=0.4$, $y=0.4$, $z=0.2$.
- Change the location to $x=0.0$, $y=0.0$, $z=0.0$.

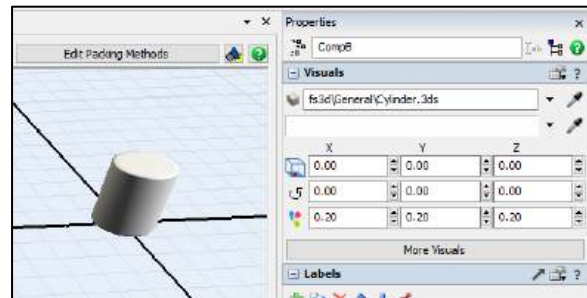


Repeat the above process for the second component.

- Duplicate the **Cylinder** item by highlighting it in the list of flow items (Box, Cylinder, Sphere, ...) and pressing the **Duplicate the selected flow item** button (to the right of the  button). The result is a new flow item at the bottom of the list called "Cylinder copy."

Select the new item *Cylinder copy* and in the **Properties** interface, make the following changes. The result should be as shown in the figure to the right.

- Change the name of the flowitem from *Cylinder copy* to *CompB*.
- Change its color from *brown* to *white* using the color palette dropdown menu.
- Change its size from the default values ($x=0.70$, $y=0.70$, $z=0.8$) to $x=0.2$, $y=0.2$, $z=0.2$.
- Change the location to $x=0.0$, $y=0.0$, $z=0.0$.



15.3 Store operation data in tables

The frequency of batch arrivals (time between arrivals of batches) and batch size vary by container type. Both of these factors are likely to be important operational considerations in the overall system design. Therefore, the frequency and batch size will be stored in **Parameters Tables** so they can be easily changed to evaluate alternatives.

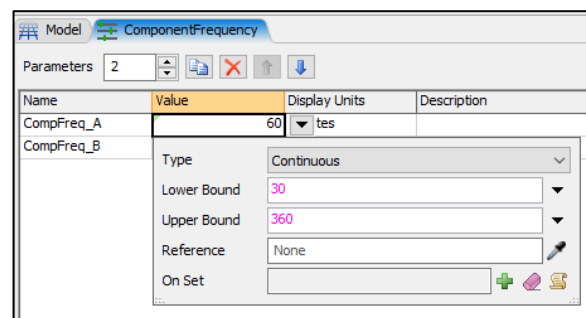
Two **Parameters Tables** are used to store values for these factors — one for the frequency of each batch's occurrence and the other for the size of the batch. The parameters could all be in one table, but using separate tables makes updating and adding information on new components easier. Using separate tables also facilitates importing the data into the model through *MS Excel*.

For the first table:

- Create a **Parameter Table** for the frequency of batch arrivals by clicking the  button at the top of the **Toolbox**, selecting **Statistics**, and then selecting **Model Parameters Table**.
- In the **Properties** window, change the name to *ComponentFrequency*.

As shown in the figure to the right:

- Increase the **Parameters** value to 2.
- Change the **Name** in each row from *Parameter1* and *Parameter2* to *CompFreq_A* and *CompFreq_B*.
- For each component, use the **Value** dropdown interface to change **Lower Bound** to 30 and **Upper Bound** to 360, respectively. This allows the time between batch arrivals to be set between 30 and 360 minutes. If a user tries to set the value outside of this range, an error message is displayed.
- Set the **Values** for *CompFreq_A* and *CompFreq_B* to 60 and 30, respectively.

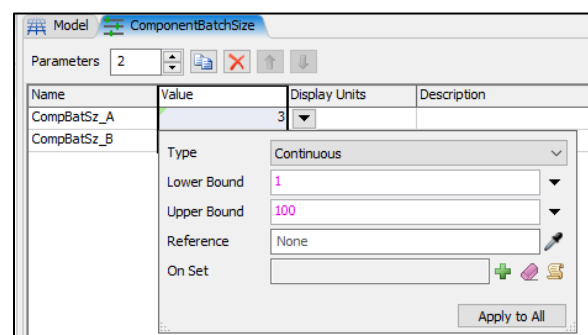


For the second table, repeat the steps above:

- Create a **Parameter Table** for the frequency of batch arrivals by clicking the  button at the top of the **Toolbox**, selecting **Statistics**, and then selecting **Model Parameters Table**.
- In the **Properties** window, change the name to *ComponentBatchSize*.

As shown in the figure to the right:

- Increase the **Parameters** value to 2.
- Change the **Name** in each row from *Parameter1* and *Parameter2* to *CompBatSz_A* and *CompBatSz_B*.
- For each component, use the **Value** dropdown interface to change the **Lower Bound** to 1 and **Upper Bound** to 100, respectively. This allows the batch sizes to be set as low as 1 and as high as 100. If a user tries to set the value outside of this range, an error message is displayed.
- Set the **Values** for *CompBatSz_A* and *CompBatSz_B* to 3 and 5, respectively.





If you haven't already done so, save the model. Recall that it is good practice to save often.

15.4 Creating batches of components

Each batch of components needs to be created by a **Source**.

For component A:

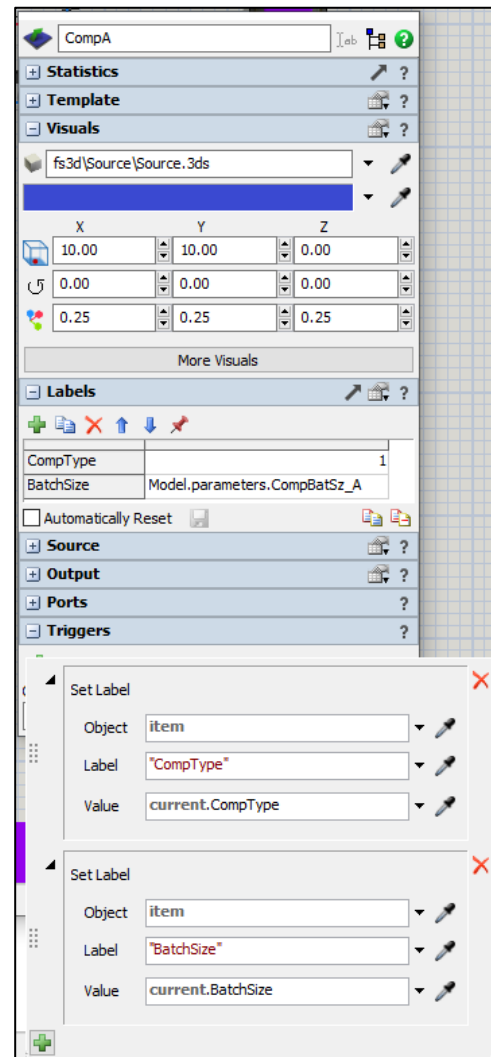
- Create a source for component As to enter the model by dragging out a **Source** object.
- Rename the **Source** *CompA*.

As shown in the figure to the right, in the **Visuals** pane,

- Set the **size** to $x=0.25$, $y=0.25$, $z=0.25$.
This does not affect the model's performance; it is just made smaller to be less apparent. Since the **Sources** are not a physical part of the system, they could be hidden, but we'll keep them small but visible.
- Set the **location** to $x=10.0$, $y=10.0$, $z=0.0$.
At this point, the location is not critical, and the object could just be moved around on the modeling surface, but for consistency, we specify the location.

As shown in the figure to the right, in the **Labels** pane, create two object labels, which will contain user-defined data.

- Create the first label by using the  button and selecting *Add Number Label*.
- Change the name of the label from *label1* to *CompType*.
- Since this object creates Component As or, in numerical terms, Component 1, set the value to *1*.
- Create the second label by using the  button and selecting *Add FlexScript Label*.
- Change the name of the label from *label1* to *BatchSize*.
- The batch size is stored in a Parameters Table so the value is the *FlexScript* command that references the appropriate cell in the table, which is `Model.parameters.CompBatSz_A`
Be sure to enter this value correctly.



When a batch item is created, it will contain the same two labels as those defined above for the logic. The labels are added through triggers that fire when an item is created. Therefore, in the **Triggers** pane:

Create the first label trigger, which will add a label to the item being created, and its value will be the same as the value on the **Source** object. Basically, the **Source** creates items that are of Type 1.

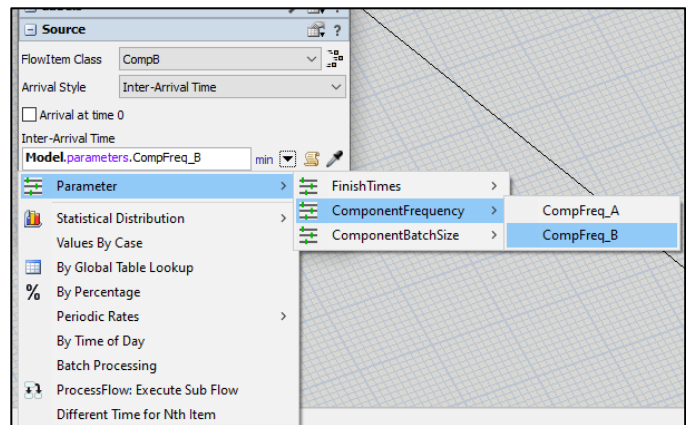
- Press the  button under **Triggers** and select *On Creation*.
- Press the  button to the right of the **On Creation** text box, select *Data*, and then *Set Label*.
- Complete the **Set Label** dialogue box as shown in the figure above – leave the **Object** value as *item*, change the **Label** to “*CompType*”, and change **Value** to *current.CompType*.

Create the second label trigger, which will add another label to the component item being created, and its value will be the same as the value on the **Source** object. Basically, the **Source** creates items that have the batch size for Type 1 components.

- Press the  button at the bottom of the **Set Label** dialogue box, select *Data*, and then *Set Label*.
- Complete the **Set Label** dialogue box as shown in the figure above – leave the **Object** value as *item*, change the **Label** to “*BatchSize*”, and change **Value** to *current.BatchSize*.

Update the properties in the **Source** pane as described below and shown in the figure to the right.

- Using the drop-down menu, change the **FlowItem Class** property from the default *Box* to *CompA*. In this case, a batch will be represented by one component, although a different flow item could be used to visually represent a batch.
- Change the **InterArrival Time** by using the dropdown menu button to the right of the value; select from the cascading submenus *Parameters*, then *ComponentFrequency*, then *CompFreq_A*.



This results in an **Inter-Arrival Time** value of **Model.parameters.CompFreq_A**, which will return the value stored in the **Parameters Table**; in this case, 60 minutes.

For Component B

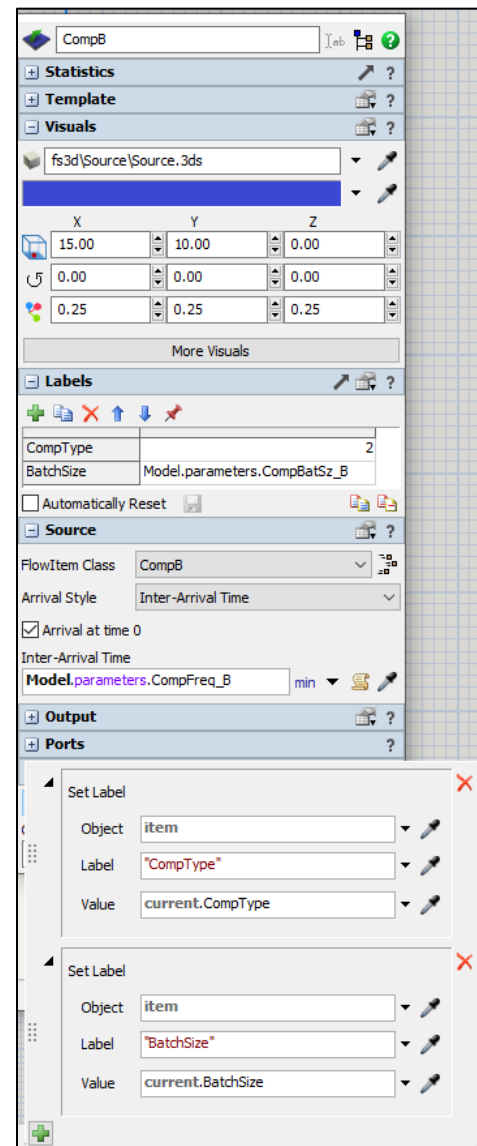
- Create a source for component Bs to enter the model by dragging out a **Source** object.
- Rename the **Source** *CompB*.

As shown in the figure to the right, in the **Visuals** pane,

- Set the **size** to $x=0.25$, $y=0.25$, $z=0.25$.
Again, this does not affect the model's performance; it is just made smaller to be less apparent.
- Set the **location** to $x=15.0$, $y=10.0$, $z=0.0$.
Again, at this point in the modeling, the location is not critical, but for consistency, we specify the location.

As shown in the figure to the right, in the **Labels** pane, create two object labels, which will contain user-defined data.

- Create the first label by using the  button and selecting *Add Number Label*.
- Change the name of the label from *label1* to *CompType*.
- Since this object creates Component Bs or, in numerical terms, Component 2, set the value to 2.
- Create the second label by using the  button and selecting *Add FlexScript Label*.
- Change the name of the label from *label1* to *BatchSize*.
- The batch size is stored in a Parameters Table so the value is the *FlexScript* command that references the appropriate cell in the table, which is.
`Model.parameters.CompBatSz_B`
Again, to avoid errors, be sure to enter this value correctly.



As in the first Source, triggers are used to create labels on the batch item when it is created.

Create the first label trigger, which will add a label to the item being created, and its value will be the same as the value on the **Source** object. Basically, the **Source** creates items that are of Type 2.

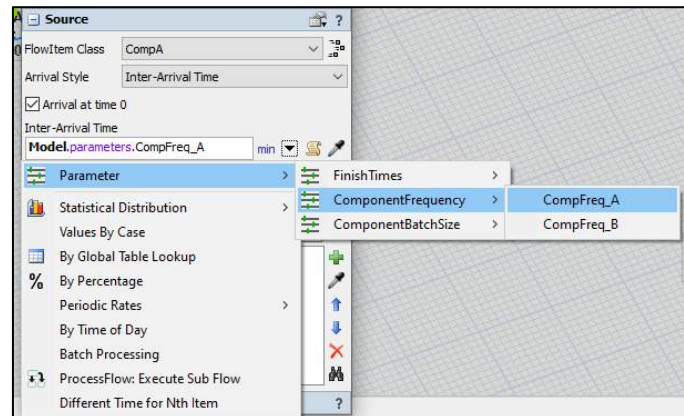
- Press the  button under **Triggers** and select *On Creation*.
- Press the  button to the right of the **On Creation** text box, select *Data*, and then *Set Label*.
- Complete the **Set Label** dialogue box as shown in the figure above – leave the **Object** value as *item*, change the **Label** to *"CompType"*, and change **Value** to *current.CompType*.

Create the second label trigger, which adds another label to the component item being created. Its value will be the same as the value on the **Source** object. Basically, the **Source** creates items that have the batch size for Type 2 components.

- Press the  button at the bottom **Set Label** dialogue box, select *Data*, and then *Set Label*.
- Complete the **Set Label** dialogue box as shown in the figure above – leave the **Object** value as *item*, change the **Label** to “*BatchSize*”, and change **Value** to *current.BatchSize*.

Update the properties in the **Source** pane as described below and shown in the figure to the right.

- Using the drop-down menu, change the **FlowItem Class** property from the default *Box* to *CompB*. In this case, a batch is visually represented by a single component.
- Change the **InterArrival Time** by using the dropdown menu button to the right of the value; select from the cascading submenus *Parameters*, then *ComponentFrequency*, then *CompFreq_B*.



This results in an **Inter-Arrival Time** value of **Model.parameters.CompFreq_B**, which will return the value stored in the **Parameters Table**; in this case, 30 minutes.

15.5 Queue of batches awaiting processing

A **Queue** is now added to the model to store the batches of components until they are unpacked by the Finishing Operator.

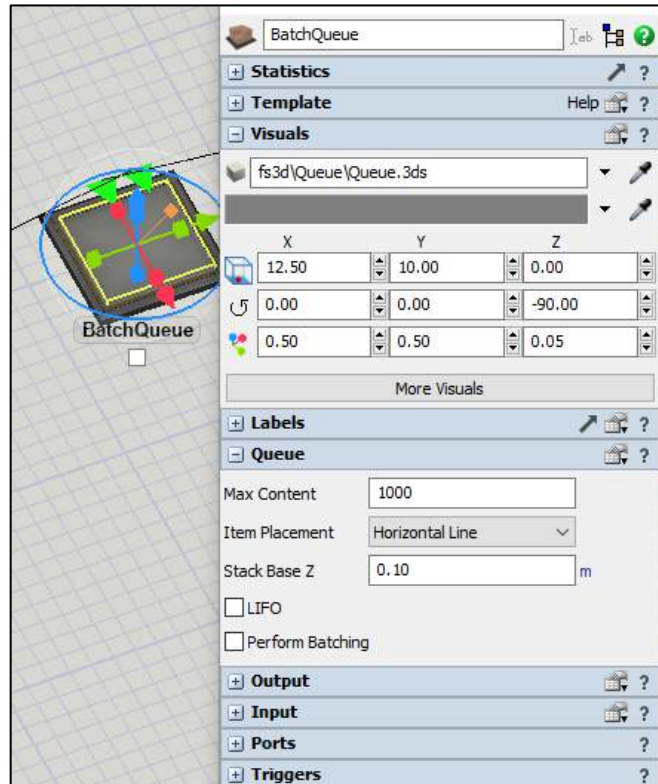
- Drag out a **Queue** object from the Library and place it between the two **Sources** that generate batches of components.
- Name the **Queue** *BatchQueue*.

Modify the **Queue**'s **Visuals** pane as shown in the figure to the right and described below.

- For the color property, select gray from the drop-down menu.
- Set the **Location** values to $x = 12.50$ and $y = 10.00$, which puts the **Queue** between the two component **Sources**.
- Set the **Size** parameters to $x = 0.50$, $y = 0.50$, and $z = 0.05$. so the Queue is just a bit larger than the components.
- Rotate the **Queue** -90 degrees about the z-axis. The flow to/from the queue is up/down not left/right; i.e., packing occurs below the queue in the layout.

Modify the **Queue**'s **Queue** pane as shown in the figure to the right and described below.

- Set the **Item Placement** property to *Horizontal Line* so the batches form a single line, and since the **Queue** is rotated, the line of batches will form away from the Packing Area.



15.6 Storage areas for components

Each component type has its place to be stored in the Packing Area. Batches of components are unloaded to the storage area, and a robot selects components from the storage area to be packed into containers.

Therefore, a separate **Queue** is used for storing each component type. Since their properties are very similar, it is easier to set up one and then copy, paste, and edit it to create the other.

Create the first component's queue.

- Drag out a **Queue** object from the Library and place it near the first component's **Source**.
- Name the **Queue** *StoreCompA*.

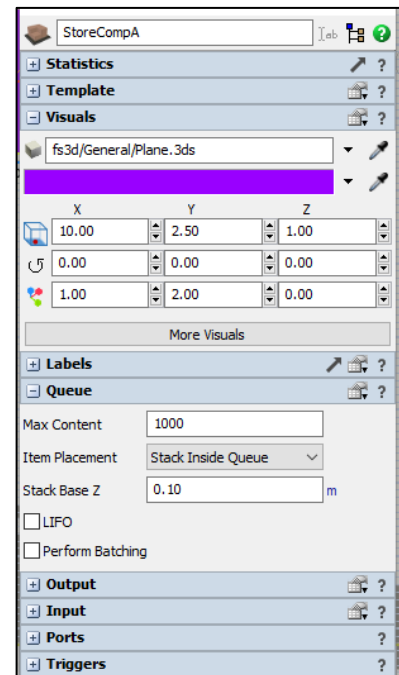
Modify the **Queue**'s **Visuals** pane as shown in the figure to the right and described below.

- Using the dropdown menu, set the 3D shape to a *Plane*. After the selection, the text box should contain *fs3d/General/Plane.3ds*, which is the *FlexSim* image file for a plane.
- For the color property, select *purple* from the color palette in the drop-down menu. The **Queue** is now the same color as the component it stores.
- Set the *x*, *y*, and *z Location* values to *10.00*, *2.50*, and *1.00*, respectively.

Note that the *z location* value is *1.0*; i.e., the queue is located one meter above the floor. This is better ergonomically for the Finish Operator. Later, a visual object – a non-functional object – will be added so it appears that the storage location is on a table, not in mid-air. This does not affect the system's performance but looks more realistic.

The table object is added later because it will be used in several places in the model.

- Set the *x*, *y*, and *z Size* values to *1.00*, *2.00*, and *0.00*, respectively.

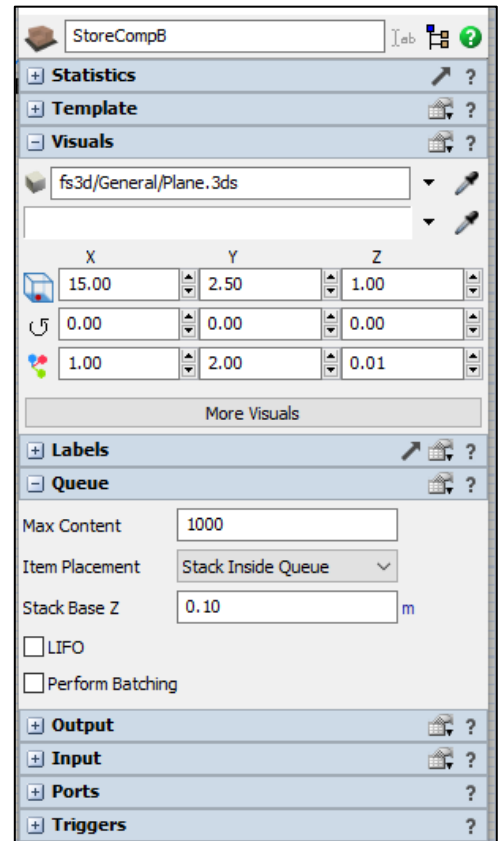


Create the second component's Queue.

- Copy and Paste the *StoreCompA Queue* by either of the following methods:
 - Select (click on) the **Queue**, press the *Ctrl-C* keys, click somewhere on the modeling surface, and then press the *Ctrl-V* keys.
 - Right-click on the **Queue**, select **Edit** from the menu, then select **Copy**. Right-click somewhere on the modeling surface, select **Edit** from the menu, and then select **Paste**.
- Change the name of the **Queue** to *StoreCompB*.

Modify the **Queue's Visuals** pane as shown in the figure to the right and described below.

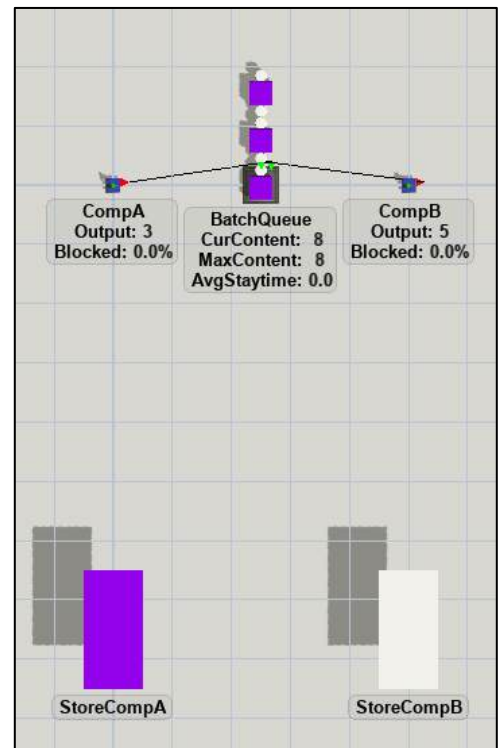
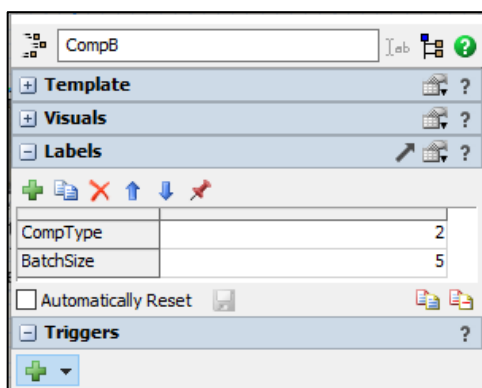
- For the color property, select *white* from the color palette in the drop-down menu. The **Queue** is now the same color as the component it stores.
- Set the *x*, *y*, and *z* **Location** values to *15.00*, *2.50*, and *1.00*, respectively.



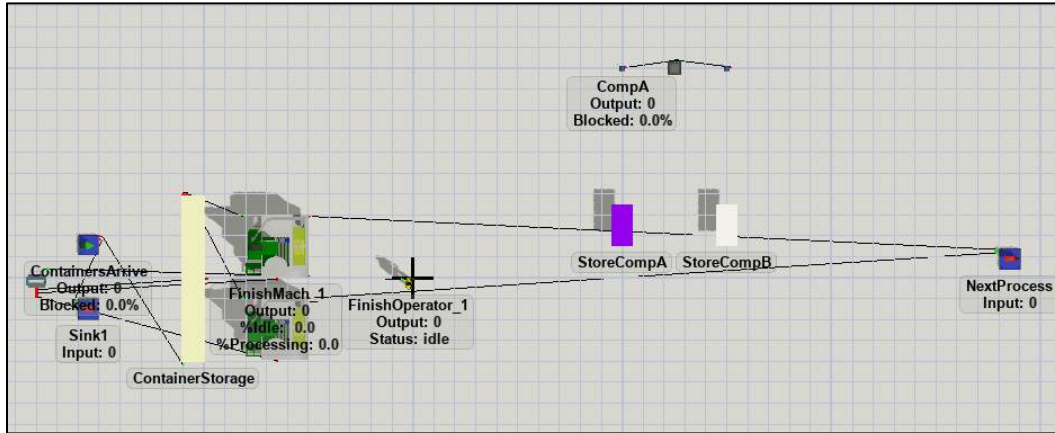
- **Reset** and **Run** the model. Validate that the arrivals of the component batches occur as expected; one batch of component As arrives every 60 minutes, and one batch of component Bs arrives every 30 minutes.

To the right is a screenshot of the model just after two hours of simulated time showing the expected queue contents – three batches of As and five batches of Bs.

Select one of the Component Bs in the batch queue. Its label value should be as shown below.



Also, check the overall state of the model, as shown in the figure below. At its current stage of development, the Packing Area needs to be completed; batches only arrive as per the prescribed schedule. Also, it is not connected to the Finishing Area. Both of these issues are addressed in the next chapter. Again, this illustrates validating and verifying as a model is developed.



If you haven't already done so, save the model. Recall that it is good practice to save often.



Use the **Save Model As** option in the **File** menu to make a copy of the existing model so that it can be customized beginning in the next section. Again, you can use any file name, but in the primer, the next model is referred to as `Primer_9`.

16 MODELING THE PACKING AREA – PART 2

Chapter 16 uses the Separator object to unpack batches of components when they arrive. The unpacking task requires the Finishing Operator. The Combiner object is used to model the packing of containers with components. A robot places components in containers. The Packing Robot is subject to state-based downtime like the Finishing Machines, and the Finishing Operator performs the repairs. A time series plot is added to show the inventory levels of the components.

This chapter is a continuation of the previous chapter and completes the modeling of the packing area. The steps needed to do so are as follows. Each step is discussed in the following sections.

1. Unpack components – each batch is unpacked into its storage area.
2. Finishing Operator tasks – the Finish Operator moves each batch from a queue to the component's storage area, unpacks the batch, and loads the components into storage for the packing operation.
3. Containers arrive by conveyor – containers are transported from the Finishing Area to the Packing Area via a conveyor; thus, they are processed in a first-in, first-out manner.
4. Packing containers by type – a mix of components is packed into each container based on the container type. Component flow items are combined or packed into containers.
5. Packing containers by robot – a robot moves components from their storage area to containers. The robot also removes containers from the incoming conveyor and moves containers to the outgoing conveyor. Robots incur downtimes that are addressed by the Finishing Operators.
6. Check component inventory levels – assess the effectiveness of the components' replenishment parameters using graphs.

The base model for the additions described in this chapter is **Primer_8** that was saved at the end of Chapter 15. However, a copy of that file was saved as **Primer_9**; thus, we begin with that file.

16.1 Unpack components using the Separator object

Each batch of components, represented as a single item, is split into the number of components in a batch. This is accomplished by using the **Separator** object. Since the customized objects will be very similar, the first object is created and customized, i.e., for Component A, then it is copied, and the copy is edited for Component B.

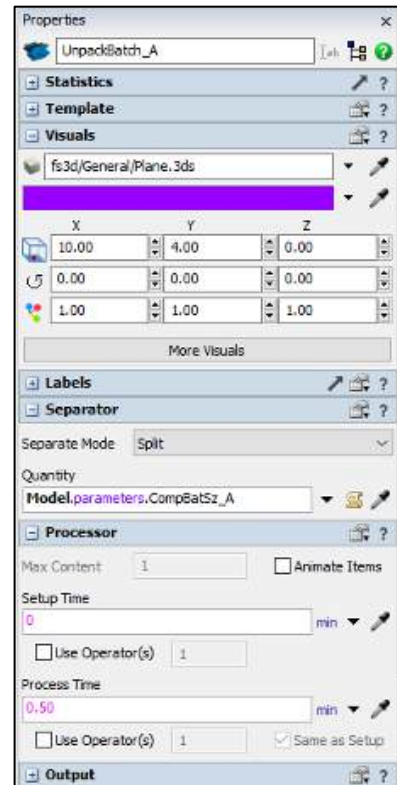
Create the first component's **Separator**.

Drag out a **Separator** object from the *Fixed Resources* section of the **Library** and place it near the first component's storage, StoreComp_A.

➤ Name the **Separator** *UnpackBatch_A*.

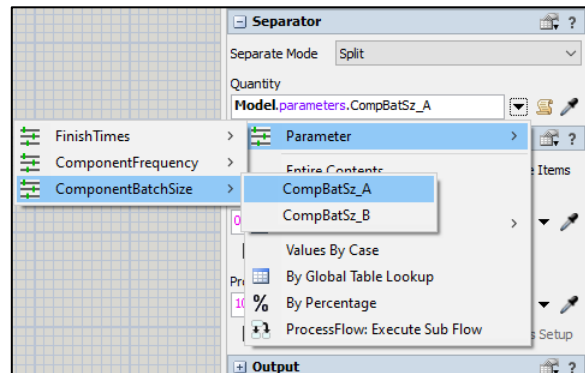
Modify the **Separator**'s **Visuals** pane as shown in the figure to the right and described below.

- Using the dropdown menu, set the 3D shape to a *Plane*. After the selection, the text box should contain *fs3d/General/Plane.3ds*, which is the *FlexSim* image file for a plane.
- For the color property, select *purple* from the color palette in the drop-down menu. The **Separator** is now the same color as the component it stores.
- Set the *x*, *y*, and *z* **Location** values to *15.00*, *4.00*, and *0.00*, respectively. Thus, the unpacking area is just behind the component's storage area.
- Set the *x*, *y*, and *z* **Size** values to *1.00*, *1.00*, and *1.00*, respectively.



Modify the **Separator**'s **Separator** pane as shown in the figure to the right and described below.

- Change the **Separate Mode** from *UnPack* to *Split* using the property's dropdown menu.
- Change the **Quantity** from *EntireContents* using the property's dropdown menu. Select *Parameter*, then *ComponentBatchSize*, then *CompBatSz_A*. This results in the property value being **Model.parameters.CompBatSz_A**, which is the current value stored in the **Model Parameters Table**.



While not used in this case, be aware that the **Separator** not only splits an item into multiple items but can unpack a container item that holds other items. This is analogous to unpacking the contents of a box. When in unpacking mode, the **Separator**, by default, routes the container out of Port 1, and the unpacked items are routed out of Port 2.

In addition to the splitting and unpacking modes, the **Separator** also acts as a **Processor**; i.e., it contains all of the capabilities of a **Processor**. After entering, an item is delayed for the time specified by the **Setup Time** and **Process Time**.

Therefore, modify the **Separator**'s **Processor** pane as shown in the previous figure above.

- Uncheck the **Animate Items** box, which by default is checked. If checked, items move across the object from one end to the other during the processing time. Since the batch of items does not move, we do not use the animation.
- Change the **Process Time** from the default value of 10 to 0.5 minutes; i.e., it takes about 30 seconds to unpack the three Component As in a batch.

Later in the primer, we'll use a label to store the batch size on the item and then calculate the processing time based on a constant plus so much time for each component.

Create the second component's **Separator**.

- Copy and Paste the **UnpackBatch_A Separator** by either of the following methods:
 - Right-click on the **Separator**, select **Edit** from the menu and then choose **Copy**. Right-click somewhere on the modeling surface, select **Edit** from the menu, then select **Paste**.
 - Select (click on) the **Separator**, press the **Ctrl-C** keys, click somewhere on the modeling surface, and press the **Ctrl-V** keys.
- Change the name of the **Separator** to **UnpackBatch_B**.

Modify the **Separator**'s **Visuals** pane as shown in the figure to the right and described below.

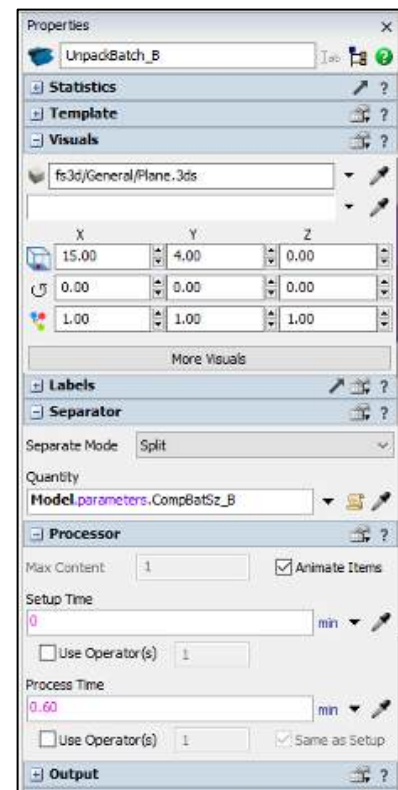
- Change the color property to *white* from the color palette in the drop-down menu. The **Separator** is now the same color as the component it stores.
- Set the **x**, **y**, and **z Location** values to 15.00, 4.00, and 0.00, respectively. Thus, the unpacking area is just behind the component's storage area.

Modify the **Separator**'s **Separator** pane as shown in the figure to the right and described below.

- Change the **Quantity** using the property's dropdown menu. Select **Parameter**, then **ComponentBatchSize**, then **CompBatSz_B**. This results in the property value being **Model.parameters.CompBatSz_B**, which is the current value stored in the **Model Parameters Table**.

Modify the **Separator**'s **Processor** pane as shown in the figure to the right and described below.

- Change the **Process Time** to 0.6 minutes; i.e., it takes about 36 seconds to unpack the five Component Bs, which is a little longer than it took for Component A.

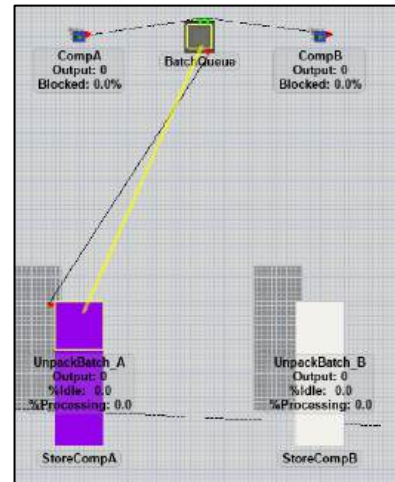


If you haven't already done so, save the model. Recall that it is good practice to save often.

16.2 Finishing Operator tasks

The Finishing Operator moves each batch of containers from the common batch queue to the container's storage area, unpacks the batch, and loads the containers into their storage area. This section primarily involves connecting objects.

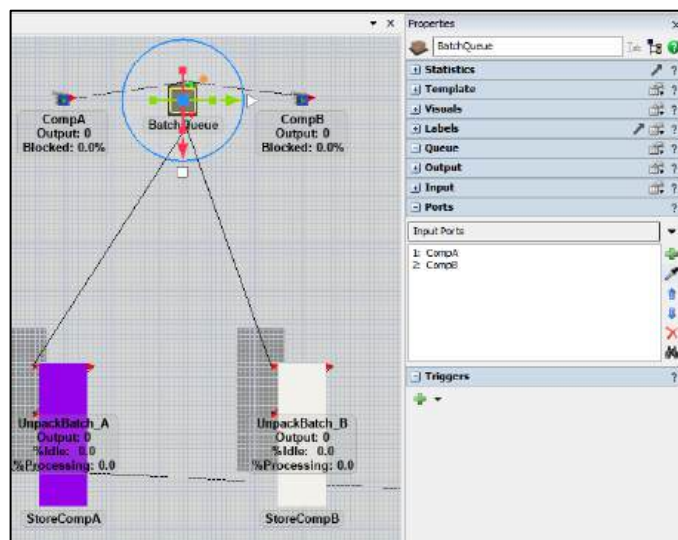
First, make the flow connections between the objects in the Packing Area using A-connections, i.e., connecting two objects while holding down the A key, as shown in the figure to the right. Recall the order in which the connections are made determines the direction of flow. Also, note that the A-connection creates the output port on the batch **Queue** and the input port on the **Combiner**.



Therefore, as shown in the figure to the right:

- Connect *Batch Queue* to *UnpackBatch_A*.
- Connect *Batch Queue* to *UnpackBatch_B*.
- Connect *UnpackBatch_A* to *StoreCompA*.
- Connect *UnpackBatch_B* to *StoreCompB*.

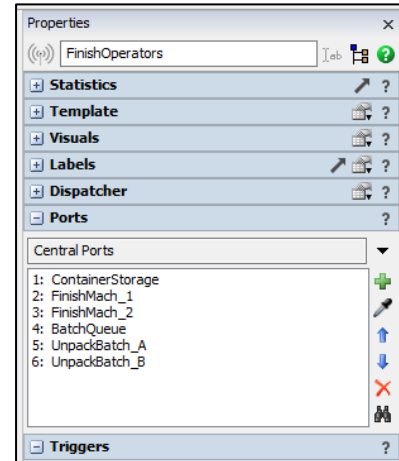
Note in the figure to the right, as shown in the **Ports** pane, the input ports for the selected object, *Batch Queue*, are the Sources *CompA* and *CompB*.



Next, connect the **Dispatcher** *FinishOperators* object with each object requesting tasks (transport and processing). These are communications connections or S-Connections, not flow connections, i.e., the objects' center ports are connected. Recall the order of connection does not matter in this type of connection.

Make the following connections. Once the connections are complete, the resulting **Ports** pane for the **Dispatcher** should resemble the figure to the right.

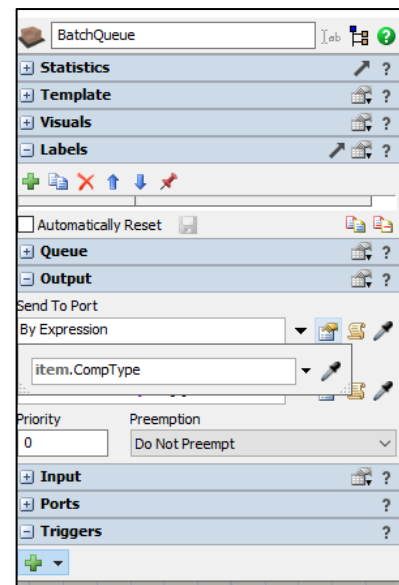
- S-Connect the Dispatcher *FinishOperators* to the *Batch Queue*.
- S-Connect the Dispatcher *FinishOperators* to each Separator *UnPackBatch_A* and *UnPackBatch_B*.



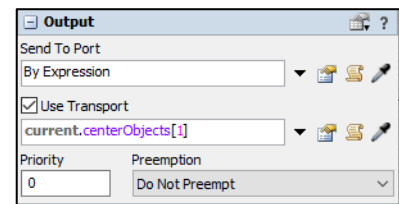
Next, associate the **Dispatcher** *FinishOperators* object with each object requesting tasks (transport and processing).

In the **Output** pane of the *BatchQueue* object, as shown in the figure to the right,

- Using the dropdown menu, change the **Send To Port** property from *First Available* to *By Expression*, then change the default *item.Type* to *item.CompType*. This will route a batch to the **Queue**'s port, whose number is the value of the component's type as defined by the label *CompType*.

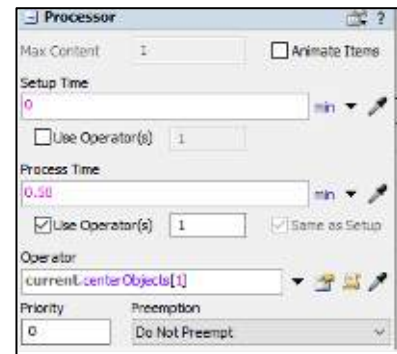


- Check the **Use Transport** box. See the completed **Output** pane in the figure to the right. Accept the default setting, *current.centerObjects[1]*, which means the **Queue** will choose the object connected to its first center port, the **Dispatcher**, to move the item to the next object.



In the **Processor** pane of the **Separator** *UnpackBatch_A*,

- Check the **Use Operator(s)** box below the **Process Time**.
- See the completed **Output** pane in the figure to the right. As with the **Queue** above, accept the default setting, `current.centerObjects[1]`, which means the **Separator** will choose the object connected to its first center port, the **Dispatcher**, to perform the separation operation.



Repeat the above steps in the **Processor** pane of the **Separator** *UnpackBatch_B*,

- Check the **Use Operator(s)** box below the **Process Time**.



If you haven't already done so, save the model. Recall that it is good practice to save often.

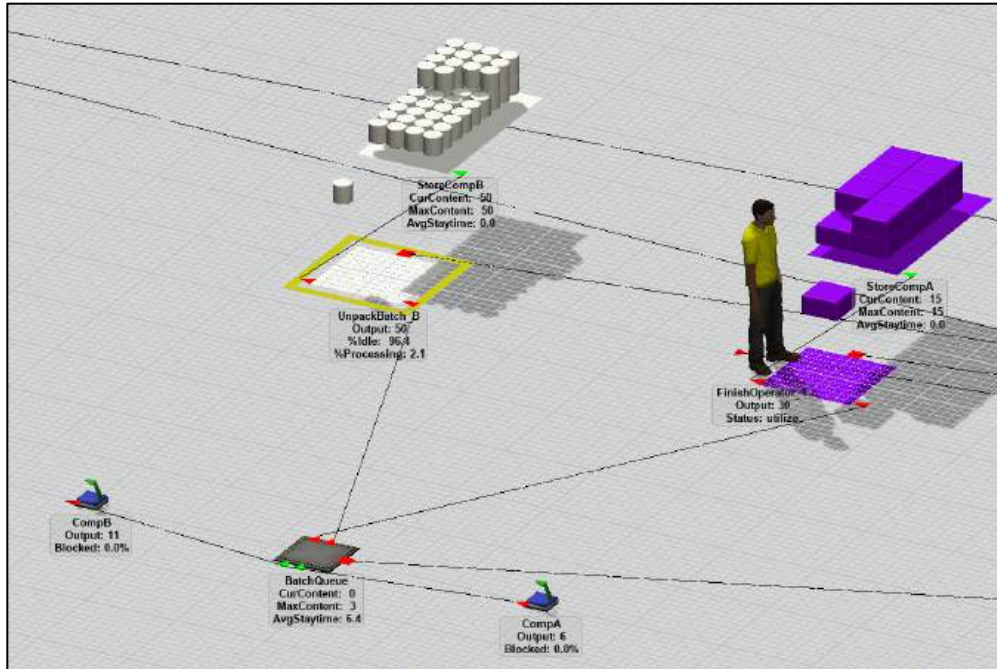
Reset and **Run** the model. Observe the activities of the Finish Operator.

- Slow down the **Run Speed** to about 0.5 using the slider or dropdown menu.

Note in the model as in the following figure.

1. The operator moves batches from the *BatchQue* to the correct **Separator** and then is delayed until the components are unpacked.
2. Components accumulate in their storage area. Currently, none are being consumed because the packing operation has not yet been modeled; that is coming in a subsequent step.
3. The location where each type of component is unpacked is a square on the floor, but the batch is positioned at about waist height (1 meter). In a later step, a non-functional visual object, a table, will be placed on the square on the floor, and the batch will appear as sitting on the table. The same type of object will be positioned under each component storage area.
4. The operator is unconstrained in its travel – it takes the shortest path between objects, ignoring any objects in the way on that path. This will be corrected later.

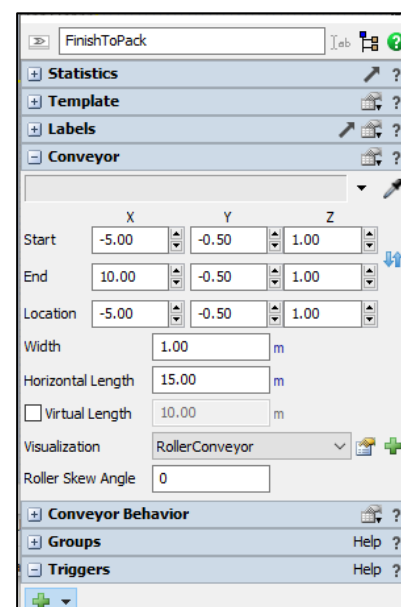
By default, **Task Executors** travel the shortest distance between objects and ignore any objects in the path. However, that can easily be corrected in one of two ways – using path networks to define travel paths or the A* tool that defines barriers or objects to be avoided and then finds the shortest path between objects by avoiding the barriers.



16.3 Containers arrive by conveyor

Just to get the Packing Area operational, we add a simple conveyor to move containers from the Finishing Area to the Packing Area. A more detailed conveyor system will be added later.

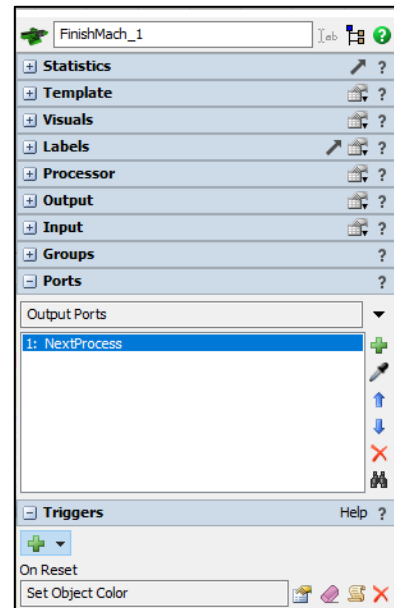
- Drag out a **Straight Conveyor** from the **Conveyor** section of the **Library**. Click near the Finishing Machines. This is the starting point of the conveyor; click near the Packing Area to end the conveyor. There is no need to be very precise because the object's size and position can be easily changed.
- Name the conveyor section *FinishToPack*.
- As shown in the figure to the right, reposition the conveyor using the parameters in the **Conveyor** pane of the *FinishToPack* object.



- Disconnect both Finishing Machines from the **Sink** *Next Process* by using the Q-Disconnect, i.e., hold down the Q key, select a Finish Machine **Processor**, and then select the **Sink**.

An alternative approach, as shown in the figure to the right, is to select the Finishing Machine *FinishMach_1*, then in the **Ports** pane, select *Output Ports*, then choose the port to delete, *1: NextProcesses* in this case, then select the  button to delete the port and thus the connection.

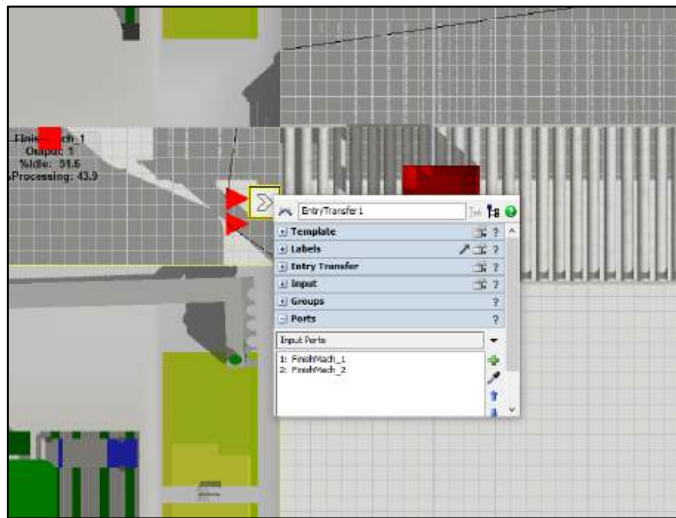
- Repeat for *FinishMach_2*.



- A-connect each Finishing Machine to the **Conveyor** *FinishToPack*.

Notice, as shown in the figure to the right, when objects are connected to a conveyor, a new object is automatically created, a transfer object. If an object provides input to a conveyor, the new object is an **Entry Transfer**, and if an object receives output from a conveyor, the new object is an **Exit Transfer**.

The figure also shows the object properties of an **Entry Transfer** and a red container being transported on the conveyor.



- A-connect the **Conveyor** to the **Sink** *NextProcess*. Note that an **Exit Transfer** is automatically created on the conveyor.



If you haven't already done so, save the model. Recall that it is good practice to save often.

16.4 Packing containers by type using a Combiner object

Now that the components have been created and are available near the packing station, they can be packed into containers. This is done through *FlexSim's* **Combiner** object, which is the object that puts components into containers. As the name indicates, it combines flow items. The **Combiner** is the “complement” of a **Separator** – the **Combiner** packs items into containers, and the **Separator** removes items from containers. The **Separator** also splits or copies items, as was described in an earlier section.

The **Combiner** object can operate in three modes specified by the *Combine Mode* property on the *Combiner* pane.

- *Pack*, the default mode option, allows items to be combined so they can be separated later. In this case, all flow items received through input ports 2 and above are placed in the item that enters port 1, referred to in general as the container item.
- *Join* combines the items permanently. The flow item entering port 1 is the only item that exits the **Combiner** once all of the components have been gathered.
- *Batch* releases all items that are collected based on the quantities specified in the *Components List*.

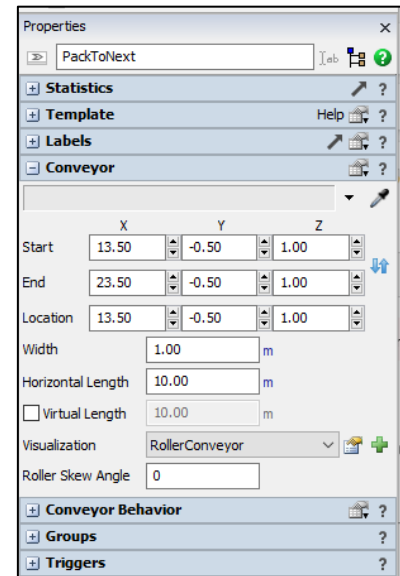
In terms of processing, the **Combiner** is similar to the **Separator** in that it contains all of a **Processor's** capabilities. Once all items that will be packed into a container have been collected by the **Combiner**, a delay is incurred that represents the time to combine the items. After the delay, the packed container is released to the next object. Like the **Processor**, the **Combiner** includes *Setup Time* and *Process Time*.

Add and customize the **Combiner** for the packing operation.

Drag out a **Combiner** object from the *Fixed Resources* section of the **Object Library** onto the modeling surface.

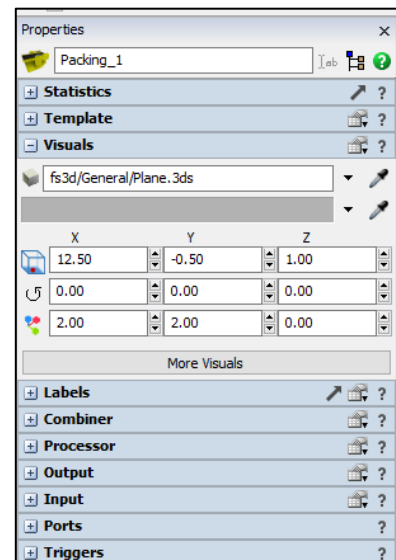
- Name the **Combiner**, *Packing_1*.
- Disconnect the **Conveyor** section prior to packing, named *FinishToPack*, from the **Sink** *NextProcess*. (Recall that disconnecting objects is done using either a Q-disconnect or the Ports list in the object.)
- A-connect the **Conveyor** to the **Combiner**.

- Drag out a new **Straight Conveyor** section and update its properties as shown in the figure to the right.
 - Name the conveyor section *PackToNext*.
 - Use the default **Horizontal Length** of 10.0.
 - Place the conveyor on the layout after the packing station. See the figure to the right for the settings.
- A-connect the **Combiner** to the new **Conveyor**, *PackToNext*.
- A-connect the **Conveyor**, *PackToNext*, to the **Sink** *NextProcess*.



As shown in the figure to the right, update the following properties in the **Combiner's Visuals** pane.

- Change the **Combiner's** 3D Shape property to *Plane.3ds* using the dropdown menu. Once the plane is selected from the dropdown list, the value should be *fs3ds/General/Plane.3ds*
- Change the **Combiner's** size to $x=2.00, y=2.00, z=0.00$
- Change the **Combiner's** location to $x=12.50, y=-0.50, z=1.00$. This should position the packing station between the two conveyors. Of course, the precise location can be adjusted once all of the objects are in place. The z value of 1.00 positions the plane at conveyor height.
- Change the color property from yellow to *gray*.



Connect the input objects associated with the **Combiner**, i.e., those that are being combined.

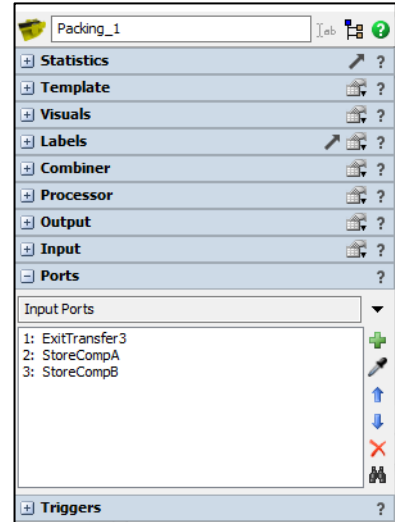
- Make an A-connection from the **Queue** *StoreCompA* to the **Combiner** *Packing_1*.
- Similarly, make an A-connection from the **Queue** *StoreCompB* to the **Combiner** *Packing_1*.

The first input connection to a **Combiner** must be the object that supplies the container items, those in which other items are packed. For each additional connection that is made to a **Combiner**, a row is added to its **Components List**, which is on the **Combiner** pane. The list is like a recipe for what to combine with the container. The **Target Quantity** is the number of items to collect from each port.

In our example, input Port 1 receives the container arriving on the **Conveyor** *FinishToPack*; Ports 2 and 3 receive items from two **Queues**, *StoreCompA* and *StoreCompB*, respectively. The default quantity is one item from each, which can be edited in the **Components List**. However, in our case, the list will vary by container type and thus must be updated dynamically based on the type of container that enters the **Combiner**.

One way to verify the port connections on any object is to examine the **Ports** pane of an object. The figure to the right shows the input to the **Combiner** from ports 1, 2, and 3, which are from the *ExitTransfer3*, *StoreCompA*, and *StoreCompB*, respectively.

Also, the ports (Input, Central, and Output) can be reordered by using the **Rank** controls and deleted with the **Delete** control; both are located to the right of the Ports list. If there are many connections to remove, it is much easier to do it from this interface than numerous Q-disconnects with the mouse.

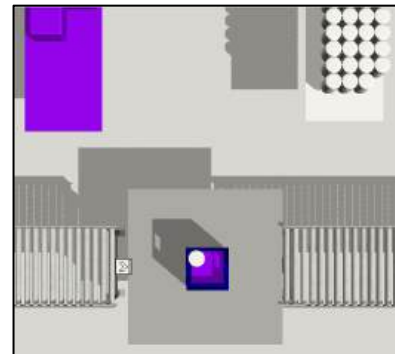


Verify that containers move from the Finishing Area, are packed with components, and move on to the next process.

➤ **Reset** and **Run** the model and observe the basic flow.

If the model is not behaving as planned, recheck the steps above.

Note, as shown in the figure to the right, every container is packed with one of each component. This is not what will occur in the real system, but at least the basic packing operation works. The number of components per container is addressed in the next step.



A trigger on the **Combiner**, along with a **Global Table**, is used to update the **Components List** dynamically.

The recipe for each container type is specified in a **Global Table** named *Packing*. As shown in the figure to the right, the table has two rows and three columns.

	1-Red	2-Green	3-Blue
Comp A	2	0	1
Comp B	0	4	4

The rows correspond to the rows in the combiner's **Component List**. In this case, they refer to Port 2 (CompA) and Port 3 (CompB).

The columns represent the Type of the container. In this case, the container has Type values 1, 2, or 3.

Therefore, the cells in the table are the **Target Quantities** for the **Combiner**. For this example, container Type 1 receives only two CompA, Type 2 receives only four CompB, and Type 3 receives one CompA and four CompB.

Note the row and column headers (*Row 1, Row 2, Col 1*, etc.). These are text fields that are useful for annotating the table - the model does not use them, but they clarify what information is stored in the table.

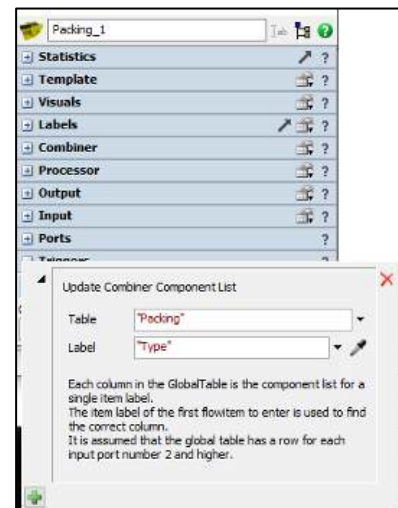
➤ Create a **Global Table** named *Packing* and edit the cells and headers to correspond to the figure above.

It would be very easy to expand this table to include many more product types (containers) than the three included in this example model and more types of components. Also, as will be discussed later, the table could be imported from *MS Excel*.

An **OnEntry** trigger on the **Combiner** is used to instruct the **Combiner** to use the recipes specified in the **Global Table**.

- Using the  button, select **On Entry**.
- Using the  button to the right of the **On Entry** textbox, select the menu option *Update Combiner Component List With Labels*.
- In the resulting interface titled *Update Combiner Component List*, as shown in the figure to the right,
 - Set the **Table** value to "*Packing*" using the dropdown menu and select from the list of **Global Tables**.
 - Retain the default value for **Label**, "*Type*". Each container has the value of the label *Type* set when it is created at the **Source**, which, in this case, is a random sample from an **Empirical Distribution**.

Thus, when a container enters the **Combiner**, its **Type** label value is checked, and the **Combiner's Component List** is updated based on the row values in the Global Table *Packing* in the *Type*-valued column.



Once the **Combiner** collects all of the items it needs from all ports, it invokes a processing time that is specified in the **Process Time** property in the **Processor** pane. This is the same as in the **Processor** object. The process time represents the combining time, such as packing, assembly, etc. The default time is a constant 10 time units. The processing time starts once the **Target Quantity** for all of the items specified in the **Components List** has been collected.

The processing time will also vary since a container's type and number of components vary. The type of container is known based on the label named **Type**. We could use the “By Case” construct to set the processing times in the Finishing Machines. However, we take an alternate approach here and use a **Global Table**. The advantage of using a **Global Table** is that changing the process times only requires changing a table value and not values within objects.

Set up a **Global Table** to store process times. The table will include finish times and packing times so that defining the processing times at each Finishing Machine can be changed to use the table data-storage approach. It is best to set up a **Global Table** before it is referenced in a model so that its name can just be selected from a list of tables and not have to type out the name, which may be entered in error.

Not counting the time to move components from storage to the packing table (**Combiner**), which will be accounted for in the **Robot** object, it is estimated that it takes about 0.60, 0.50, and 0.75 minutes to pack container types 1, 2, and 3, respectively.

Add a **Global Table** to the model from the **Toolbox**, as shown in the table to the right. Again, this contains both the processing and packing times.

- Name the table *ProcessTimes* and set **Rows** to 3 and **Columns** to 2.
- Name the row and column headers as shown in the figure.
- Enter the cell values as in the figure.

Model		ProcessTimes	
	FinishTimes	PackingTimes	
Type 1	15	0.60	
Type 2	20	0.50	
Type 3	30	0.75	

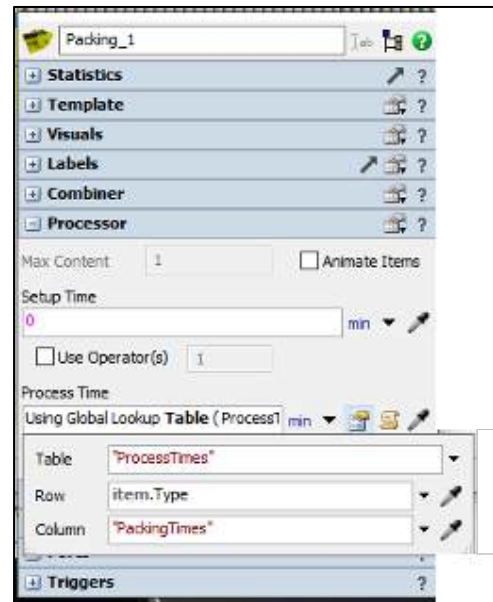
Note that when selected, the line that divides the rows or columns in the headers turns into an arrow that allows changing the row and column sizes.

Change the **Process Time** property on the **Combiner** to reference the *ProcessTimes* table.

➤ In the **Processor** pane, using the dropdown menu to the right of the **Process Time**'s textbox, select the *By Global Table Lookup* option and complete the interface as shown to the right and described below.

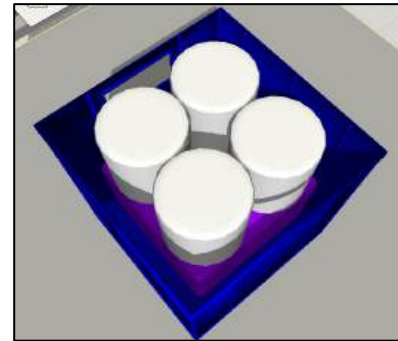
- For the **Table** property, select *ProcessTimes* from the drop-down list of **Global Tables**.
- The default **Row** property is not changed since it uses the container's label **Type** for the lookup.
- Change the **Column** property from 1 to 2 so that the values in the second column of the table are used.

Alternatively, enter the header name of the column in quotes, i.e., "PackingTimes." This way, if the columns are rearranged, this value does not need to be changed at all of the packing stations. Of course, if the column header name is changed, the objects will need to be updated.



➤ **Reset** and **Run** the model to verify that the containers are packed with the correct components and that the packing times vary by container type. Container type 3 is shown in the figure to the right; it contains one purple Component A (At the bottom of the container) and four white Component Bs.

One way to observe the contents of the containers after packing is to temporarily disconnect the **Sink NextProcess** from the **Conveyor PackToNext**. This causes packed containers to back up on the conveyor so they can be checked.



If you haven't already done so, save the model. Recall that it is good practice to save often.

16.5 Placing components in containers by robot

The packing station is automated. It uses a robot to move the required mix of components into the containers. The robot's tasks are:

- Move each empty container from the incoming conveyor onto the packing table.
- Move the proper components from their storage location to the container.
- Move each packed container to the conveyor that goes to the next process.

This section has two subsections; the first defines the robot's operations, and the second defines downtime and repairs of the robot.

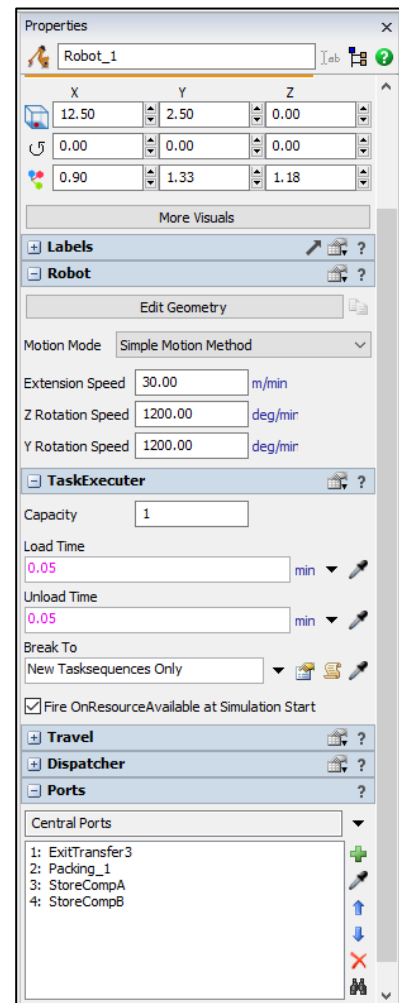
16.5.1 Defining the robot's operations

This section defines the properties of the **Robot** object to model the proposed packing operation at DPL.

- Create a **Robot** object by dragging its entry in the *Task Executors* pane of the **Library** to a location between the two component storage areas.
- Name the object *Robot_1*.
- Set the following property values in the object's *Visuals* pane, as shown in the figure to the right.
 - The specified location values should position the **Robot** between the two storage areas and close enough to the packing table.
 - When setting the size, all that is needed is the *x* value; the software automatically sets the *y* and *z* values based on the *x* value.

DPL has yet to develop the specifications of the packing robot. In *FlexSim*, the **Robot** object has four motion alternatives (the *Motion Mode* property) for the robot's operation. The manufacturing engineers reviewed the options and suggested using the *Simple Motion Method* and its default values. The **Robot** object can model many aspects of robots, but that detail is not needed for this example. Also, an explanation of the various modes is beyond the scope of the primer.

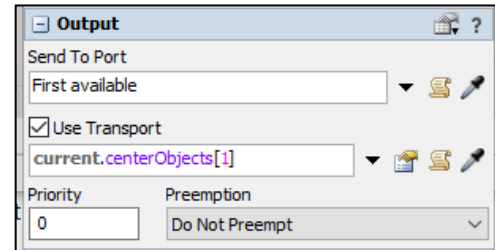
- Therefore, as shown in the figure to the right, in the *Robot* pane, select the *Simple Motion Method* option for the *Motion Mode*.
- As shown in the figure to the right, in the *TaskExecutor* pane, change both *Load Time* and *Unload Time* to 0.05 minutes (3 seconds); i.e., it is assumed that it takes 3 seconds for the **Robot** to secure an item to move and the same to release an item once moved.
- Connect the center ports of the following objects that use the **Robot** to *Robot_1* using S-Connects. The result should be what is shown in the *Ports* pane in the figure to the right



- **Exit Transfer** on the **Conveyor** *FinishToPack*.
- **Combiner** *Packing_1*, the packing operation.
- **Queue** *StoreCompA*, Component A's storage area.
- **Queue** *StoreCompB*, Component B's storage area.

➤ Check the *Use Transport* box in the **Output** pane of the following objects that call the **Robot** to move items. Use the default transport reference, `current.centerObjects[1]`, which is the object connected to the current or selected object's center port. An example is shown in the figure to the right.

- **Exit Transfer** on the **Conveyor** *FinishToPack*.
- **Combiner** *Packing_1*, the packing operation.
- **Queue** *StoreCompA*, Component A's storage area.
- **Queue** *StoreCompB*, Component B's storage area.



16.5.2 Downtime in the Packing Area

The only source of downtime in the Packing Area is the **Robot**. The downtimes are random failures that the Finishing Operator addresses. The times between failures and the times to repair are both considered random variables or probabilistic and thus are based on probability distributions.

Defining this part of the model is the same as the process for defining the failures and repairs on the Finishing Machines using the **MTBF MTTR** tool (section 14.5 of the primer). The process for doing so is repeated here with the estimated values for the robot; however, refer to the previous example in section 14.5 for definitions and explanations.

Since the packing robot has yet to be selected and downtime can significantly affect system performance, the distributions and parameters represent initial engineering estimates. Therefore, as more information becomes available on the robots, the parameter values used in the simulation need to be revised.

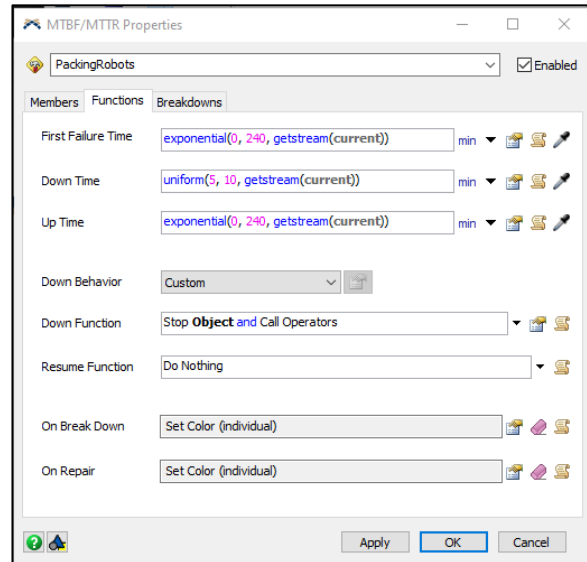
➤ In the **Toolbox**, press the  button and again select the **MTBF MTTR** tool from the drop-down menu.

Modify the **Members** tab as follows.

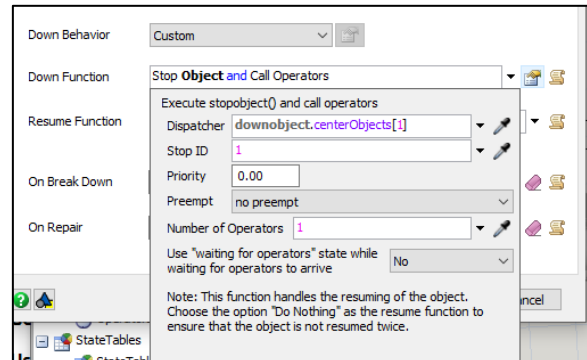
- Change the name of the downtime to *PackingRobots*.
- On the Members tab, use the  button and select the **Robot** *Robot_1*.

Recall that the **Functions** tab defines how often downtimes occur and their duration. It also defines what actions occur in the simulation whenever a downtime occurs.

- Change the values for the following properties, as shown in the figure to the right.
 - **First Failure Time** to an exponentially distributed random variable with a mean of 240 minutes (4 hours).
 - **Down Time** to a uniformly distributed random variable with values between 5 and 10 minutes.
 - **Up Time**, same as **First Failure Time**, exponentially distributed with a mean of 240 minutes (4 hours).



- Change the **Down Function** to *Stop object and call operators* via the property's drop-down menu. As shown in the figure to the right, all default values in the interface are used.
- Change the **Resume Function** value to *Do Nothing* via the drop-down menu. The *Stop object* option for the **Down Function** handles the **Resume Function** logic.



The **On Break Down** and **On Repair** triggers are used to change the color of the object when it is down.

- For the **On Break Down** trigger, select **Set Color (individual)**, then change the **Color** value to *Color.red*, as shown in the figure to the right.
- Similarly, for the **On Repair** trigger, select **Set Color (individual)**, then set the **Color** value back to orange using the *Color.orange* option.

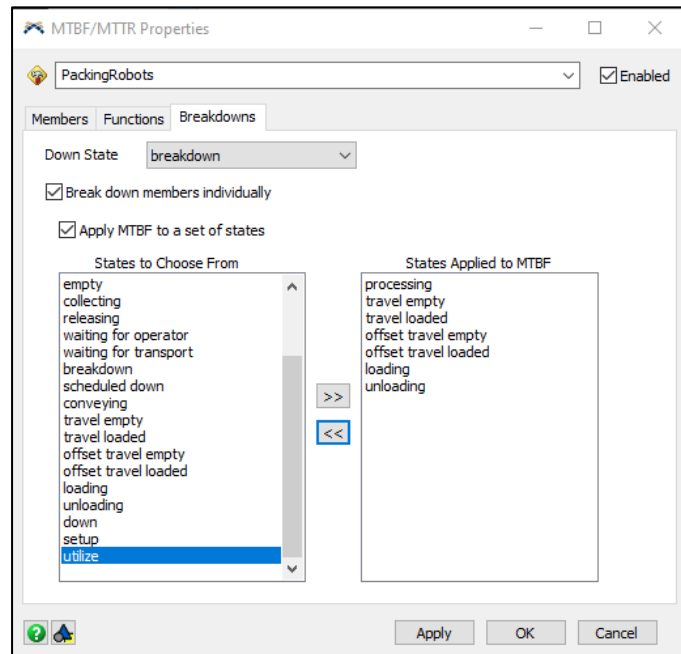


As with the Finishing Machines, the time between failures (downtimes) is based on the object's state and not on the simulation clock. The robots accrue downtime only when they are performing tasks. However, the tasks are more extensive than those used in the fixed resource objects, like **Processors** and **Combiners**. A **Robot** is a **Task Executer** object. While it does not move like an **Operator** or **Transporter** object, i.e., its base does not move, its arm moves to transport items between objects. Therefore, it accrues downtime in more states. Those states are identified in the figure below.

The above situation is defined on the **Breakdowns** tab and shown in the right figure.

- Check the box **Apply MTBF to a set of states**.
- Move states from the **States to Choose From** list to **States Applied to MTBF** using the >> move button.

Multiple states can be selected by holding down the shift key when selecting.



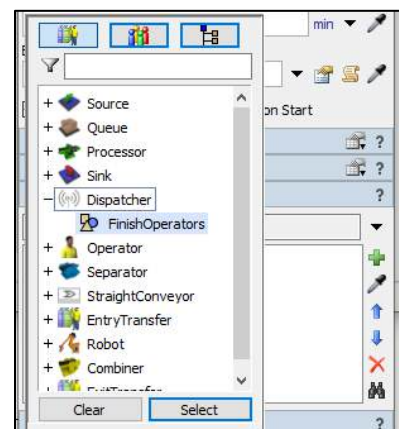
If a robot is down at the end of a simulation, its color remains red. Therefore, on the **Robot**

- Create an **On Reset** trigger to change the color back to orange by clicking the  button under **Triggers** and selecting *On Reset*.
- Using the  button to the right of the **On Reset** textbox, select *Set Object Color*.
- In the **Set Object Color** interface
 - Change **Object** to *current*.
 - Change Color to *Color.orange*.

Since a Finishing Operator is used to repair a robot, the two objects need to be able to communicate.

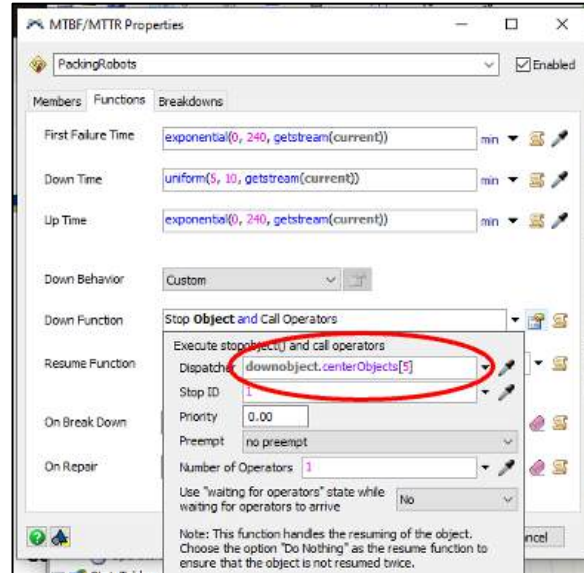
- Connect the **Dispatcher** *FinishOperators* to the **Robot** *Robot_1* using a center port connection, i.e., an S-Connection.

This can also be done in the **Ports** pane of the **Robot**, as shown in the figure to the right. Use the  button, then select the **Dispatcher** *Finish Machines* from the menu.



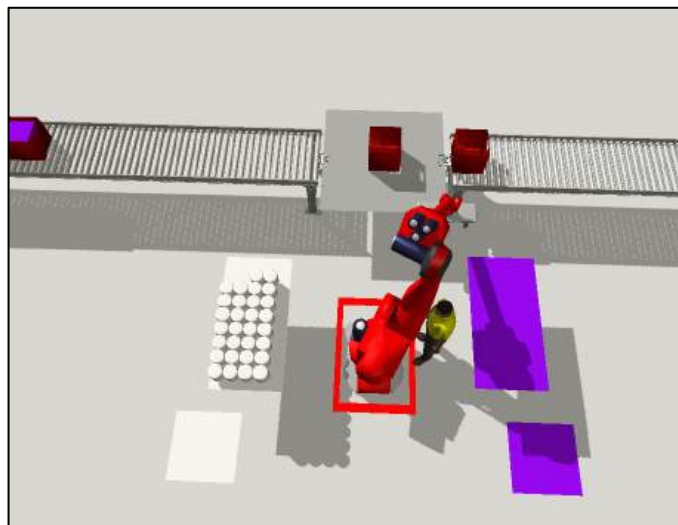
The **Dispatcher** is now the fifth object to be connected to the **Robot** with center ports, i.e., the **Robot** now has five center ports. Therefore, as shown in the figure to the right and explained below, change the port reference to the **Dispatcher**.

- Go to the **Functions** tab on the **MTBF/MTTR Properties** window for *PackingRobots*. Select the **Down Function** to obtain the interface shown in the figure. Change the port number reference in the brackets in the *FlexScript* command to 5, as highlighted by the red oval in the figure to the right.



The screenshot in the figure to the right shows a robot in the breakdown state (colored red), and the finishing operator is performing the repair.

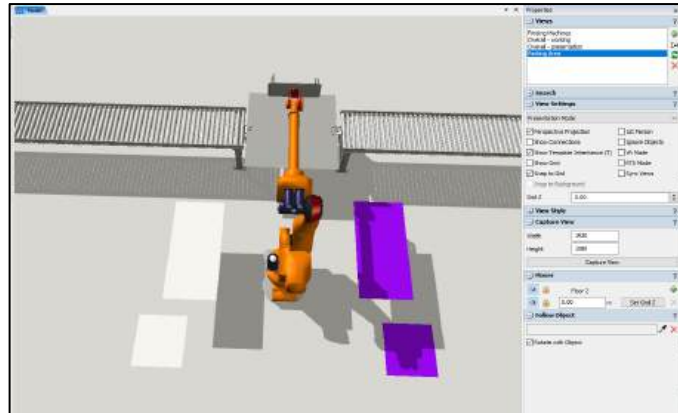
Also, note the red container that is on the packing table was being packed when the robot breakdown occurred.



Now that at least the initial modeling of the Packing Area is complete, create a model view of the Packing Area similar to the one shown above. Also, update the overall model view to include the Packing Area. Recall that this is done through the **View** pane of the **Properties** window when the cursor is clicked anywhere on the modeling surface but not on an object.

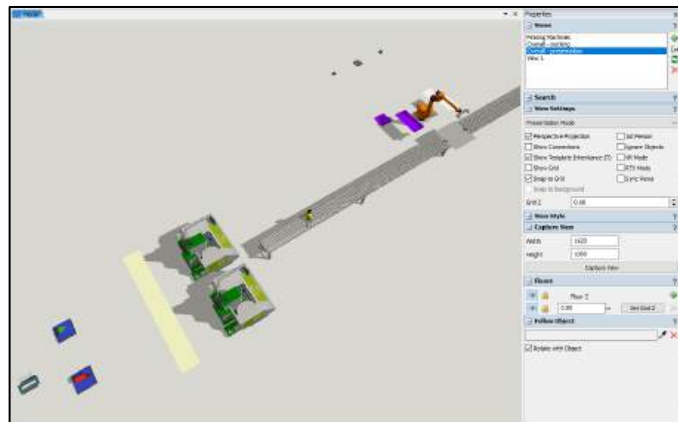
Create the view of the Packing Area, as shown in the figure to the right

- Orient the model to focus on the Packing Area, similar to the screenshot to the right.
- Press the  button in the righthand portion of the **Views** pane.
- Rename the view *PackingArea*.



Update the overall model view, as shown in the figure to the right

- Select the view *Overall – presentation*.
- Zoom out and arrange the model until all of the objects are visible including the Packing Area.
- Press the refresh button  in the righthand portion of the **Views** pane.
- Repeat the above three steps for the *Overall – working* model view.



If you haven't already done so, save the model. Recall that it is good practice to save often.

16.6 Check component inventory levels

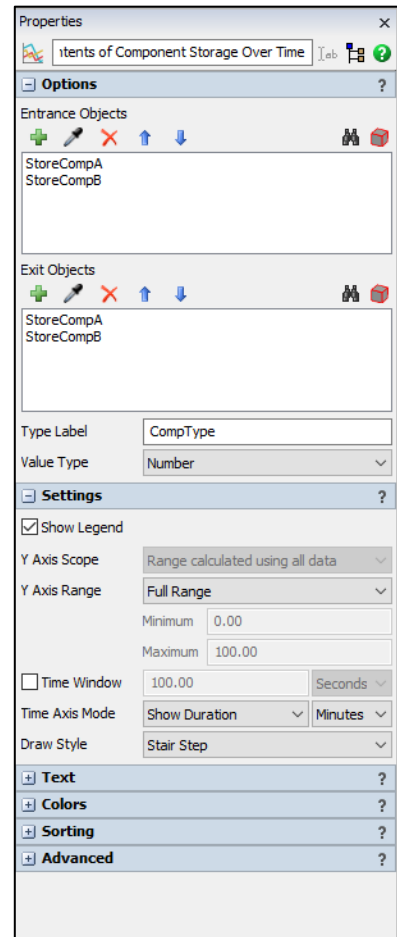
An important system design issue is how often components in the packing area are replenished and in what quantity. One means to assess this is through plots of the inventory level over time.

Therefore, create a Dashboard that displays the contents of the component queues (storage areas) over time.

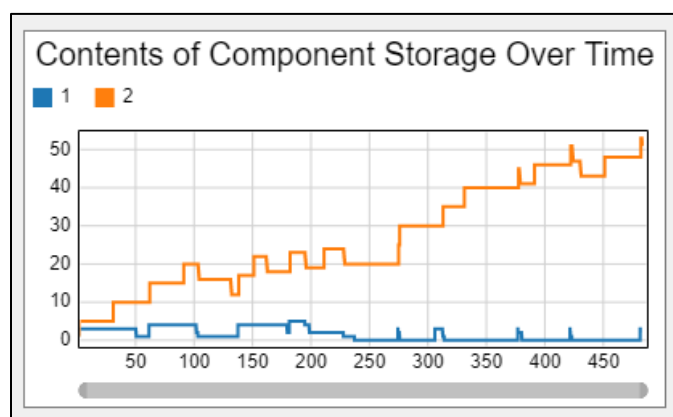
- Create a new dashboard by using the **Dashboard** button on the **Main Menu** bar or through the **Toolbox**.
- Name the Dashboard *Components*.
- From the **Content** pane in the **Dashboard Library**, select *WIP By Type* and then *Line Chart*, then click on the dashboard. Using the chart's handles, resize the graph to fit the window.

As shown in the figure to the right

- Name the chart, *Contents of Component Storage over Time*.
- Using the  button in the **Entrance Objects** section of the **Options** pane, add two objects, both component Queues, *StoreCompA* and *StoreCompB*.
- Repeat the above step for the **Exit Objects** section of the **Options** pane.
- On the **Settings** pane, change **Time Access Mode** to *Show Duration* and set the time units to *Minutes*.



- **Reset** and **Run** the model. The contents of the component storages should be similar to the plots shown in the figure below.



Based on this short run of about eight hours (480 minutes), it appears that the trend in the number of Component Bs waiting to be packed in general continues to grow over time. Also, the inventory of Component As remains quite low, and there is quite a bit of time when the inventory is zero, which may cause the packing station to have a lot of idle time due to not having Component A.

Therefore, the time between batches and/or the batch size should be adjusted. The simulation model can be used to decide the best settings. Also, in the model there is no limit on the number of components that can be stored before packing. The physical space limitations need to be considered.



If you haven't already done so, save the model. Recall that it is good practice to save often.



Use the ***Save Model As*** option in the **File** menu to make a copy of the existing model so that it can be customized beginning in the next section. Again, you can use any file name, but the next model is referred to as Primer_10 in the primer.

PART V – RESOURCE TRAVEL, EXPERIMENTATION, CONVEYORS, AND LISTS

The primer's simulation model is further developed by exploring and using several powerful features of *FlexSim* – alternative means to control task executor travel, experimentation and the analysis of scenarios, the use of conveyors to transport items, and the use of the List tool to implement more complex routing logic.

- Chapter 17 considers two ways to control Task Executor travel: via a path network and the A* Navigator.
- Chapter 18 introduces *FlexSim*'s Experimenter, which provides a convenient means to consider simulating multiple scenarios and replicating each scenario. The Experimenter is used to study two aspects of the system modeled so far – (1) the effect of the size of the buffer of containers prior to Finishing on performance and (2) the effect of component replenishment plans on performance.
- Chapter 19 introduces the Conveyor objects and adds conveyors in the model to transport containers between the Finishing and Packing Areas and after packing to the warehouse.
- Chapter 20 adds more complex routing logic in the Finishing Area regarding how containers are selected for finishing. The logic is implemented using the List tool.

17 MANAGING OPERATOR TRAVEL

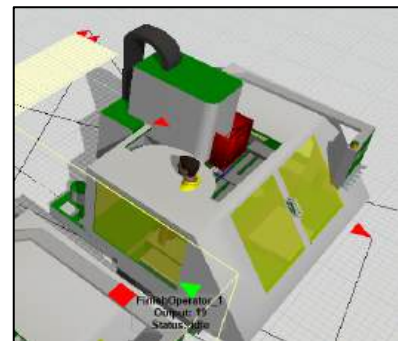
Chapter 17 considers two ways to control Task Executer travel: via a path network and the A* Navigator.

The base model for the additions described in this chapter is **Primer_9** that was saved at the end of Chapter 16. However, a copy of that file was saved as **Primer_10**; thus, we begin with that file.

When running the model, you have most likely noticed some “odd” **Task Executer (Operator)** behavior, mostly resulting from the use of default behaviors, i.e., not further customizing the object. As pointed out in this primer, at least for beginners, models are built starting from a very simple representation, with many assumptions, and then evolving the level of complexity to where it needs to be in order to address the issues simulation is meant to tackle. With this approach, it is expected that objects or sections of a model will be updated later in the model-development process.

One of the odd behaviors is that as the **Task Executer, Operator** *FinishOperator_1*, moves around in the model, it has no regard for other objects. By design, when **Task Executors** start a travel task, they determine the shortest distance between their current location and the location of the requesting object. In many cases, this behavior is okay, but it becomes problematic when there are barriers in the **Task Executer’s** path, which the object ignores, which is the case in this example. Therefore, the **Task Executer’s** travel path needs to be controlled.

Controlling the travel path affects the model’s aesthetics - it certainly doesn’t help build layperson confidence in a model when people and equipment walk through machines, conveyors, walls, etc. An example is shown in the figure to the right, where the Finish Operator waits for its next task at a Finish Machine, but it appears the operator is inside the machine.



Controlling travel paths also affects estimated system performance – avoiding barriers increases travel distance. This additional distance adds time to travel tasks, which decreases resource utilization because they have to wait longer to be served.

Two options for controlling a Task Executer’s travel path are discussed – using **Path Networks** and **A* Navigation**. Basically, a **Path Network** defines where a **Task Executer** can go, and **A* Navigation** defines where a **Task Executer** cannot go.

17.1 Controlling task executor travel with path networks

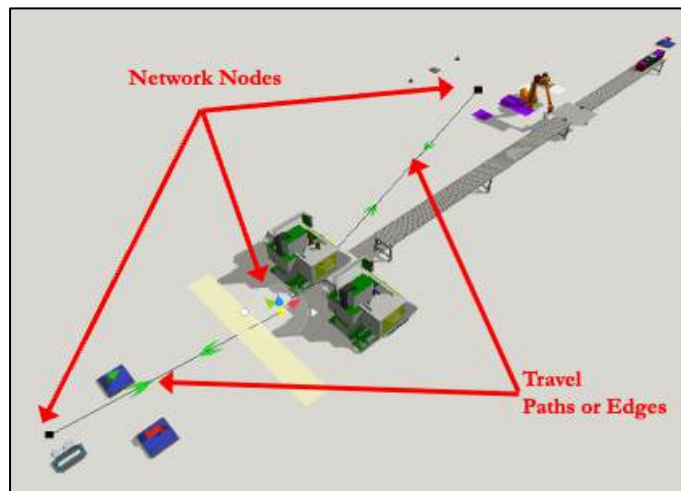
Again, by default, when **Task Executors (TEs)** travel between two objects in a model, they use the straight-line path and are not inhibited by other objects. To better match reality and to provide more realistic performance measures, **TEs** can be constrained to defined paths that are formed by what is referred to as a **path network**, which is similar to a roadway. A path is defined by connecting a series of nodes, referred to as **Network Nodes (NNs)**. The nodes are objects located in the **Travel Networks** section of the **Object Library**, just below the **Task Executors** section.

Only a simple network is constructed here since, in the next section, a different means of controlling **TE** travel, the **A* Navigator**, is used. The **Operator** will travel to three locations in the model – the Finishing Area, the Packing Area, and the Break Area, where the operator goes when on break. Therefore, travel is controlled with three nodes, one at each location. In this simplified approach, the Finish Operator will not travel to each object, just to a general centralized location in each area. If the **Path Network** approach were used extensively in this model, the network would use more **Network Nodes** and be more complex.

Construct the simple Path Network to be introduced to its basic constructs and how it works.

- Drag three **Network Nodes** from the **Library** to the modeling surface and place them in the locations indicated by the red arrows in the figure to the right. **Network Nodes** are small square black objects.

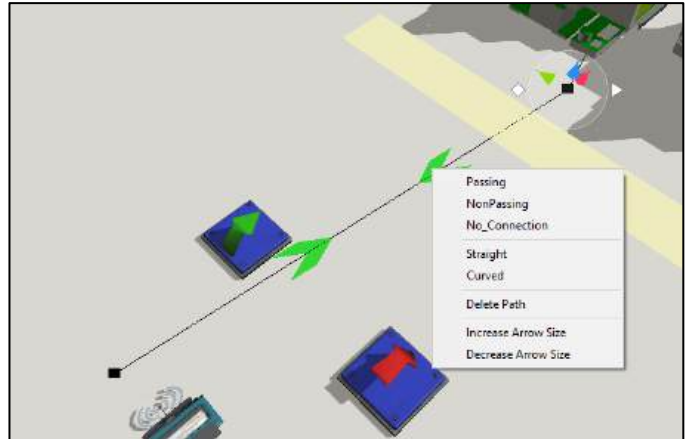
To get the exact placement of the **Network Nodes**, you may uncheck the **Snap To Grid** box in the **View Settings** pane in the **Properties** window. Otherwise, nodes will be placed on the closest grid point. Recall that this interface is accessed when clicking anywhere on the modeling surface where there is no object.



- Since it is good modeling practice to name objects, name the **Network Nodes**: *nn_FinishingArea*, *nn_PackingArea*, and *nn_BreakArea*. No other properties need to be modified.
- As shown in the figure above, connect the nodes with A-connections, creating travel path segments, also called *edges*. These are straight-line paths, but they will be changed to curved paths to avoid objects

The green arrows on the edges denote the permitted direction of travel. By default, the edges are bidirectional. As shown in the figure to the right, right-clicking on an arrow provides the following options:

- **Passing** permits multiple **TEs** on the same path to pass or move past each other. This is the default.
- **NonPassing** restricts faster **TEs** from passing a slower **TE** on the same path.
- **No_Connection** restricts the direction of travel.
- **Straight** declares edges, or travel paths between nodes, to be straight lines. This is the default.
- **Curved** declares edges, or travel paths between nodes, to be curves that can be adjusted.
- **Delete Path** eliminates an edge between nodes.
- **Increase Arrow Size** makes the directional arrows larger.
- **Decrease Arrow Size** makes the directional arrows smaller.



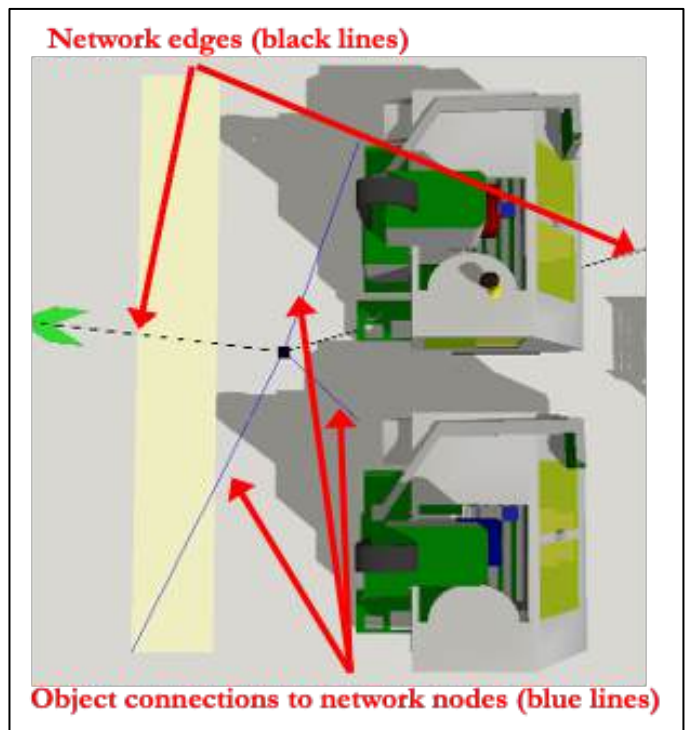
For a network to function, fixed objects must be associated with **Network Nodes**. This is so that a **TE** knows where an object is located on the network. For example, if an **Operator** is to travel to a **Queue** in order to pick up an item, and the **TE** must stay on the network path, it must know which node to travel to in order to reach the **Queue**.

Objects are associated with **Network Nodes** by making an A-connection from the node to the object. When connected, a blue line denotes the connection, as shown in the figure to the right.

The figure to the right distinguishes between the black lines that are network edges and blue line which associate an object with a network node.

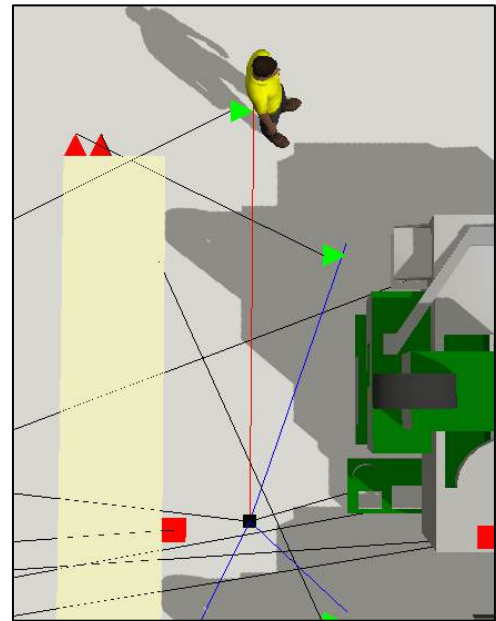
A single object can be associated with multiple nodes; e.g., if an object can be accessed from more than one side or if an object is associated with multiple networks.

Similarly, a single node can be associated with multiple objects, e.g., if a **TE** interacts with a group of objects from a common point. This approach is used since it makes the network quite simple. Recall that the **Path Network** approach is only used for informational purposes; another approach will replace it in the next section.



- Connect each network node to its associated object(s) as defined below. The second case is shown in the figure above.
 - From **Network Node** *nn_BreakArea* to
 - **Dispatcher** *FinishOperators*.
 - From **Network Node** *nn_FinishingArea* to
 - **Processor** *FinishMach_1*
 - **Processor** *FinishMach_2*
 - **Queue** *ContainerStorage*.
 - From **Network Node** *nn_PackingArea* to
 - **Queue** *BatchQueue*
 - **Separator** *UnpackBatch_A*
 - **Separator** *UnpackBatch_B*
 - **Robot** *Robot_1*.

To use a network, each **TE** must be associated with the network. This is accomplished by an A-connection between the **TE** and only one of the **Network Nodes**. This is considered the **TE**'s "home" node – where it will be located when the model is **Reset**. The resulting connection between the **TE** and the **Network Node** is indicated by a red line, as shown in the figure to the right.

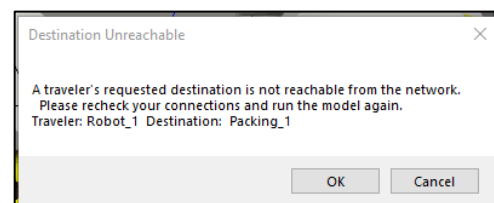


- Connect the **Operator** *FinishOperator_1* to the **Network Node** *nn_FinishingArea*.

When a model is run, an error will result if any of the objects where a **TE** needs to perform a task is not associated with a network node.

In this case, the error message shown to the right results because the **Network Node** *nn_PackingArea* has not been connected to **Combiner** *Packing_1*.

- Therefore, make an A-Connection from *nn_PackingArea* to the **Combiner** *Packing_1*.



Note this should have been in the list of connections above for **Network Node** *nn_PackingArea*, but was not to demonstrate what happens when a needed connection is not made and an error results.

Since the **Robot** needs to be on the network so the **Operator** can repair it, the objects the **Robot** serves must also be on the network.

- Therefore, connect *nn_PackingArea* to.
 - **Conveyor's Exit Transfer** *ExitTransfer3*
 - **Conveyor's Entry Transfer** *EntryTransfer2*

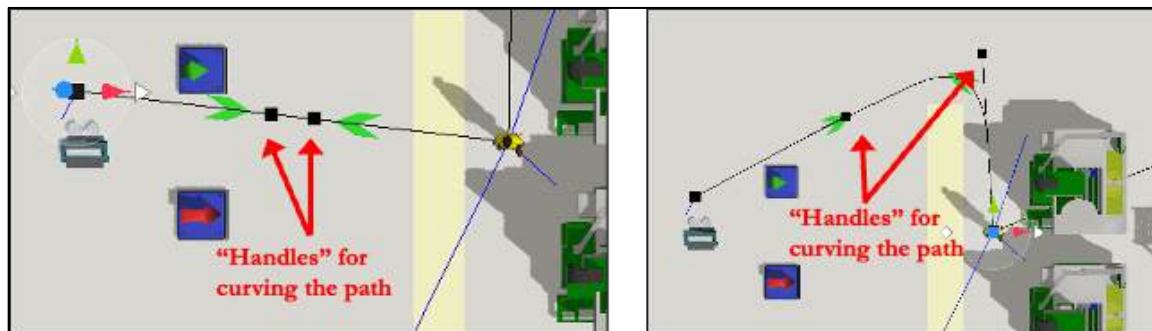
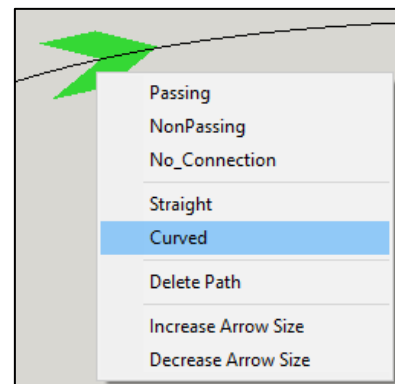
Note this should have been in the list of connections above for **Network Node** *nn_PackingArea*.

If a **Task Executer** is not connected to the network, then the **TE** will not travel on the network, but there will be no error messages. However, it should be obvious by watching the model that the network paths do not constrain the TE.

For the Operator to avoid other objects, the network paths between areas need to be curved.

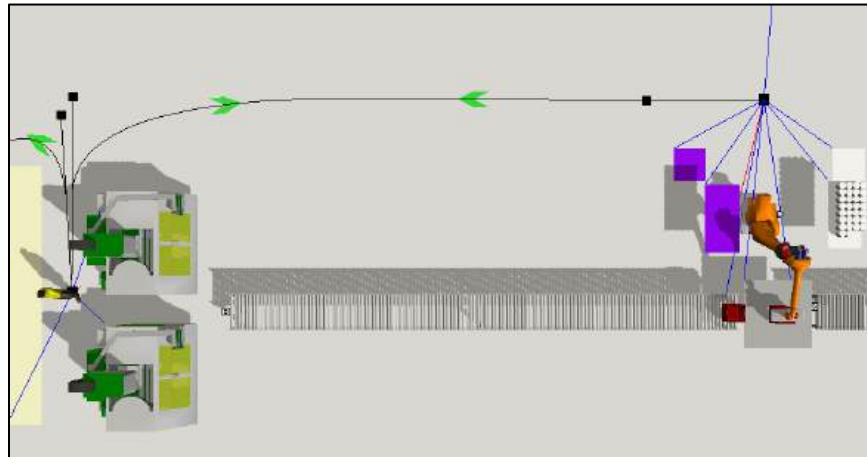
Create a curved network path between the Break Area and the Finishing Area.

- Right-click on one of the green arrows on the edge/path and select *Curved*, as shown in the figure to the right.
The Curved setting creates two "handles" on the path or edge, as shown in the figures below.
- Use the "handles" that are now available on the path to form a curved path that avoids other objects, as shown in the figure below – the one on the left shows the initial position of the handles and path, and the second shows the final position.



Repeat the steps for the network path between the Finishing Area and the Packing Area,

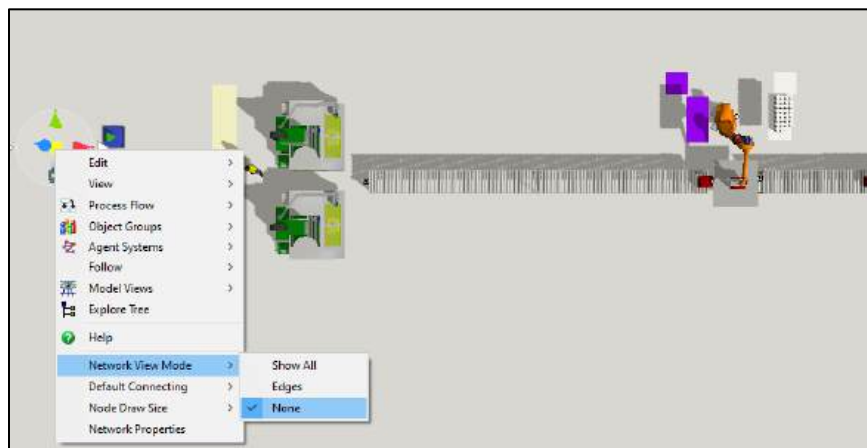
- As before, right-click on one of the green arrows on the edge/path and select *Curved*.
- Use the handles that are now available on the path to form a curved path that avoids other objects, as shown in the figure below.



Visualization of the network can be changed by

- Right-clicking on one of the **Network Nodes**, then selecting *Network View Mode* from the dropdown menu, and then selecting one of the following options:
 - *Show All*, the default, and what is seen when the networks are created.
 - *Edges*, which shows only the connecting lines and not the nodes.
 - *None*, which shows only one **Network Node**, the one selected, and no edges or paths.

The last case is shown in the figure below. However, the model that is available with the primer uses the *Show All* option so the network is apparent to anyone viewing the model.



A **Task Executer (TE)** will remain on a network path until it gets to a **Network Node** associated with the task it executes. At the node, it can either *travel offset* or remain at the node to execute the task. By default, it uses *travel offset*. This activity was discussed earlier in the introductory chapter on **Task Executors**. At that time, the **Operator** *FinishOperator_1* was set not to travel offsets. Therefore, in this case, the **Operator** performs all its tasks in the Packing Area at the node. However, if a **TE** is set to travel offsets, it travels to the object where the task is performed, no matter how far it is from the node. Recall, whether a **TE** travels offset or not is a property that is controlled by a checkbox on the **Operator's Task Executer** pane.



If you haven't already done so, save the model. Recall that it is good practice to save often.



Use the *Save Model As* option in the **File** menu to make a copy of the existing model to be further customized in the next section. Again, you can use any file name, but in the primer, the next model is referred to as *Primer_10A*.

17.2 Controlling task executor travel with A* Navigation

As described earlier, by default, **Task Executors**, such as **Operators** and **Transporters**, travel in straight lines between points in a model. While this is efficient, it can be unrealistic – in reality, operators cannot go through machines, conveyors, etc. The straight-line path not only doesn't look right, but it can affect estimated system performance – it takes longer to follow an object-avoidance route compared to the straight-line distance between objects.

In the previous section, the travel paths of the Finishing Operator, are controlled through a network of connected **Network Nodes**. Each node is considered an object and is placed on the modeling surface at key locations to establish the paths, which can have either straight or curved segments. In essence, this approach tells the **Task Executor** where to go. The alternative approach considered here, referred to as A* or AStar, tells the Task Executor where *not* to go; thus, the **TE** decides the path that avoids the specified obstacles.

A* (pronounced A-star) is a popular search algorithm commonly used in computer science and mathematics. It uses heuristics to find the shortest path between points, considering barriers or no-travel zones. As with all other topics in the primer, only the basics are introduced here, and the default settings are mostly used.

Utilizing the A* algorithm in *FlexSim* is quite easy. The objects associated with the A* algorithm are in the section of the **Object Library** named **A* Navigation**, just below the **Visual** section.

17.2.1 The basics of A* using a simple study model

Before implementing A* in the current primer model, use a simple study model to understand the basics.

- Create a new model using all default settings.

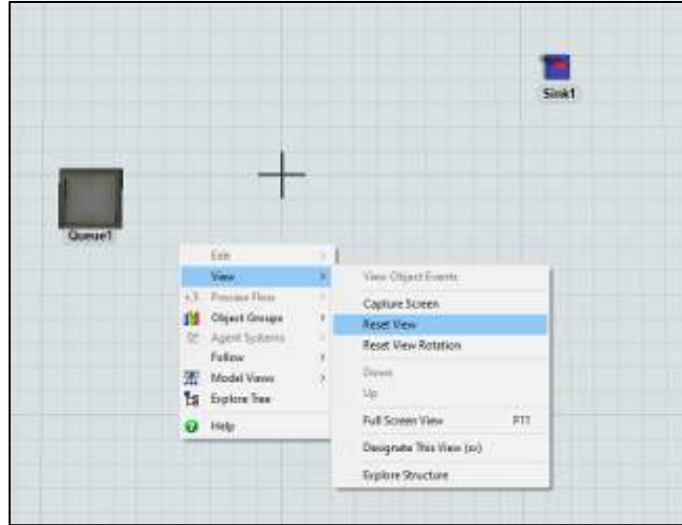
It is good modeling practice to try new concepts outside of the main model being constructed. These simple models, often referred to as *study* models, are useful in learning new concepts or understanding how *FlexSim* features behave. They focus on a particular problem or a portion of complex logic.

As shown in the figure below, create a simple study model of an operator transporting items between two locations, from the Source to the Sink. A **Queue** is included that is not connected to any other objects – it is used to represent a storage area that the operator should avoid. For all objects, use the default property values.



- Drag out a **Source**, **Sink**, **Operator**, and **Queue** from the **Library**.
- A-connect the **Source** to the **Sink**.
- Reset the view so that it is in planar view.

As a general note, anytime you want to reset the model view, right-click anywhere on the modeling surface, but not on an object. Then, select **View**, then *Reset View*, as shown in the figure to the right. The resulting view is of the x-y plane centered at the origin.



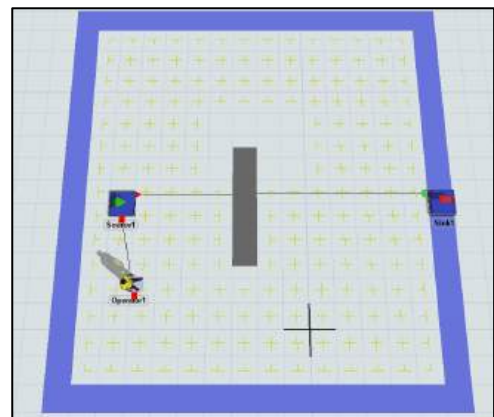
- Place the **Queue** on the straight-line path between the **Source** and the **Sink**.
- Center-connect the **Operator** to the **Source**.
- On the **Output** pane of the **Source**, check the property *Use Transport*.
- Save the model as *A-StarStudy*.
- **Reset** and **Run** the model. As expected, the **Operator**'s path should be through the **Queue**, as shown in an earlier figure.

Now use **A* Navigation** to control operator travel.

- Delete the **Queue** object; we'll replace it with a "barrier." A **Barrier**, like a **Divider**, is an **A* Navigation** tool that creates obstacles for **Task Executors** to travel around.
- Drag out a **Barrier** object from the **A* Navigation** pane.
- Resize the **Barrier** by making $\alpha = 1$, and locate it between the **Source** and **Sink**, as shown in the figure to the right.

When the model is **Reset**, a blue box encompasses the model. It is automatically created and is called the **Grid Bounds**.

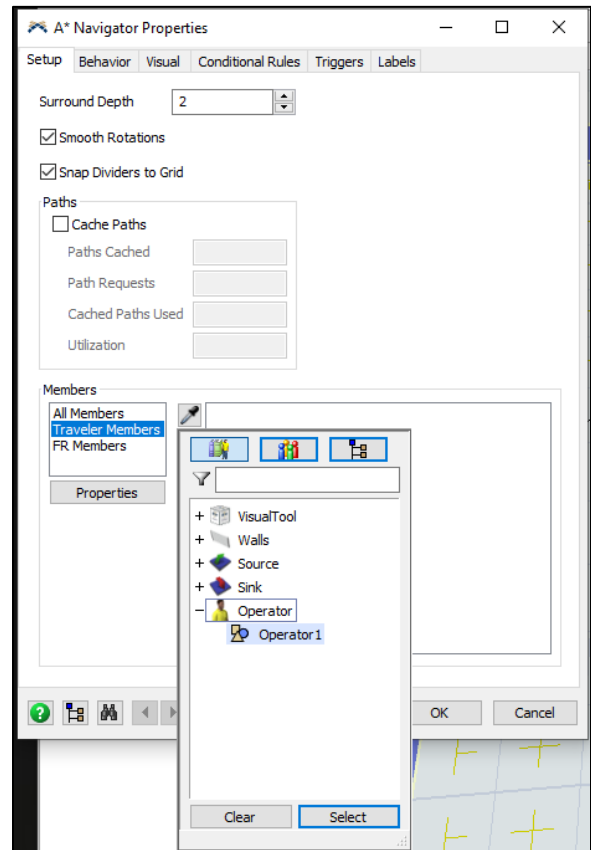
The box can be hidden by unchecking *Show Bounds* on the **Visual** tab of the **A*Navigator** tool.



When an **A* Navigator** object is added to a model, it is automatically added to the **A* Navigator** in the **Toolbox**.

- Double-click the **A*Navigator** tool in the **Toolbox** and note that it has several tabbed interfaces, such as *Setup*, *Behavior*, and *Visual*.
- In the **Members** section of the **Setup** tab, select *Traveler Members*, then use the  button to add objects. In this case, select *Operator1* in the **Operators** section, as shown in the figure to the right. Objects selected here will have the A* algorithm applied to them.
- Also in the **Members** section is *FR Members*, where FR is an abbreviation for Fixed Resource. The interface is just below the *Traveler Members*, again on the **Setup** tab. No *FR Members* need to be added in this simple example.

Since the **Barrier** is an **A* Navigator** object, it is automatically assumed to be a member of the objects the *Traveler Members* must avoid.



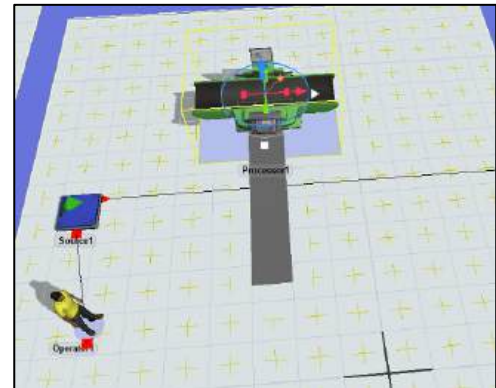
- **Reset** and **Run** the model. The **Operator** should identify and follow the shortest path between the **Source** and **Sink** that avoids the **Barrier**.

As noted above, a blue box provides an outline of the objects that are considered by A*. The box can be hidden by unchecking **Show Bounds** on the **Visual** tab of the **A*Navigator** tool.

The size and location of the blue box may need to be adjusted since the operator will not travel beyond its boundaries. It can be repositioned by selecting anywhere on the blue border and dragging the box to the desired location. It can be resized by selecting and dragging one of the red arrows that appear on the border when the object is selected.

TEs can avoid basic **Fixed Resource** objects as well. An example is illustrated here. A **Processor** object is placed next to the **Barrier**, so the **Operator** needs to avoid both objects.

- Drag out a **Processor** object and place it adjacent to the **Barrier**, as shown in the figure to the right.
- Double-click the **A*Navigator** tool in the **Toolbox** to open its interface. In the **Members** section of the **Setup** tab, select *FR Members*, and use the  button to select *Processor1* in the **Processor** section.
- **Reset** and **Run** the model and observe the behavior. The **Operator** should identify and follow a path between the **Source** and **Sink** that avoids the **Barrier** and **Processor**.

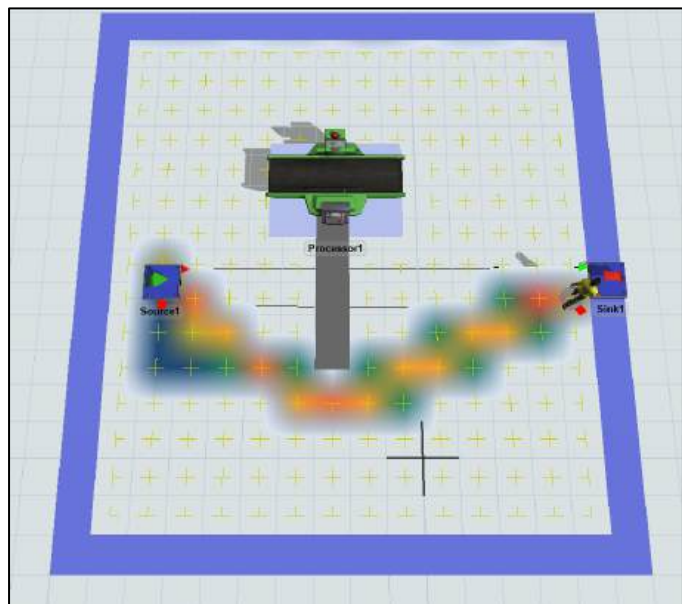


A heat map can be used to see the path the operator travels and the frequency at which the path is used.

- In the A* Navigation tool, check the *Show Heat Map* box on the **Visual** tab.
- Also, just below this property, check the **Transparent Base Color** box.
- **Reset** and **Run** the model and observe the behavior. The **Operator** should identify and follow a path between the **Source** and **Sink** that avoids the **Barrier** and **Processor**, as shown in the figure to the right.

The “warmer” zone on the heat map - yellow, as opposed to green - indicates a higher frequency of travel on that path.

Experiment with the placement of objects and note the effects on the operator’s travel path.



If you haven’t already done so, save the model. Recall that it is good practice to save often.

17.2.2 Implementing the A* algorithm in the primer model

Now that the basics of using the **A* Navigation** approach have been introduced through a simple study model, it is incorporated into the primer model.

The base model for the additions described in this section of the chapter is **Primer_10** that was saved at the end of Section 17.1. However, a copy of that file was saved as **Primer_10A**; thus, we begin with that file.

The A* approach for controlling task executer travel is considered to be better for representing the system modeled in the primer example. However, this is not always true; sometimes, using a Path Network is better. Both approaches are introduced in the primer so the modeler can decide which is best based on the system they are modeling.

As to why the A* approach is better in this situation, consider the focus of the primer example. In this case, a simple system of objects is created so they can be scaled up to design and evaluate alternatives for a new production system in one of DPL's facilities. With the A* approach, the path network does not have to be updated each time a different design alternative or layout is considered.

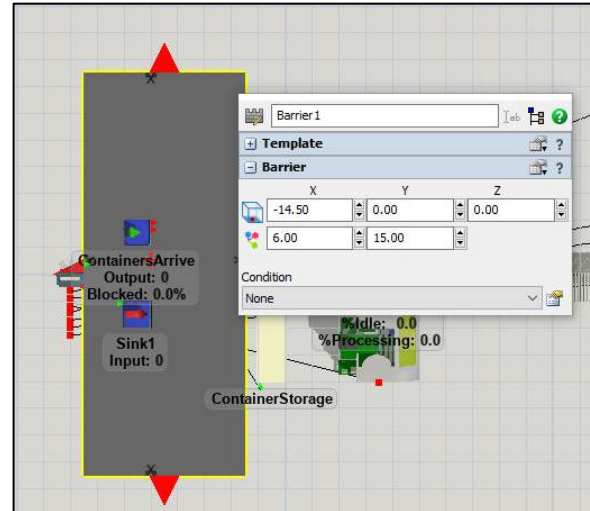
Since A* Navigation will replace the Path Network approach, all of the objects and connections used in the network approach will be removed. Keeping them in the model just adds unneeded complexity. Removal is fairly easy and does not require reversing all of the steps that were followed when it was created.

Remove the Path Network elements

- Assuming the path network was hidden except for one **Network Node** *nn_BreakArea* as described earlier, right-click on that node and select *Network Node View*, then *Show All*. This displays all aspects of the network.
- Select and delete each Network Node – *nn_BreakArea*, *nn_FinishArea*, and *nn_ParkingArea* – by selecting the node and then pressing the keyboard's delete key. This not only removes the node but also all connections to the objects that used the node.

Recall the **Operator** travels to the location of the **Dispatcher** for breaks. Set up an area the Finish Operator needs to avoid to get to the Break Area.

- Drag out a **Barrier** object from the Library; name, size, and locate it as shown in the figure to the right.



Configure the A* Navigation properties.

- Open the **A* Navigator** tool (at the bottom of the **Toolbox**) by double-clicking it.
- On the **Setup** tab and in its **Members** section, use the  button to add objects to the **Traveler Members** list. In this case, select the *FinishOperator_1* in the **Operators** section. Objects selected here have the A* algorithm applied to them.

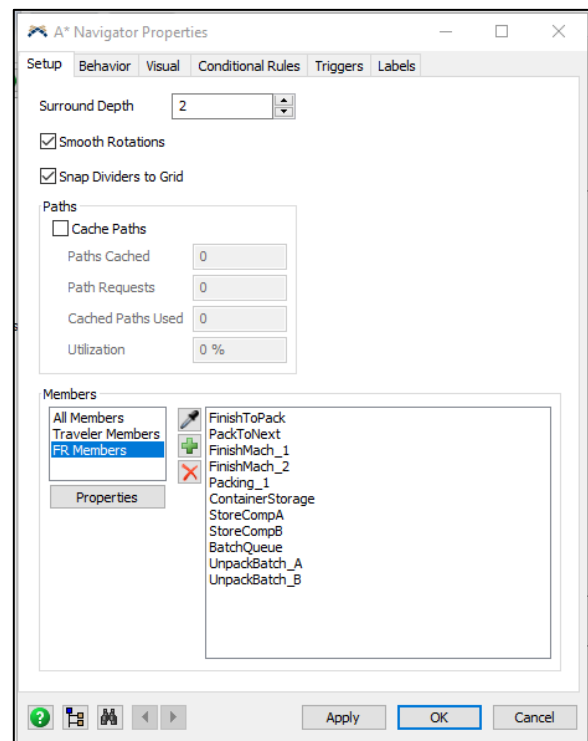
In a similar manner, add the **FR Members**.

Consider what objects the Operator needs to avoid – the conveyors (straight), Finishing Machines (Processors), packing table, stores (Queues), and component unbatching (Separators).

As shown in the figure to the right, the interface is just below the **Traveler Members**, again on the **Setup** tab.

- Select the objects shown in the figure to the right.
By selecting an object category, e.g., **Queues**, all objects in that category are selected; in this case, the objects *ContainerStorage*, *StoreCompA*, *StoreCompB*, and *BatchQueue* are added to the **FR Members** list.
Multiple object categories can be selected at a time by holding down the Shift key when selecting categories.

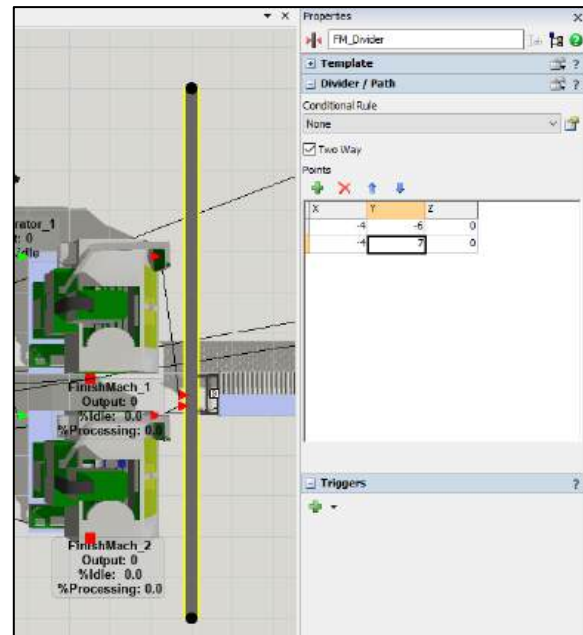
The other object types in the model are not selected since they are not considered to be barriers to the Operator's travel.



Besides the basic *FlexSim* modeling objects that are used in the model, the only other thing the **Operator** should avoid is the facility's walls. For the **Operator** to avoid the walls, the **Divider** object in the **A* Navigation** section of the **Object Library** or the **Walls** object in the **Visuals** section can be used. Since the **Divider** is an **A* Navigation** object, it is automatically considered a member; if any **Walls** objects are used, they need to be added to the **FR Members** list in the **A* Navigation** tool.

So that the **Operator** does not travel between the two Finishing Machines, add a **Divider** to restrict travel.

- Select the **Divider** object from the **Object Library** and click on one end of the vertical wall to the right of the finishing machines. (This wall divides the finishing area from the packing area). Then click on the other end of the wall on the layout. Press the Esc key to stop drawing the divider. This results in a line segment with a circle at each end, as shown in the figure to the right.
- Name the object *FM_Divider*.
- Adjust the shape as necessary by dragging a circle end point or the line segment. The wall can be located more precisely by double-clicking the Divider object and modifying the table values, as shown in the figure to the right. Each row in the table is the x-y-z coordinate of one of the circles on the **Divider**.



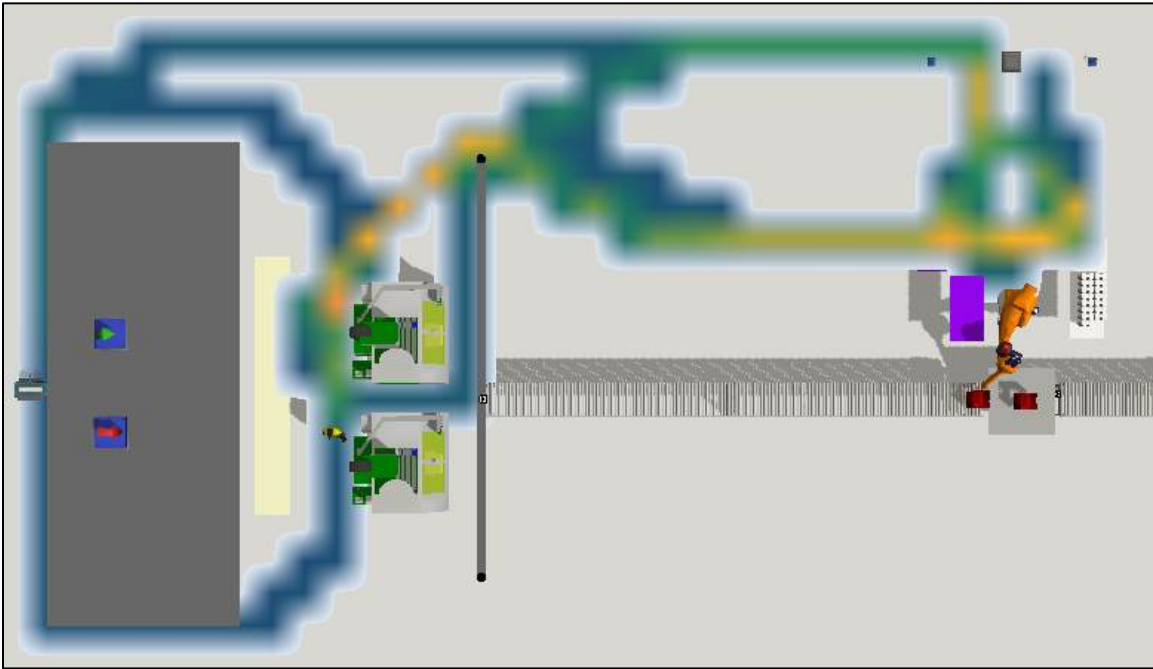
Note that **Barriers** and **Dividers** have no height; the z-value is only for the location of the barrier/divider in the z-direction. To represent a wall in a model, use the **Walls** object in the **Visual** section of the **Object Library**.

Check the following conditions.

- The **Robot** object *Robot_1* should not be in the **Members** list for the **A* Navigator** tool.
- For the **Robot**,
 - The **Navigator** property should be set to *None* in its **Travel pane**. If not, use the  button to delete the specified navigator. While the **Robot** is a **Task Executer**, it is not a mobile resource.
 - The picklist below the **Navigator** property should be set to *Travel offsets for load/unload* tasks. If not, use the dropdown menu to set the desired value.
- On the **Visual** tab of the **A* Navigator** tool,
 - Check the *Show Heat Map* box.
 - Check the *Transparent Base Color Heat Map* box.

For clarity, any or all of the boxes on the **Visual** tab of the **A* Navigator** tool can be unchecked. This does not affect the underlying logic, but makes the model a bit cleaner visually. However, you may want to leave *Show Barriers* and *Show Heat Map* checked.

- **Reset** and **Run** the model. Verify that the **Operator** avoids all of the fixed resources, barriers, and dividers, as shown in the figure below.



If you haven't already done so, save the model. Recall that it is good practice to save often.



Use the **Save Model As** option in the **File** menu to make a copy of the existing model to be further customized in the next section. Again, you can use any file name, but in the primer, the next model is referred to as Primer_11.

18 EXPERIMENTATION IN FLEXSIM

Chapter 18 introduces *FlexSim's* Experimenter, which provides a convenient means to consider simulating multiple scenarios and replicating each scenario. The Experimenter is used to study two aspects of the system modeled so far – (1) the effect of the size of the buffer of containers prior to Finishing on performance and (2) the effect of component replenishment plans on performance.

Simulation models are created for analyses. Simulations approximate the behaviors of a system operating under a set of conditions. The behaviors are expressed in terms of measures of system performance, which are considered the output from a simulation. The analyses typically use the measures to find the “best” set of conditions under which to operate or understand the sensitivity of changes in operating conditions on performance.

In the models so far, output has been limited to a single run and primarily to basic object statistics displayed on the object and in **Dashboards**. While this is good for testing a model's behavior, it is insufficient for analysis.

Since models involve randomness, results vary from run to run, just as in the real world, performance varies over time. Therefore, decisions should not be made using a single simulation run based on a set of conditions operating over a set time period. Instead, each simulation scenario is replicated a specified number of times. The number of replications must be user-specified, but since deciding the number involves probability and statistics, discussion of how to choose the number of replications is outside the scope of the primer. In general, the more replications, the better, but it depends on the amount of variability in the system. Also, the number of replications determines how long it takes to run an experiment, and there are diminishing returns. Therefore, in many cases, between ten and thirty replications is a reasonable number.

FlexSim, through its **Experimenter** tool, provides a convenient means for developing information to support analyses by automating the running of a model with different inputs and collecting the results, i.e., running multiple simulations under a set of conditions and multiple simulations under different sets of conditions.

In *FlexSim*, using the **Experimenter** means running a **Job**, where a **Job** is defined by specifying the following:

- Performance Measures, often called the output from a simulation, are the key indicators of how a system performs, and their values are used to decide which alternative is best.
- Decision Variables and their values, often called the inputs to a simulation, are the properties of a system that can be changed to improve performance. In statistical terms, decision variables are analogous to factors and their values are levels of the factor.
- Scenarios are the alternatives being considered; each Scenario is specified as a combination of decision variables and their values. Scenarios can be defined by several means; these means are a part of the Analysis Strategy, which is defined below.

- Experiment Settings are the values for run length (**Stop Time**), number of replications per scenario, and length of the warm-up period (**Warmup Time**) if applicable.
- Analysis Strategy – FlexSim provides the following types of Jobs representing the Analysis Strategies.
 - *Experiment* – a specified set of scenarios and performance measures. Simulation uses the values of the input variables to provide measures of system performance for each scenario.

The scenarios may be defined as a specific set of operational alternatives or using a Design of Experiments (DoE) approach. DoE is a branch of applied statistics that provides efficient means to study the effect of multiple input variables on a system’s response. It provides a structured and systematic means for analysis and avoids using approaches such as trial and error or changing one variable at a time, which overlooks the combined effect of multiple variables. Discussion of DoE is beyond the scope of this primer.
 - *Optimization* provides a methodology for effectively searching the design space for the “optimal” or “best” scenario, i.e., the best combination of decision-variable values. The search is carried out based on one or more objectives, which are based on the system’s performance measures.

Unlike the Experiment job, the user does not specify the scenarios, just the variables to be considered, their data type, and their operational range of values to be considered. For example, if one of the decision variables is the number of operators to use, then it would be specified that this factor is integer-valued, and the system could use one to five operators.

Scenarios are defined by the optimizer – as part of its algorithm scenarios are defined on the past search history and the objectives, which are based on performance measures.

FlexSim does not include an optimization tool. However, it interfaces seamlessly with a prominent commercial optimizer, *OptQuest*, and provides links to *C++* and *Python* so the two software can communicate and perform an optimization.
 - *Range-Based*. Similar to the Optimization strategy in that specific scenarios are not specified – only the decision variables to be considered and their minimum value, maximum value, and step size are specified. For example, if processing time can be set to a value between 10 and 20 with a step size of 2.5, then this option would automatically consider the following five values for processing time: 10.0, 12.5, 15.0, 17.5, and 20.0. Thus, this decision variable or factor would be considered at five levels.

Be aware that the number of scenarios can grow quickly. If there were three decision variables being considered, each at five levels, then there would be 125 scenarios ($5 \times 5 \times 5 = 125$). If each scenario is replicated ten times, then there would be 1,250 simulations performed.

The **Experimenter** collects results for the simulations and stores them in an external file and not in the model file; the file is referred to as the Results Database File. This has several advantages, such as it does not rerun the experiments if they have already been run and are in the database file, additional replications of any scenario are appended to the file, and it reduces memory requirements. This is mentioned for awareness and possible future use; the primer does not work with the database file.

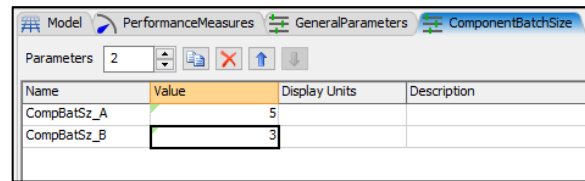
As has been mentioned before, analyzing simulation models requires a foundation in probability and statistics and includes a variety of methodologies. It is beyond the scope of this primer to discuss simulation analysis - it only introduces the means in *FlexSim* for designing and carrying out experiments and obtaining output from a

simulation model. For more information on simulation analysis, see for example, Chapters 10 and 11 in the *Applied Simulation Modeling and Analysis Using FlexSim* textbook.

Two examples of experimenting with the current simulation model are discussed in the subsequent sections.

However, before proceeding with the **Experimenter**, you may have noticed when running the model that Component A's buffer seems to be empty a lot, and Component B's buffer seems to stay rather full. The modeler checked with DPL's production engineers and discovered that they provided the wrong values for their initial batch size estimates. The values were reversed. Therefore, update the values in the **Parameter Table**.

➤ In the **Parameters Table** *ComponentBatchSize*, set the **Values** for *Component A* and *Component B* to 5 and 3, respectively, as shown in the figure to the right.



Name	Value	Display Units	Description
CompBatSz_A	5		
CompBatSz_B	3		

The base model for the additions described in this chapter is **Primer_10A** that was saved at the end of Chapter 17. However, a copy of that file was saved as **Primer_11**; thus, we begin with that file.

18.1 Effect of buffer size on performance

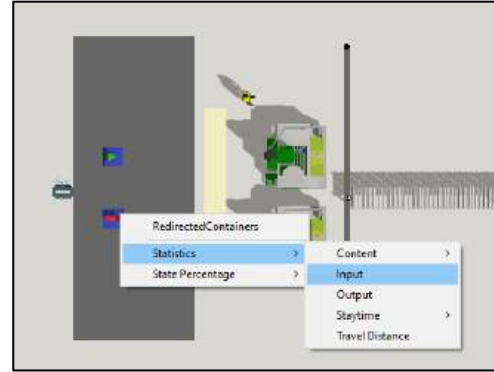
The first **Experimenter** example is used to assess the effect on system performance of the size of the containers' buffer (the **Queue** object named *ContainersStorage*) that is located prior to the Finishing Area. The effect is measured by the number of containers that are redirected to another location because there is no space in the buffer for an arriving container.

Create a measure to assess the effect of various alternatives on system performance. The **Performance Measure Table** tool in the **Toolbox** provides a means for defining these measures. Each model contains a blank table named *Performance Measures*. Like the **Model Parameters Table**, multiple **Performance Measures Tables** can be created and used. However, we'll just use the default.

- Before setting up the table, rename the **Sink** *Sink1* to *RedirectedContainers*. Recall that *Sink1* receives any containers for which there is no space in the **Queue** *ContainerStorage* when the containers arrive.
- Double-click the **Performance Measures** table in the **Toolbox** to open it. The default table name is okay.
- Change the name of the first measure from *PerformanceMeasure1* to *RedirectedContainers*.

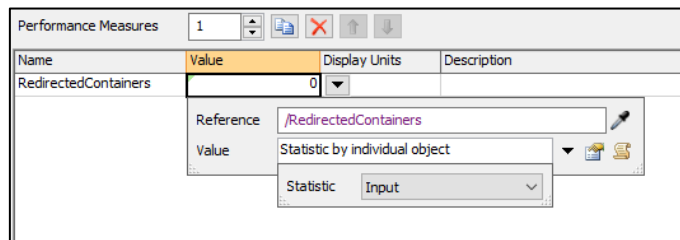
- Use the dropdown menu to set its value.
For **Reference**, use the sampler tool (eyedropper) to select the **Sink** *Redirected Containers* in the 3D model.

When the object is selected, select *Statistics*, then *Input*, as shown in the figure to the right. This will measure the number of items entering this **Sink**.



The selection automatically populates the **Value** dropdown menu for performance measure *RedirectedContainers*, as shown in the figure to the right.

Again, this measure provides a total count of the number of containers that are redirected when they arrive.

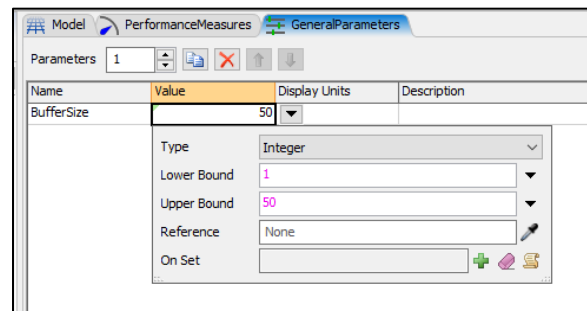


To change the **Queue**'s buffer size during an experiment, set this variable up as a **Model Parameter**.

- Since the existing **Model Parameter Tables** have been set up for specific types of model parameters, create a new table for general parameters. Through the **Toolbox**, add a **Model Parameters Table** named *GeneralParameters*. To create the table, in the **Toolbox**, select *Model Parameter Table* under the **Statistics** section.

- Rename *Parameter1* as *BufferSize*.

- Set the **Value** as shown in the figure to the right, where **Type** is *Integer*, **Lower Bound** is *1*, **Upper Bound** is *50*. Set the initial value to 50.



Change the **Queue**'s maximum content to reference the **Model Parameters Table** and not use the current fixed value of 50. However, the input for the **Max Content** property value on the **Queue** pane can only be numeric. Therefore, the reference to the **Parameter Table** will be done with an **OnReset** trigger and a small amount of *FlexScript* code.

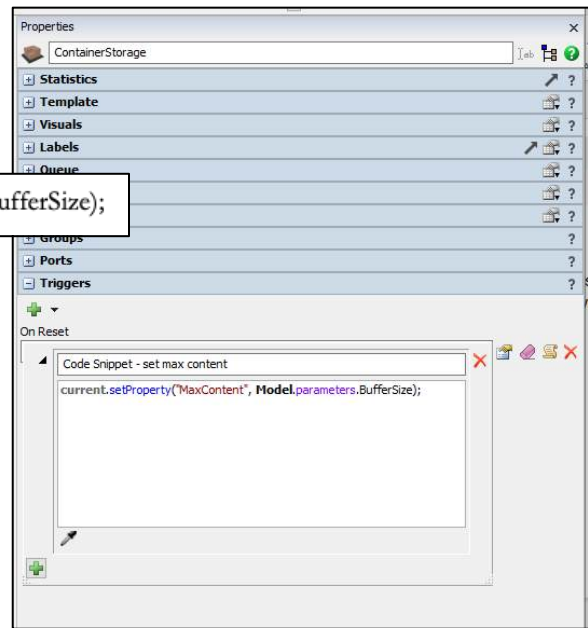
On the **Queue ContainerStorage**,

- Add an **On Reset** trigger.
- Using the  button to the right of the **On Reset** text box, select the last item, *Code Snippet*
- As shown in the figure to the right, type in the following.

```
current.setProperty("MaxContent", Model.parameters.BufferSize);
```

Note as you type, the interface provides the syntax and keywords.

This command sets the property named **MaxContent** on the current object (in this case, **Queue Containers.Arrive**) to the value stored in the **Model Parameter** named **BufferSize**.



Test the changes so far.

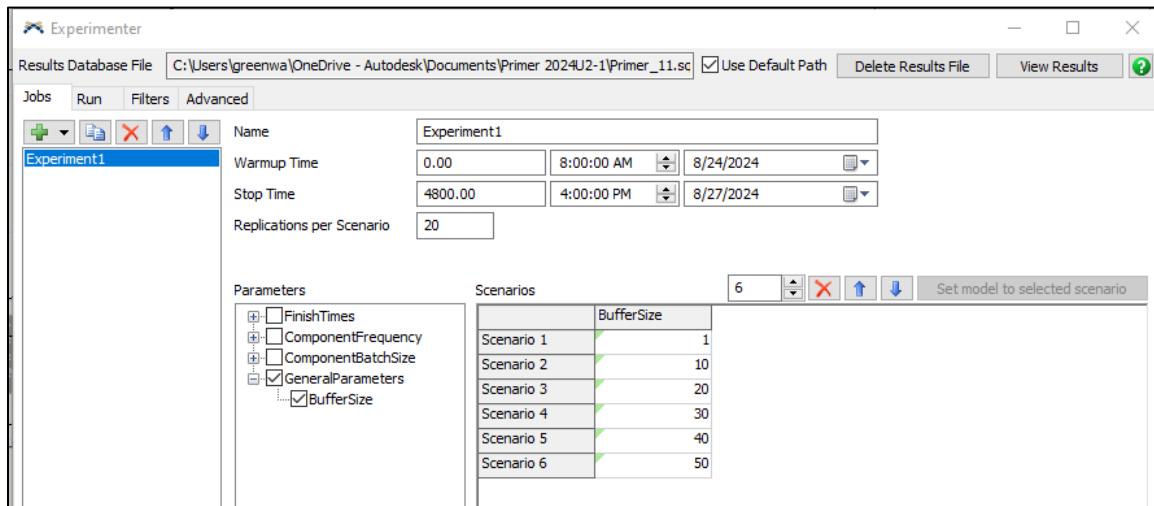
- In the **Model Parameters Table General Parameters**, set the **BufferSize** parameter to a smaller value, say 1.
- **Reset** and **Run** the model.
- Check the value of **RedirectedContainers** in the **Performance Measures Table**. In the case shown to the right, only eight containers were redirected when the buffer size was limited to one container.

Model PerformanceMeasures GeneralParameters			
Performance Measures 1			
Name	Value	Display Units	Description
RedirectedContainers	8		



If you haven't already done so, save the model. Recall that it is good practice to save often.

- Access the **Experimenter** by selecting it from the **Statistics** dropdown on the **Main Menu** or the **Statistics** section in the **Toolbox**. This opens the **Simulation Experiment Control** window, which, as shown below, contains multiple tabs and properties. For now, only the minimal features are considered.



The first tab, **Jobs**, is for specifying the Analysis Strategy, Experiment Settings, Decision Variables Scenarios, and Performance Measures, which were defined earlier.

The type of Analysis Strategy is an *Experiment*. Recall that the other types are *Optimization* and *Range-Based*. By default, the **Experimenter** starts with an Experiment named *Experiment1*. The strategy can be changed, or new strategies can be added by using the  button on the left side of the interface. In this example, the experiment considers a range of buffer sizes, which are the maximum content of the **Queue ContainerStorage**.

For Experiment Settings

- Change the **Name** (call this experiment *BufferSize*).
- Leave the default value of 0.00 for **Warmup Time**.
- Change **Stop Time** to 4800 minutes.
- Set **Replications per Scenario** to 20.

Decision Variables are selected from **Parameters Tables**.

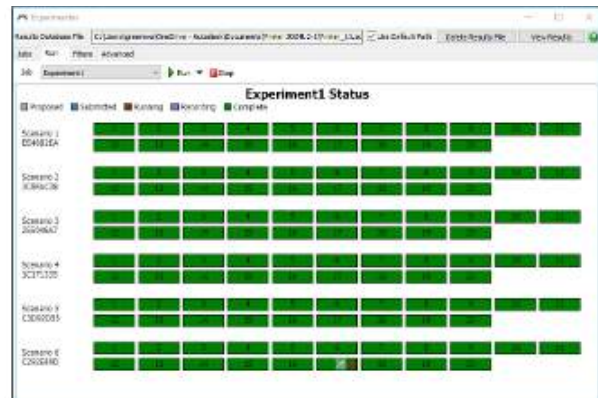
- In the **Parameters** section of the interface, select **Buffer Size**, the only factor in **Model Parameters Table GeneralParameters**.

Scenarios are the alternatives being considered.

- In the **Scenarios** section of the interface, use the spinner to create six scenarios.
- As shown in the figure above, enter the six alternatives (1, 10, 20, 30, 40, and 50) in the *BufferSize* column.

All Performance Measures defined in the model are evaluated in each experiment. Therefore, action is only necessary if a desired performance measure needs to be defined. In this case, the measure *RediurectedContainers* has already been defined and will be evaluated for all replications of all scenarios.

- On the **Run** tab, as shown in the figures below, click the **Run** button (green triangle) just to the right of the **Job** dropdown menu (*Experiment1* or *BufferSize* should be selected).

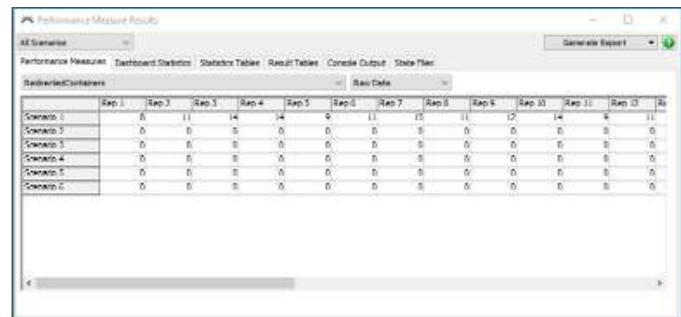
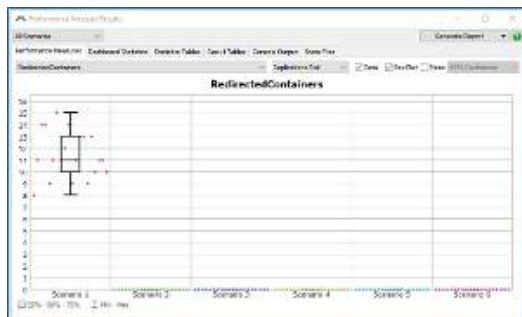


When the run begins, as shown in the figure to the left above, a gray display box is created for each replication of each scenario. The boxes change color as that simulation is run. The experiment is complete when all of the boxes are green, as shown in the figure to the right above

- To view the result, press the **View Results** button in the upper right portion of the interface.

As shown in the figure to the left below, the default view is a *Replications Plot*. It is a Box-and-Whiskers Plot, where the ends of the box denote the 25th and 75th percentiles, the horizontal bar in the box indicates the median (50th percentile), and the end bars are the extremes, i.e., the minimum and maximum number of redirected containers.

Other views of the results of the experiment are available through the dropdown menu to the right of the name of the performance measure. The figure on the right below, referred to as *Raw Data*, shows the value for each replication and scenario.



Note that in this case, items are diverted to the **Sink RedirectedContainers** only in the first scenario, where the buffer size is 1. In this scenario, items were diverted due to a full storage area in all 20 replications. No containers were diverted in any of the other scenarios. This indicates not much storage space is needed for containers entering the system, at least under the current conditions. This could change as other parts of the model are developed and as system parameters change.

- For now, to be conservative, set **BufferSize** parameter to 5 in the **Parameters Table GeneralParameters**.



If you haven't already done so, save the model. Recall that it is good practice to save often.

18.2 Effect of component replenishment plan on performance

The second Experimenter example is used to assess the effect on system performance of the component replenishment properties.

Continue with the same model file as in the previous section; i.e., continue with the file named **Primer_11**.

The replenishment of components that are packed into containers is driven by two decision variables: the frequency of replenishment deliveries and the quantity replenished in each delivery (batch size). In this case, only batch size is considered. Three batch sizes are considered for each component. Therefore, there are nine combinations of conditions or scenarios ($3^2 = 9$). The experimental design is shown in the table to the right. The design considers all combinations of Component A at three levels (batch size = 3, 4, 5) and Component B at three levels (batch size = 2, 3, 4). In statistics, this is referred to as a full-factorial experimental design.

Scenario	Batch Size	
	Comp A	CompB
1	3	2
2	4	2
3	5	2
4	3	3
5	4	3
6	5	3
7	3	4
8	4	4
9	5	4

Component batch sizes are already stored in a **Model Parameters Table** so that they are readily available.

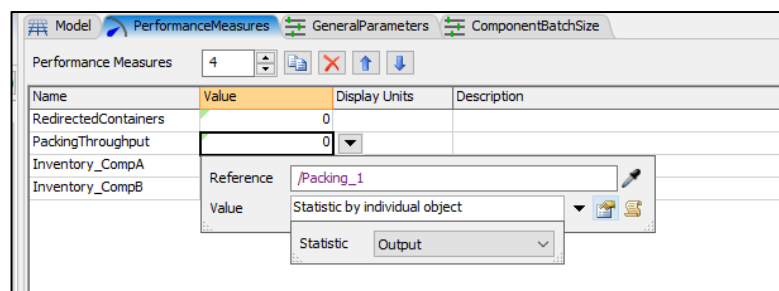
However, no performance measures have been specified to evaluate the batch size alternatives. In this example, we'll consider the throughput of the packing area and the maximum inventory in each component's storage area.

Therefore, add these performance measures in the **Performance Measures Table** in a manner similar to the process described in the previous section and as shown in the figure to the right.

➤ Add three measures and name them as in the figure to the right.

➤ Define the measure *PackingThroughput* as shown in the figure to the right, i.e., select the **Value** dropdown and use the sampler tool to select the **Combiner** object *Packing_1* in the 3D model. Then select *Statistic by individual object*, and for the **Statistic**, select *Output*.

This measures the output of the packing station or its throughput.



- Similarly, define the measure *Inventory_CompA* - select the **Value** dropdown and use the sampler tool to select the **Queue** object *StoreCompA* in the 3D model. Then select *Statistic by individual object*, and for the **Statistic**, select *Maximum Content*.

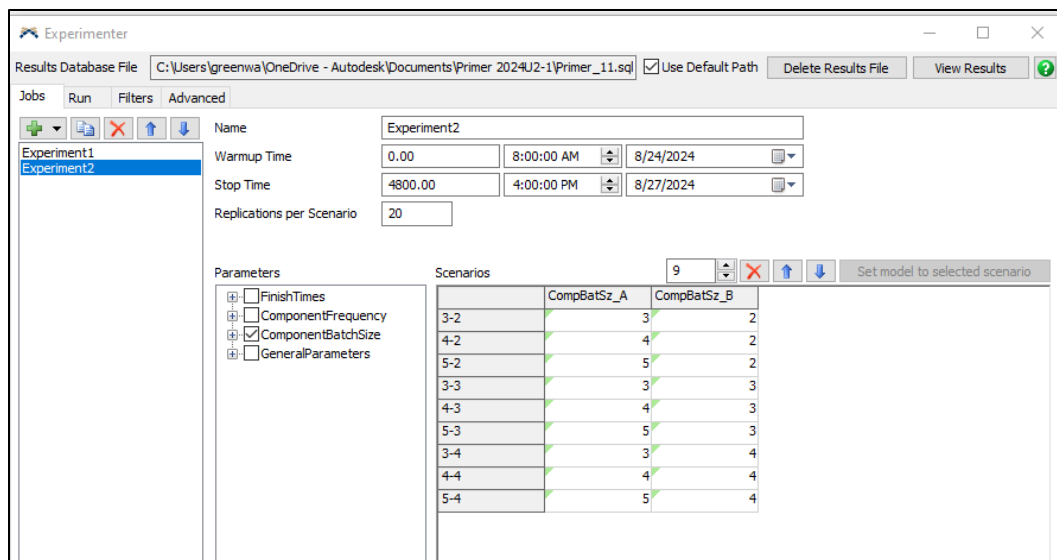
This measures the maximum number of component As that were stored in that area during the simulation.

- Similarly, define the measure *Inventory_CompB* - select the **Value** dropdown and use the sampler tool to select the **Queue** object *StoreCompB* in the 3D model. Then select *Statistic by individual object*, and for the **Statistic**, select *Maximum Content*.

This measures the maximum number of component Bs that were stored in that area during the simulation.

Now that the parameters and performance measures that support this analysis have been defined, open the **Experimenter**, either through **Statistics** on the **Main Menu** or the **Toolbox**.

Define the experiment as shown in the following figure and discussion below.



- Using the  button on the **Jobs** tab, add an *Experiment* type of **Job**.

Experiment Settings

- Change the **Name** from *Experiment2* to *ComponentBatchSize*.
- As in the previous experiment, leave the default value of 0.00 for **Warmup Time**, change **Stop Time** to 4800 minutes, and set **Replications per Scenario** to 20.

Decision Variables are selected from **Parameters Tables**.

- In the **Parameters** section of the interface, select both *CompBatSz_A* and *CompBatSz_B* from the **ComponentBatchSize** table.

Note that as each of the above parameters is selected, a column is added to the **Scenarios** section of the interface.

Scenarios are the alternatives being considered.

- Use the spinner to create nine scenarios in the **Scenarios** section of the interface.
- Rename the scenarios – change their names from *Scenario1*, *Scenario2*, etc., to those shown in the figure above.

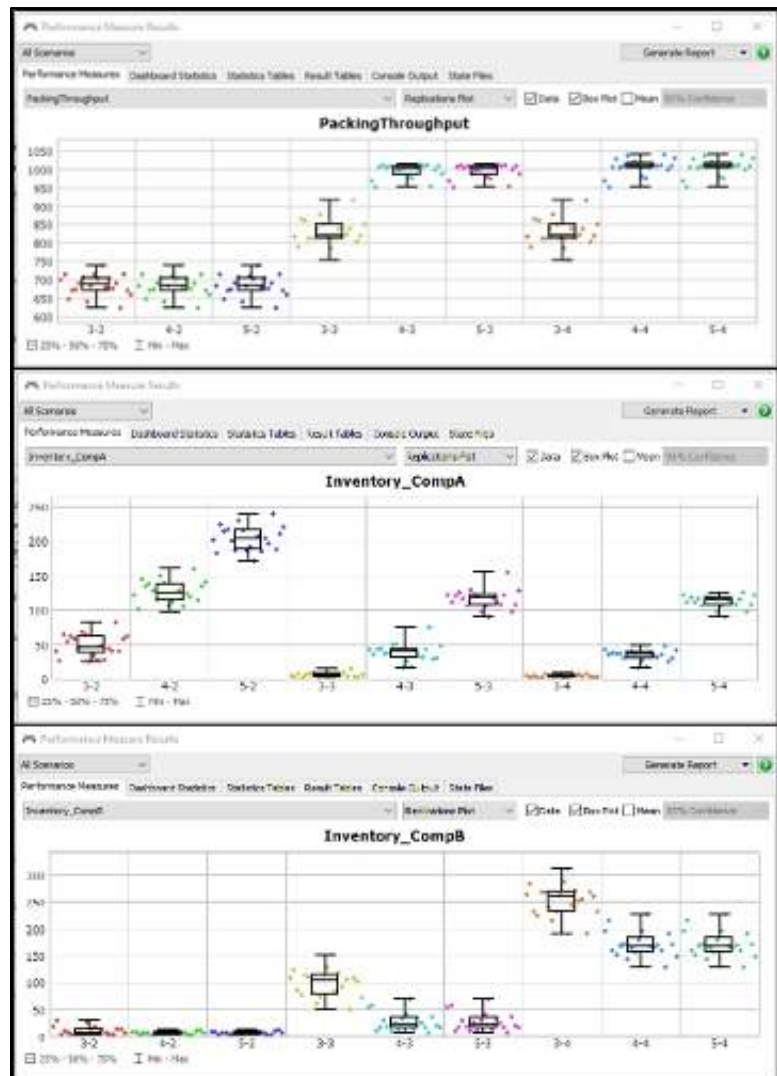
When the results of the experiment are displayed it will be easier to identify in operational terms which scenario is best. For example, if *Scenario 1* is best, it will be identified as 3-2, which is the batch size for Component A and B, respectively.

- As shown in the figure above, enter the nine alternatives in the two columns.

Based on the analysis from the previous experiment, be sure the incoming containers' buffer size is set to 5 (value for the parameter **BufferSize** in the **GeneralParameters** table).

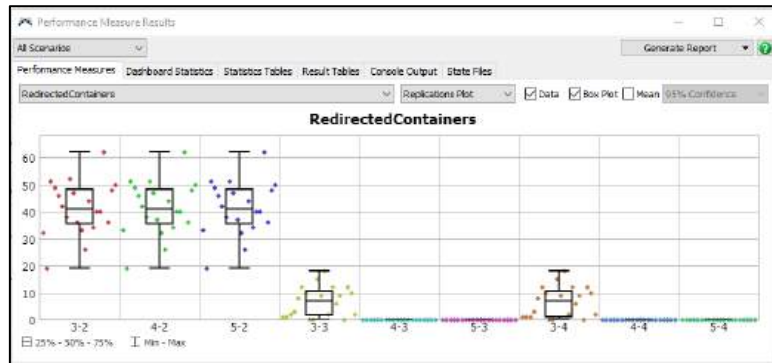
- To run the experiment, click the **Run** button (green triangle) on the **Run** tab of the Experimenter.
- When all of the cells are green, view the results by pressing the **View Results** button in the upper right portion of the interface.

Based on the Box-and-Whiskers plots in the figure to the right, it appears **Scenario 4-3** may be best. This would set the batch sizes of Components A and B to 4 and 3, respectively. This combination has a high throughput for the Packing Area and a low, but not the lowest, maximum level of inventory in the component storage areas. Of course, this is only based on the current conditions; the analysis would need to be performed again once the model is scaled up to the forecasted production rates. However, everything is now set up to run the experiments.



- Based on the current analysis, update the **Values** in the **Parameters Table** *ComponentBatchSize* to 4 and 3 for Components A and B, respectively.

Revisiting the other performance measure, *RedirectedContainers*, as shown in the figure to the right, indicates that **Scenario 4-3** results in minimal redirected containers when they arrive at the Finishing Area.



If you haven't already done so, save the model. Recall that it is good practice to save often.



Use the **Save Model As** option in the **File** menu to make a copy of the existing model to be further customized in the next section. Again, you can use any file name, but in the primer, the next model is referred to as *Primer_12*.

19 MATERIAL MOVEMENT VIA CONVEYORS

Chapter 19 introduces the Conveyor objects and adds conveyors in the primer model to transport containers between the Finishing and Packing Areas and after packing to the warehouse.

A common way to move items through a system is via conveyors. Conveyor systems are often complex, but *FlexSim* has extensive capabilities to model such systems. Of course, for this introductory primer, only the basics are considered. Objects associated with conveyors are in a section in the **Object Library** just below *Travel Networks*.

Modeling conveyor systems in *FlexSim* is discussed in the following two sections. The first section introduces the main conveyor objects, and the second section describes how to model conveyors in the primer example.

19.1 General description of conveyor objects

The primer example will use only a subset of the available conveyor-related objects:



Straight Conveyor



Decision Point



Curved Conveyor

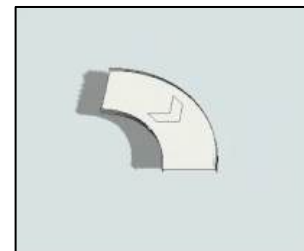


Photo Eye

Therefore, only these are discussed in this section.

Conveyors are much like **Fixed Resources**, e.g., **Source**, **Queue**, and **Processor**, with several key differences.

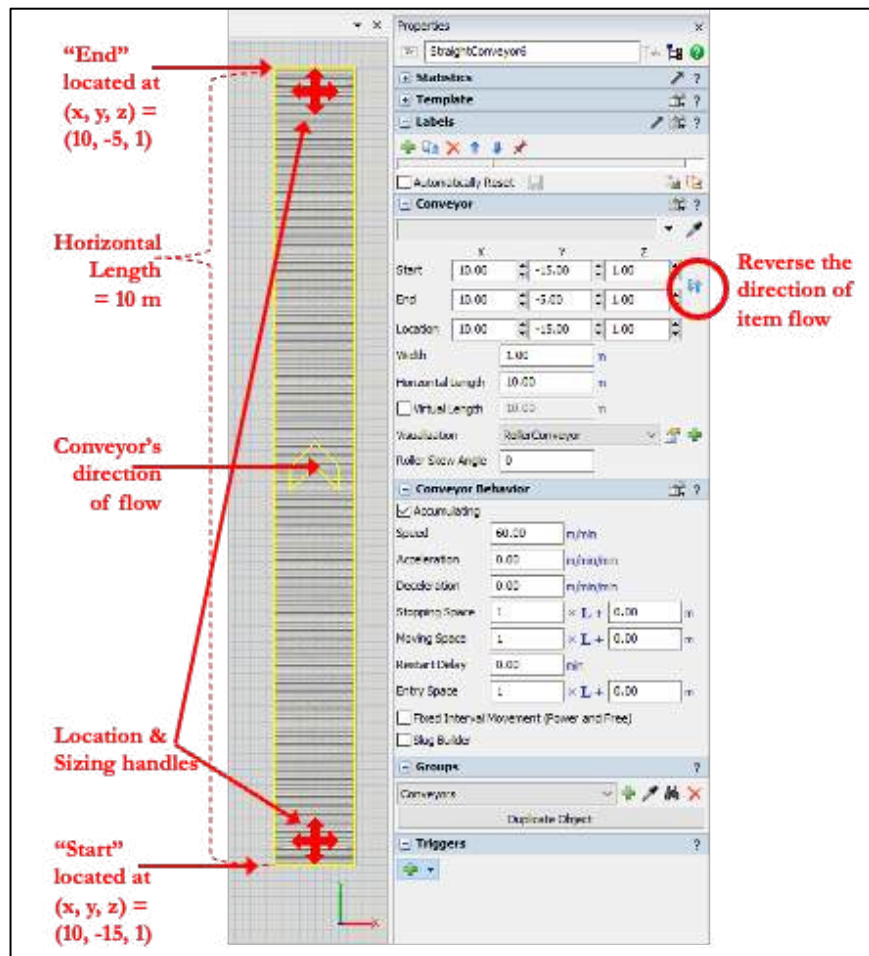
- **Conveyors** have two ends, referred to as *Start* and *End*, that can be modeled independently.
- **Conveyors** cannot be rotated in the x, y, or z directions as **Fixed Resources** can. However, the height (z location) can be changed at either end, resulting in a ramp or elevated conveyor.
- As shown in the GIF to the right, **Conveyors** contain manipulation handles to easily change their size and location. The crossed red arrows change the x and y location of a **Conveyor's** end. The double-headed green arrow changes the **Conveyor's Radius**.

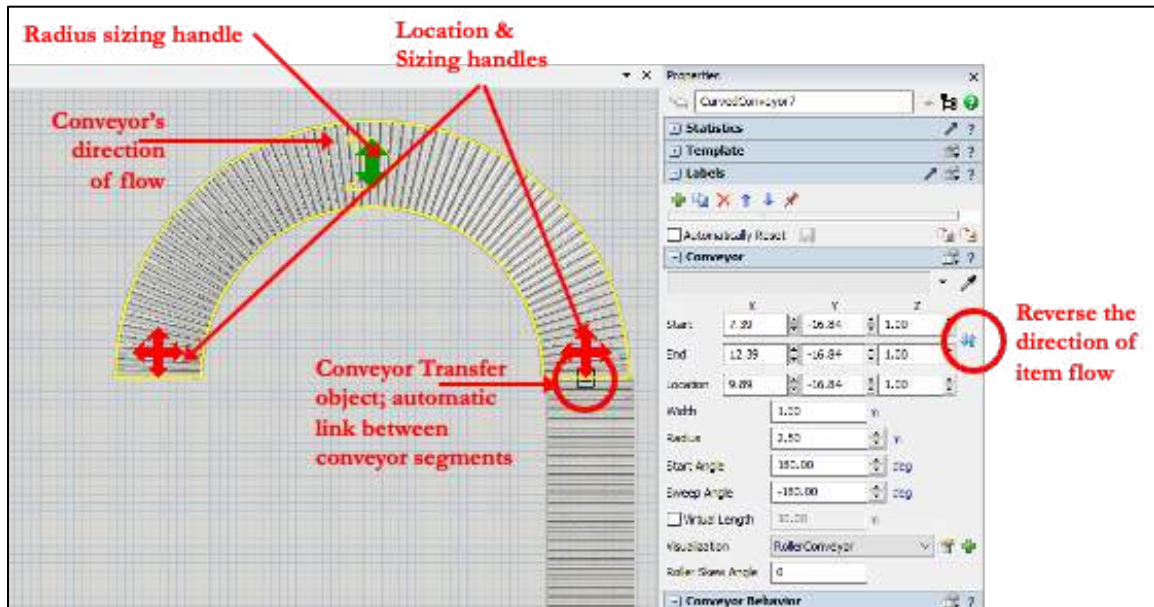


- As discussed earlier, when **Straight Conveyors** were briefly introduced to move flow items between the Finishing and Packing Areas, **Fixed Resources** and **Conveyors** were connected via transfer objects, either by an **Entry Transfer** or **Exit Transfer**. These transfers are automatically created when an A-Connection is made between the two object types; therefore, transfer objects are not a part of the **Library**.

- **Conveyor** segments are automatically joined. When two **Conveyor** segments are brought together, they become joined; i.e., segments are not connected using A-connections. The connecting object is called a **Conveyor Transfer**.
- Each **Conveyor** segment has a direction of travel, indicated by an arrow on the conveyor surface that is visible when the object is selected.
- A **Conveyor** may be *accumulating* (the default) or *non-accumulating*. An *accumulating* conveyor operates like a “roller conveyor,” where an item travels along the conveyor until the end and stops if it cannot be removed (downstream object or transport is unavailable). Subsequent items continue to flow on the conveyor and stop behind the one at the front. A *non-accumulating* conveyor operates like a “belt conveyor,” where, as in a roller conveyor, an item travels down the conveyor until the end and then stops if it cannot be removed (downstream object or transport is not available). However, on a belt conveyor, when one item stops, the belt stops, and then all other items on the conveyor stop in their current location.

The figures to the right and below show the basic properties of a **Straight Conveyor** and a **Curved Conveyor**, respectively.





Refer to the two figures above.

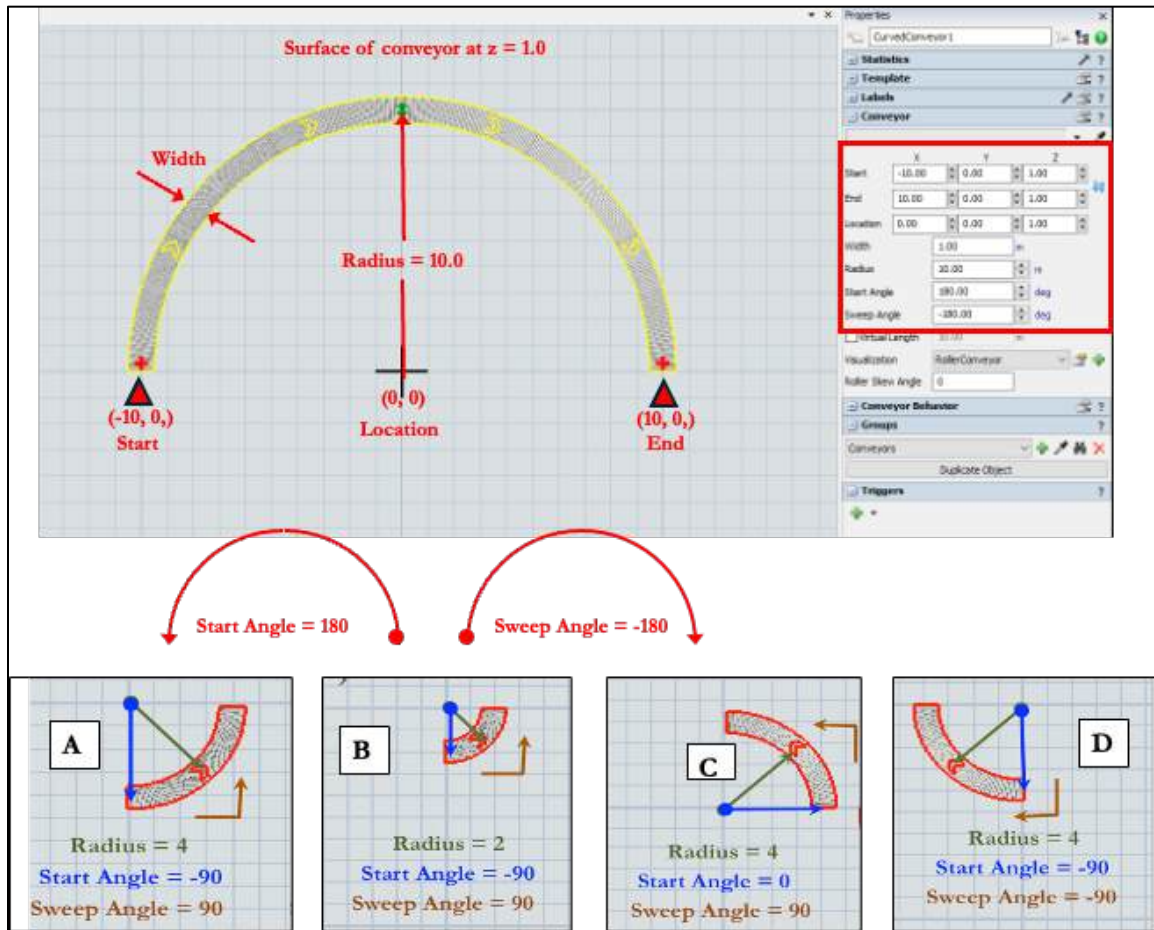
- The size and location of both types of **Conveyors** can be manipulated via handles at either end of the **Conveyor** segment (the crossed red arrows) or, more precisely, in the **Properties** window.
- The radius of **Curved Conveyor** objects can be changed either by the radius handle, the double-headed green arrow located in the middle of the conveyor segment, or, more precisely, in the **Properties** window.
- The direction of flow of a **Conveyor** can be changed using the double-green-arrows button on the **Properties** window, as highlighted by the red circle in the figure above.

The following figure provides more detail on the properties of the **Curved Conveyor**. Again, these properties are changed either through the resizing handles on the objects or the **Properties** window. The figure below provides a few examples of how these properties affect the location and shape of a **Curved Conveyor**.

- **Radius** is the size of the curvature of the conveyor, as measured from the center of a hypothetical circle to the midpoint of the conveyor width.
- **Start Angle** is the rotation, in degrees, to the location of the Start of the conveyor.
- **Sweep Angle** is the rotation, in degrees, from the Start to the End of the conveyor.

Also, in the figure below:

- Case A shows a conveyor segment that starts 90 degrees below the horizontal (**Start Angle** = -90) and is formed by sweeping up to the horizontal (**Sweep Angle** = 90) with a radius of 4 grid units (meters).
- Case B is the same as Case A but with a smaller radius.
- Case C is the same as Case A but starts at a different location (**Start Angle** = 0).
- Case D is the same as Case A, except it sweeps in the opposite direction (**Sweep Angle** = -90).



It is important to note that some **Conveyor** properties are not independent. For example, in a **Straight Conveyor**, the **End Location** and **Horizontal Length** are related; i.e., changing the **Length** automatically changes the **End Location**. Similarly, for a **Curved Conveyor**, the **End Location** and **Radius**, **Start Angle**, and **Sweep Angle** are related; i.e., changing either the **Radius**, **Start Angle**, or **Sweep Angle** automatically changes the **End Location**.

Obtaining the proper shape involves experimenting with the settings.

Two objects related to **Conveyors** are used in the example model – a **Decision Point** and a **Photo Eye**, both of which are objects that are placed on **Conveyors**.

A **Decision Point** is an object that allows logic to be placed on a **Conveyor**. The logic is typically implemented through a **Trigger** that fires when an item enters the **Decision Point** or leaves it.

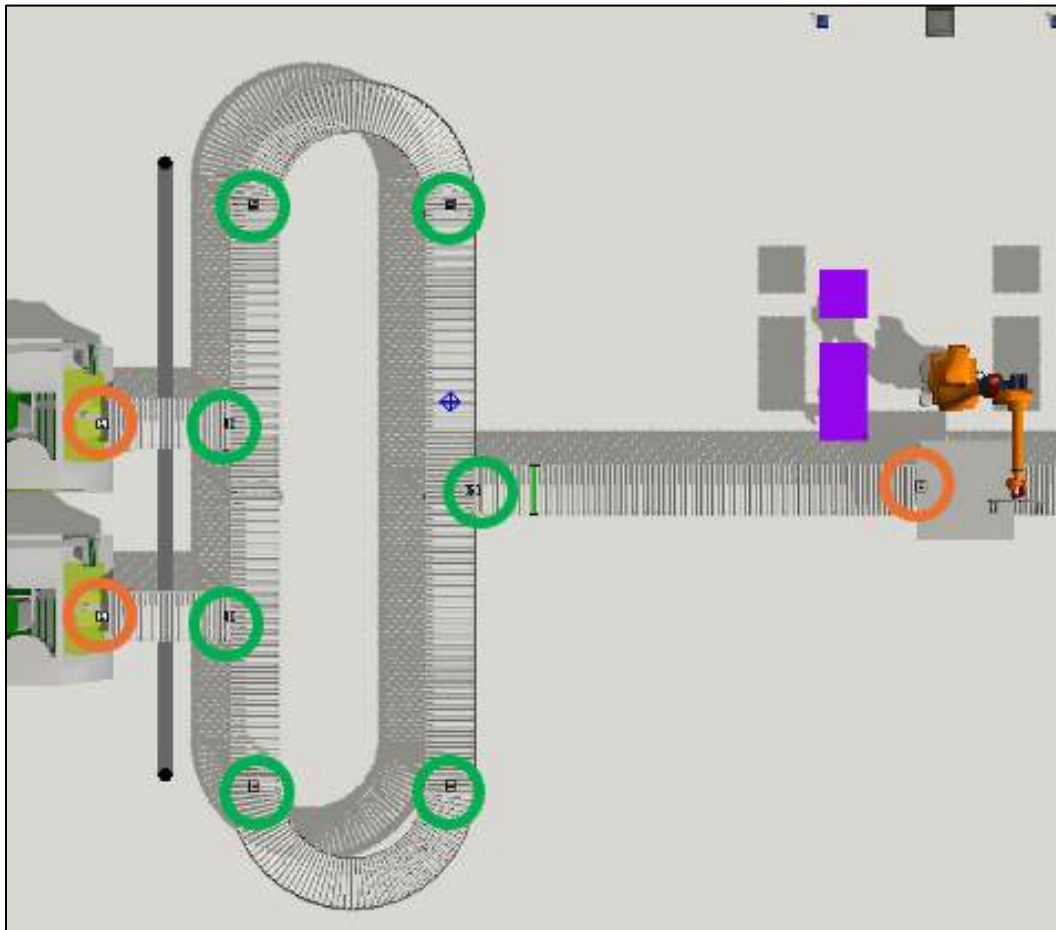


A **Photo Eye** is an object that also allows logic to be placed on a **Conveyor**, and its logic is also implemented through a **Trigger**. For example, the logic fires when an item blocks or covers the **Photo Eye**; similarly, logic fires when a **Photo Eye** is not covered or blocked.

The base model for the additions described in this chapter is **Primer_11** that was saved at the end of Chapter 18. However, a copy of that file was saved as **Primer_12**; thus, we begin with that file.

19.2 Incorporating conveyors into the primer model

The conveyor system for the primer model is shown in the figure below. It links the Finishing and Packing Areas. The conveyor loop provides buffer capacity if work builds up in the Packing Area. Containers move from the Finishing Machines onto the loop conveyor and proceed to the conveyor segment leading to the Packing Area. If the Packing Area segment is full when a container arrives, it continues on the loop and tries the segment again on the next pass. This process continues until the container enters the Packing Area. This loop may move to the Packing Area in a later design, but this is sufficient to introduce **Conveyor** objects.



Note in the figure above the **Entry/Exit Transfers** that are highlighted by orange circles; they link **Fixed Resources** and **Conveyor** segments. Also, note the **Conveyor Transfers** that are highlighted by green circles; they link **Conveyor** segments.

As described earlier, when precisely placing objects on the modeling surface, it is often best to work in a two-dimensional (2D) environment without perspective. Therefore

- Set the model view to 2D by right-clicking on the modeling surface, selecting **View**, and then **Reset View**.
- Turn off the perspective by right-clicking on the modeling surface, then unchecking the *Perspective Projection* box in the *View Setting* pane of the **Properties** window.

The major steps for modeling the conveyor system are as follows.

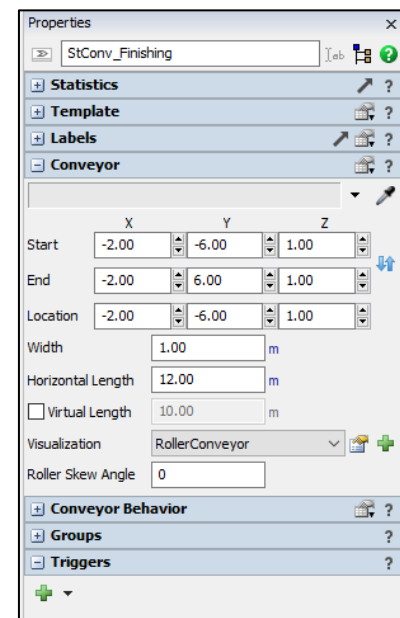
1. Construct a loop using **Straight** and **Curved Conveyor** objects.
2. Use **Straight Conveyors** to add spur lines to the loop, from the Finishing Machines and to the Packing Area.
3. Use a **Decision Point** and **Photo Eye** to add control logic so containers go to packing if there is space on that spur line. If the packing line is full, containers loop around until space is available on the packing spur.

The loop conveyor system will replace the **Conveyor** segment named *FinishToPack*.

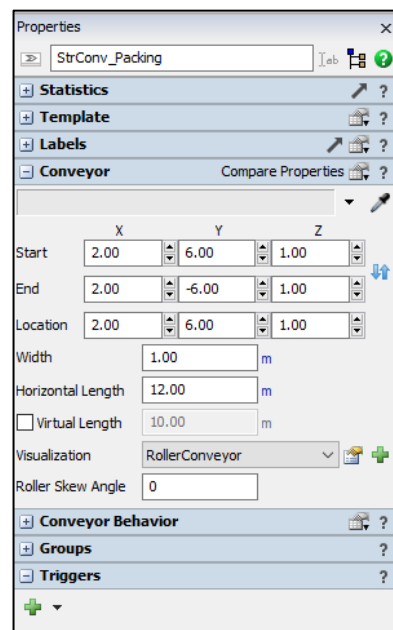
- Select the **Conveyor** *FinishToPack* and press the Delete key. Deleting the object automatically deletes all of its connects, from the Finishing Machines and to the Packing Station.

Construct a loop using **Straight** and **Curved Conveyor** objects.

- Select the **Straight Conveyor** object from the **Library**. This will be considered the straight conveyor near the Finishing Area.
- Click on the modeling surface near the *Start* location of the **Conveyor**.
- Click on the modeling surface near the *End* location of the **Conveyor**.
- Press the **ESC** key on the keyboard to get out of “add conveyor” mode. If **ESC** is not pressed, each click on the modeling surface will repeat the two steps above and continue to add conveyor segments.
- Name the object *StConv_Finishing*.
Again, objects can have any name, but it makes it easier to reference objects when they have meaningful names. How and where object referencing is used involves more advanced topics, but it is important to build good modeling practices early in the learning process.
- Position this 12-meter segment just to the right of the **Divider** object *FM_Divider*. Its precise position information is shown in the figure to the right.

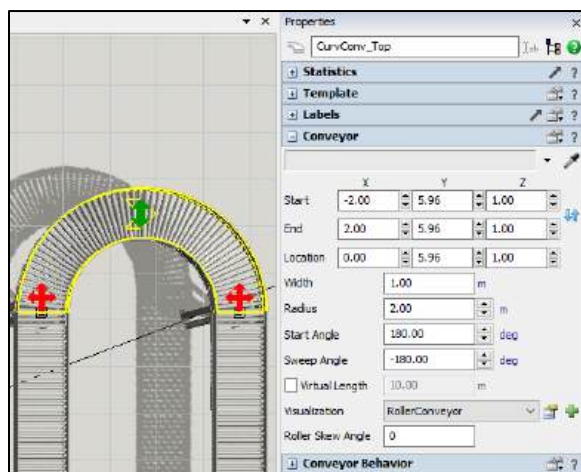


- Copy and paste the **Straight Conveyor** and position it about four meters to the right of *StConv_Finishing*. It will be the straight conveyor near the Packing Area.
- Rename the object *StConv_Packing*.
- Its precise position information is shown in the figure to the right.
- Use the double-green-arrow button to reverse the direction of flow on the **Conveyor**.



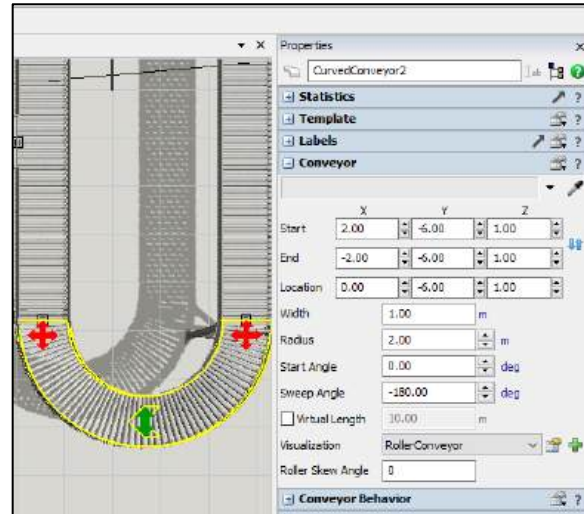
The following steps will connect the two straight segments with a **Curved Conveyor**, and the properties of the **Curved Conveyor** segment should be similar to that shown in the figure to the right.

- Select the **Curved Conveyor** segment from the **Library**.
- Click near the top of the **Straight Conveyor** *StConv_Finishing*. This will be the **Start** of the **Curved Conveyor**.
- Click near the top of the **Straight Conveyor** *StConv_Packing*. This will be the **End** of the **Curved Conveyor**.
- Rename the object *CurvConv_Top*.
- Check that the properties are similar to those shown in the figure to the right.
- Check that the three conveyor segments are connected via **Conveyor Transfers**.



Similarly, the following steps will again connect the two straight segments with another **Curved Conveyor**, thus creating a loop. The properties of the **Curved Conveyor** segment should be similar to that shown in the figure to the right.

- Select the **Curved Conveyor** segment from the **Library**.
- Click near the bottom of the **Straight Conveyor** *StrConv_Packing*. This will be the **Start** of the **Curved Conveyor**.
- Click near the bottom of the **Straight Conveyor** *StrConv_Finishing*. This will be the **End** of the **Curved Conveyor**.



- Rename the object *CurvConv_Bottom*.
- Check that the properties are similar to those shown in the figure to the right.
- Check that the four conveyor segments are connected via **Conveyor Transfers**, thus forming a closed loop with the flows going clockwise.

Add three spur lines to the loop.

The spur lines connect the two Finishing Machines and the packing station to the loop.

The Finish Machines put finished containers onto the conveyor for transport to packing.

- Select the **Straight Conveyor** segment from the **Library**.
- Click anywhere on the modeling surface to set the **Start** or left end of the segment.
- Click on the modeling surface about 2.5 meters to the right of the **Start** location, creating a horizontal conveyor spur.
- Press the **ESC** key on the keyboard to get out of “add conveyor” mode. Recall that if **ESC** is not pressed, each click on the modeling surface will repeat the two steps above and continue to add conveyor segments.
- Name the object *StrConv_FM_1*.
- Make an A-connection from the **Processor** *FinishMach_1* to the spur, **Straight Conveyor** *StrConv_FM_1*. Note the automatic creation of the **Entry Transfer** object on the spur segment.

- Move the segment into position as shown in the figure to the right.
- Position the segment so that the spur, **Straight Conveyor** *StrConv_FM_1*, automatically connects to the loop conveyor *StConv_Finishing*. Note the connection is made once a **Conveyor Transfer** object links the two conveyor segments.



Create the spur for the second Finish Machine, which is shown already connected in the figure to the right.

- Copy and paste the spur created above, **Straight Conveyor** *StrConv_FM_1*, anywhere on the modeling surface.
- Name the object *StrConv_FM_2*.
- Make an A-connection from the **Processor** *FinishMach_2* to the new spur, **Straight Conveyor** *StrConv_FM_2*. Note the automatic creation of the **Entry Transfer** object on the spur segment.
- Move the segment into position, as shown in the figure above.
- Position the segment so that the spur, **Straight Conveyor** *StrConv_FM_2*, automatically connects to the loop conveyor *StConv_Finishing*. Note the connection is made once a **Conveyor Transfer** object links the two conveyor segments.

Create the spur from the loop to the packing station.

- Copy and paste the spur created above, **Straight Conveyor** *StrConv_FM_2*, anywhere on the modeling surface.
- Name the object *FinishToPack*.
- Make an A-connection from the **Conveyor** *FinishToPack* to the packing station, **Combiner** *Packing_1*. Note the automatic creation of the **Exit Transfer** object on the spur segment.
- Move the segment into position, as shown in the figure below, and ensure it connects with the loop conveyor. If the two segments are near each other, a **Conveyor Transfer** object will automatically join them.



Add control logic

The control logic will ensure containers only go to packing if there is space on the spur line. If the packing line is full, containers remain on the conveyor loop until space is available on the packing spur. The logic is implemented using the **Photo Eye** and **Decision Point** objects.

Add a **Photo Eye** to the packing spur to detect when containers are backing up on the conveyor.

If a container covers the photo eye, then the conveyor is considered full, and no more containers should be sent down the spur until the photo eye is uncovered.

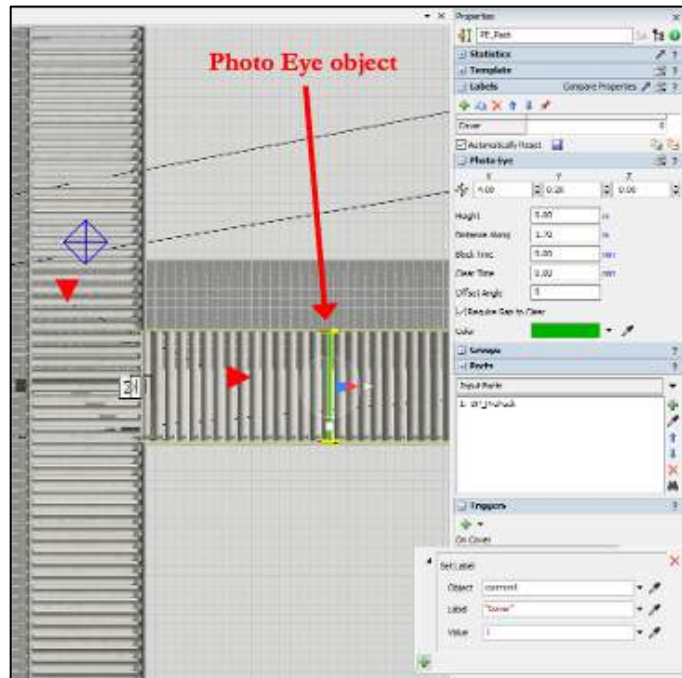
- Drag out a **Photo Eye** object from the **Library** and place it on the spur conveyor *FinishToPack*, as shown in the figure to the right.

- Name the object *PE_Pack*.

- Using the  button on the **Labels** pane, select *Add Number Label* to create a label.

- Name the label *Cover*.
- Use the default value of *0*.
- Check the **Automatically Reset** box. This will set the label value back to *0* whenever the model is **Reset**.

The label **Cover** will have a value of *1* when the **Photo Eye** is covered by a container and *0* when it is not.



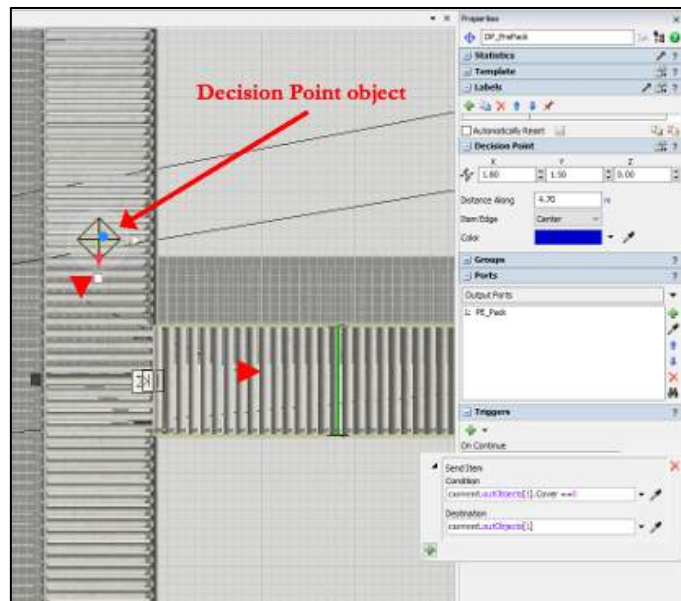
- Create an *On Cover* trigger using the values in the figure shown to the right.

When the **Photo Eye** is covered by a container, the **Cover** label value will be set to *1*.

- Create an *On Uncover* trigger using the same values as above, except **Value** is 0.
When the **Photo Eye** is not covered by a container, the **Cover** label value will be set to 0.
- Notice in the **Ports** pane that this **Photo Eye** is connected to a **Decision Point** that is created in the next step.

Add a **Decision Point** to check to see if containers are backed up so far on the packing spur that it sends the current container to the spur. As defined above, if a container covers the photo eye, then the conveyor is considered full, and no more containers should be sent down the spur until the photo eye is uncovered. Therefore, the **Decision Point** sends the current container straight ahead so it remains on the loop.

- Drag out a **Decision Point** object from the **Library** and place it on the **Straight Conveyor** *StrConv_Packing*, as shown in the figure to the right.
- Name the object *DP_PrePack*.
- Make an A-Connection from the **Decision Point** to the **Photo Eye** defined above.
Note the connection is shown in the **Input Ports** part of the **Ports** pane.
- Create an *On Continue* trigger using the values shown in the figure to the right. This is a *Send Item* trigger that decides where to send the item based on current conditions.



The **Condition** it evaluates when an item enters the **Decision Point** is
`current.outObjects[1].Cover == 0`

This is a test to see if the value of the label named **Cover** on the object that is connected to **Decision Point's** first output port, which is the **Photo Eye** object, is equal to 0.

If the label value is equal to 0, meaning the **Photo Eye** is uncovered, then the container is routed to the **Destination** whose value is `current.outObjects[1]`. The destination is the **Photo Eye** object, which puts the container on the spur conveyor to the Packing Area.

If the label value is not 0, meaning the **Photo Eye** is covered, then the container is not routed to packing, and it continues on the conveyor it is currently on and goes around the buffer loop.

Add all of the **Conveyor** objects to the **A* Navigator** tool so that the operator avoids these objects when traveling.

- Open the **A* Navigator** tool and select *FR Members* in the **Members** section of the interface.
- Using the  button, select all **Straight Conveyor** and **Curved Conveyor** objects.

➤ **Reset** and **Run** the model.

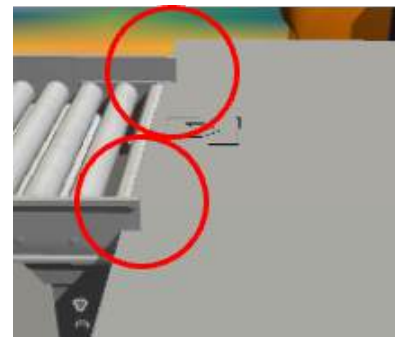
The operation should resemble the figure shown below.

Note how the containers move on the conveyor system. Under the current model's settings, it is unlikely the Packing Area conveyor will back up enough to cause containers to be routed around the loop.

Also, note the Finish Operator's travel path as shown by the **A* Navigator's** heat map.



If the **Operator** travels through an object and the object is in the **A* Navigator's** list, make sure there is no gap between objects. For example, the **Operator** might pass through the Conveyor and Packing Table. In this case, note the slight overlap between the two objects that is highlighted by the red circles in the figure to the right. If there is a gap, a **Task Executer** will pass through no matter how small the gap; i.e., the **TE's** size is not considered.



If you haven't already done so, save the model. Recall that it is good practice to save often.



Use the **Save Model As** option in the **File** menu to make a copy of the existing model to be further customized in the next section. Again, you can use any file name, but in the primer, the next model is referred to as Primer_13.

20 USING AN ITEM LIST FOR COMPLEX MODELING LOGIC

Chapter 20 adds more complex routing logic in the Finishing Area in terms of how containers are selected for finishing. The logic is implemented using the List tool. A few Parameter values are updated prior to the logic change and two charts are added to track the contents of the conveyors.

In the Finishing Area of the primer example, the current model uses the default logic in the **Queue** object to determine the order in which containers are picked up by the Finishing Operator. That default logic is first-come, first-served (FIFO); i.e., items are processed in the order in which they arrive. In other words, the items that have waited the longest are processed first. This is a common way for queues to operate; many human systems operate this way because it is perceived to be the fairest.

However, there are exceptions to this means of processing, such as when items are processed by priority. For example, patients in a hospital waiting room do not follow the FIFO rule - patients are triaged based on the seriousness of their situation. Also, customer orders are often processed before those that have waited longer if they are designated as close to a due date or late, or if the customer paid a premium to get the order processed quickly.

The container processing logic will be changed in this system.

First, it will be changed to process containers by using the shortest processing time (SPT) rule – process the items that are the quickest to process first. In production control theory, processing by SPT results in more throughput than by using FIFO. Based on the process times, the preferred order for finishing containers is: Type=1 first, then Type=2, and then Type=3. (Recall that finishing times are 15.0, 20.0, and 30.0 minutes for types 1, 2, and 3, respectively.)

One problem with SPT is that some containers, Type 3 in this case, may wait too long. Therefore, the new processing rule will be to use SPT, but if a container waits more than a threshold, say 30 minutes, then it will be processed first.

However, before implementing this new processing rule, some of the operational parameters are updated based on revised engineering estimates. This also provides a refresher on some aspects of the model.

The base model for the additions described in this chapter is Primer_12 that was saved at the end of Chapter 19. However, a copy of that file was saved as Primer_13; thus, we begin with that file.

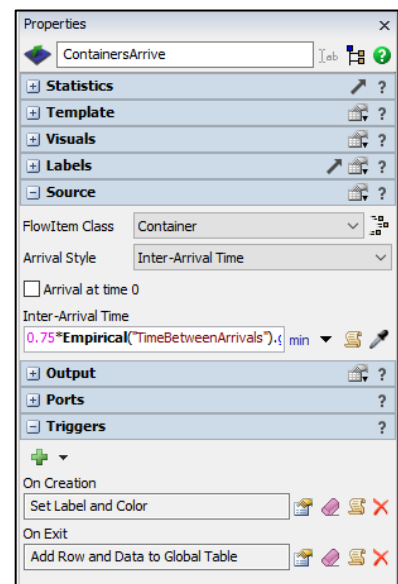
20.1 Parameter updates

Three main factors are driving the updating of some of the model parameters:

1. Engineering believes the Finishing Area can process more than the current production rate with two machines and one operator. It estimates it can accommodate 25 percent more production without exceeding the new container buffer size of 10.
2. Marketing sees a shift in the product mix with less demand for Type 1 and more demand for Product 3. They estimate the mix will be 30% Type 1, 35% Type 2, and 35% Type 3.
3. Based on the two changes above, Engineering has updated the frequency and batch sizes for the components. One change is that both components will be replenished hourly. Recall that Component A was replenished hourly, but Component B was every half hour.

Implement the changes as follows.

- Increase the production rate of the containers. If the average rate increases 25%, then the average time between arrivals decreases by 25%. Assume the same **Empirical Distribution** is relevant; just the sampled values need to be reduced by 25%. Therefore, multiply the **Empirical Distribution** value by 0.75 in the **Inter-Arrival Time** property on the **Source's Source** pane, as shown in the figure to the right.
- Change the value of **Buffer Size** to 10 in the **General Parameters** table.
- Update the values in the **Empirical Distribution Product Mix** to 0.30, 0.35, and 0.35 for Container Types 1, 2, and 3, respectively.
- Change the frequency value for **CompFreq_B** to 60 in the **ComponentFrequency Parameters Table**.
- Change the batch size values as follows:
 - **CompBatSz_A** to 4 in the **ComponentBatchSize Parameters Table**.
 - **CompBatSz_B** to 11 in the **ComponentBatchSize Parameters Table**.



To help in the subsequent analyses, add two new charts to a new Dashboard. One will track the number of containers on the packing spur conveyor over time, and the other will track the number of containers on the loop conveyor between the Packing and Finishing Areas.

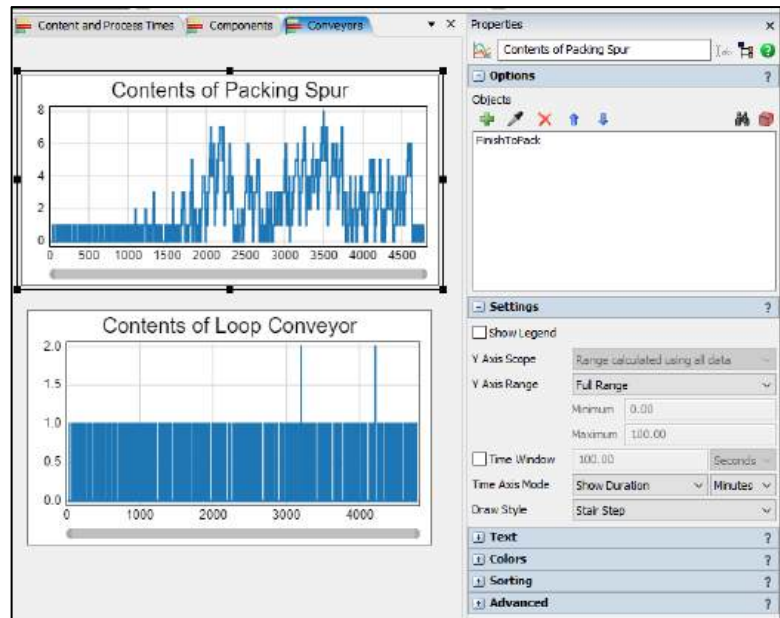
- Create a new **Dashboard** and name it *Conveyors*.

Based on the figure to the right, build a chart to display the number of containers on the packing spur conveyor over time.

- Select a *Content* chart from the **Content** section of the **Dashboard Library** and drag it onto the new Dashboard.

- Name the chart *Contents of Packing Spur*.

- Define the chart in the **Options** and **Settings** pane in the **Properties** window shown in the figure to the right.

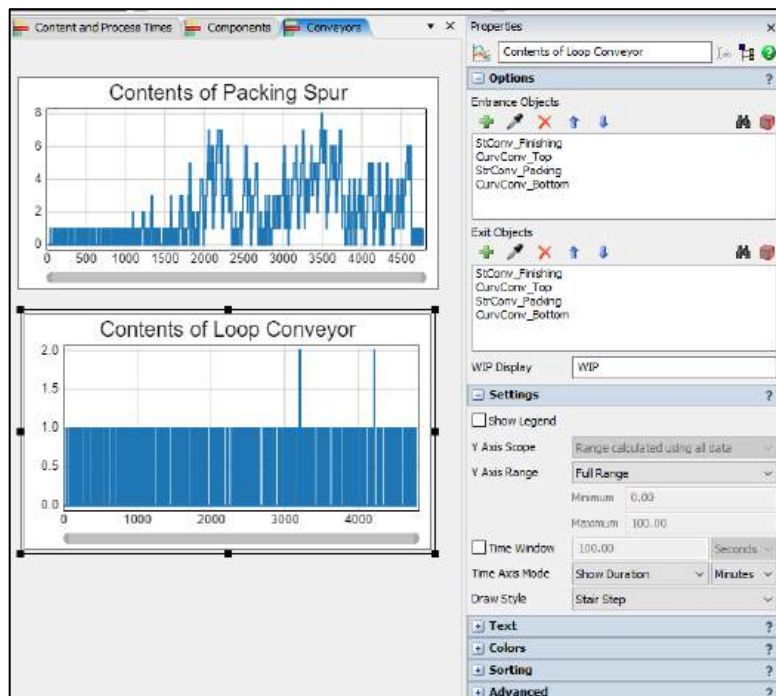


Based on the figure to the right, build a chart to display the number of containers over time on the loop conveyor between the Finishing and Packing Area.

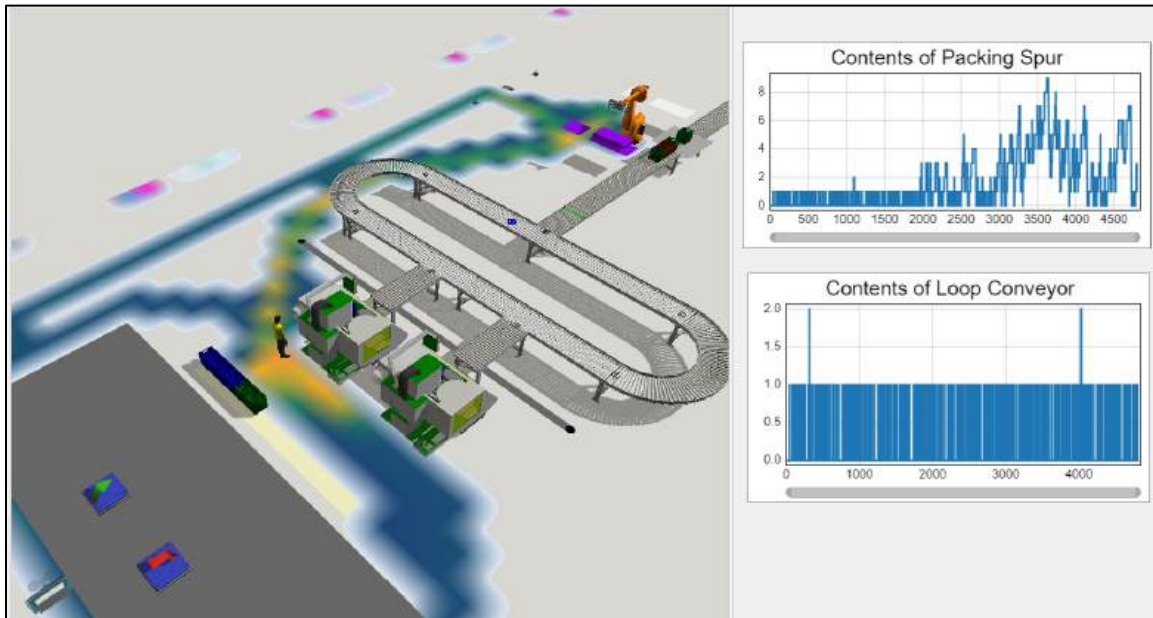
- Select a *Composite WIP* chart from the **Content** section of the **Dashboard Library** and drag it onto the new Dashboard.

- Name the chart *Contents of Loop Conveyor*.

- Define the chart in the **Options** and **Settings** pane in the **Properties** window shown in the figure to the right.



- **Reset** and **Run** the model. Observe the model behavior and the charts. A screenshot during a model run is shown in the figure below.



If you haven't already done so, save the model. Recall that it is good practice to save often.

20.2 Simple priority routing using an object trigger

To get more throughput, DPL wants to process containers by using the shortest processing time (SPT) rule, i.e., process the quickest items to produce first. Based on the process times, the preferred order for finishing containers is: Type=1 first, then Type=2, and then Type=3.

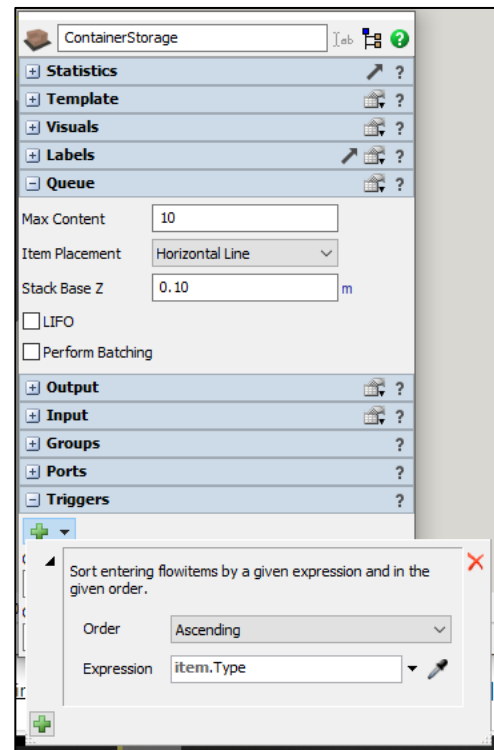
To represent this in *FlexSim* is quite easy. As shown in the figure to the right, the contents of the **Queue** can be reordered whenever an item enters.

- In the **Triggers** section of the **Queue ContainerStorage** **Properties** window, add an **OnEntry** trigger via the  button.

- Use the  button to the right of *On Entry* textbox, select the **Control** category, then *Sort by Expression*.

- As shown in the figure to the right, change the **Order** property from the default *Descending* to *Ascending* using the dropdown menu. Ascending is chosen because the Type=1 containers should be processed first, then Type=2, etc.; i.e., items should be arranged in the **Queue** by increasing/ascending values of the label *Type*.

- The default **Expression** property, *item.Type* is used as the sort criteria, i.e., the value of the label *Type* on the items in the **Queue**. If multiple items of the same type are in the **Queue**, then they are arranged in the order in which they arrived. Therefore, two ordering rules are in effect: first, order by type, then by age.



Note that this approach changes the order of the items in the **Queue** – the contents of the **Queue** are sorted each time an item arrives to the **Queue**. The item at the front of the queue is sent to the first available object (a **Processor** in this case). Thus, containers are finished (processed at the **Processors**) based on the SPT priority rule. Many systems behave this way.

However, other systems maintain the order in which items arrive and are selected from the **Queue** based on the SPT rule. This would be the case if items arrive on a conveyor - items maintain the order in which they arrive since they cannot physically be reordered. While in many cases, this is a subtle difference, it can affect performance. For example, in this case, if a priority container is the last to arrive, then the Finishing Operator needs to travel to the end of the **Queue** to load the container rather than taking it from the front of the queue, resulting in a longer travel time. Of course, the level of detail is always an important modeling decision. In many cases, it might be best to use the quick change as done above, note an assumption that this does not significantly impact performance, and then revisit the assumption later.

The use of the **List** tool, described in the next section, does not reorder the queue; it just identifies the next item to process based on a specified rule.



If you haven't already done so, save the model. Recall that it is good practice to save often.



Use the **Save Model As** option in the **File** menu to make a copy of the existing model to be further customized in the next section. Again, you can use any file name, but in the primer, the next model is referred to as **Primer_14**.

20.3 Multiple-criteria routing using Lists

One drawback of using the SPT rule in this example is that the Type 3 containers may wait a very long time. This occurs because if any Type 1 or 2 items are in the **Queue** when a Finishing Machine becomes available, they will be processed before any Type 3. This issue is addressed by using the following rule:

Process containers based on their Type unless some items have waited “too long,” then process them first.

Of course, “too long” must be specified, such as one hour. This is called the **Wait Threshold**. The **List** tool is used to implement this more complex rule.

Prior to introducing **Lists**, two new constructs, “pull” logic and “push” logic, are defined.

- **Push logic**. In the current model, items are “pushed” from the **Queue** to the **Processors** (from *ContainerStorage* to *FinishMach_1* and *FinishMach_2*); i.e., the routing decision is in the **Queue**, and it pushes items to the first available **Processor** by default or by a rule specified in the **Send To Port** trigger on the **Output** pane of the **Queue**. The default rule is *First Available*, but many others are available on the dropdown menu, such as *Random*, *Round Robin* (alternate between machines), *By Expression* (using the value of a label, such as Type), etc.
- **Pull logic**. The routing decision between the **Queue** and **Processors** is moved from the **Queue** to the **Processors**. **Processors** pull items based on specified criteria rather than have items pushed from the **Queue**. The pull logic is enabled on the **Processor's Input** pane by checking the **Pull** box. The values for the **Pull Strategy** and **Pull Requirement** are then specified.

The base model for the additions described in the remainder of this chapter is **Primer_13** that was saved at the end of Section 20.2. However, a copy of that file was saved as **Primer_14**; thus, we begin with that file.

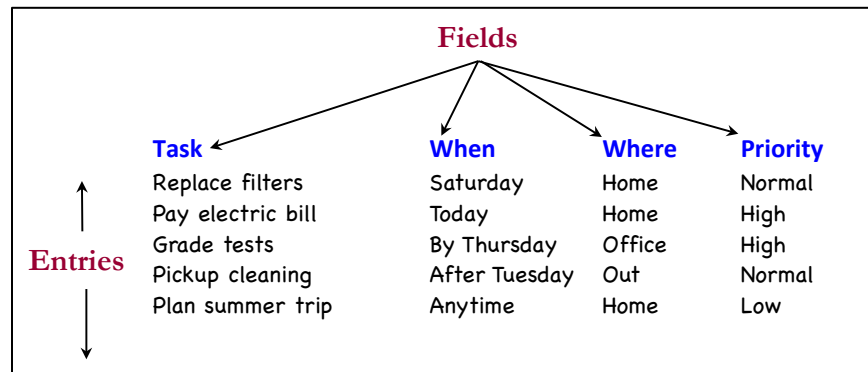
Before proceeding with the model, it may be better to turn off the heat map from the **A* Navigator** tool.

➤ Uncheck the **Show Heat Map** box on the **Visual** tab.

Lists, also called **Global Lists**, have many uses in modeling. **Lists** are tools that can create complex flows between objects. There are many types of **Lists** available in *FlexSim* - **Fixed Resource List**, **Item List**, **Task Sequence List**, **Task Executer List**, and **General List**. In this example, an **Item List** is used to implement the routing logic specified above.

Information is pushed to and pulled from lists when events occur. Typically, an event causes an entry to be placed on a list, and another event causes an entry to be pulled or removed from a list. The information about an entry is contained in fields. The overall structure of a list is like a table.

As a simple example, consider the simple To-Do list in the figure below. An entry is something that needs to be done and is a row in a list. Entries are put on (pushed to) a list as a result of an event occurring. For example, in the To-Do List, when we think of something or are told to do something, we make an entry onto or push to the list. Entries are typically added at the bottom of the list. Entries are removed (pulled) from a list again when an event occurs. For example, in the To-Do list, an entry is removed or pulled from the list when it is completed or deemed no longer necessary to be done.



The columns in the list are called fields and represent characteristics of entries, such as in the To-Do List, a brief description of what needs to be done (**Task**), when it needs to be done (**When**), where it needs to be done (**Where**), and how important it is (**Priority**). Thus, each entry has a *value* associated with each field.

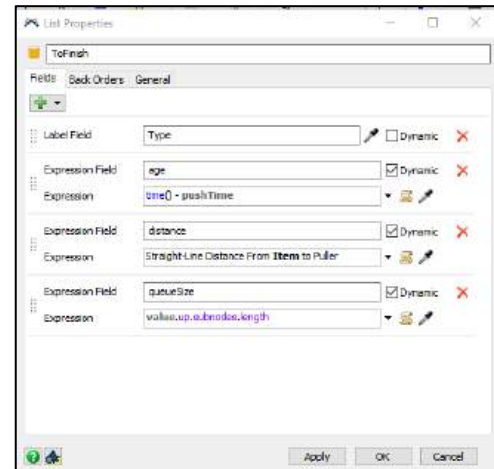
In the case of the primer example, as containers enter their storage **Queue**, the **Queue** *pushes* information about the item to a **List**, i.e., a list entry is made. When available, **Processors** scan all of the list entries and pull an item to process based on specified criteria. Scanning the list and selecting an item is referred to as a query.

The threshold value for the longest time an item should wait is stored in a **Parameters Table**. This way, the threshold value can be easily changed. Simulation experiments with this model will be used to set the threshold value to find the right balance of maintaining the SPT rule but not allowing items to wait too long.

- Add a row to the **Parameters Table** named *GeneralParameters*.
- Name the parameter *Threshold*.
- Clicking in the **Value** cell brings up a button to define the following
 - **Lower Bound** 15 minutes
 - **Upper Bound** 120 minutes
- Set the current **Value** to 30.
- Use the  button in the **Toolbox** to create an **Item List** from the **Global List** tool in the **Toolbox** and name it *ToFinish*, as shown in the figure to the right.

Fields must be defined for the **List** and how their values are determined for each entry, i.e., what information about the item is put onto the list and how that information is determined.

Fields may be of different types, e.g., a label value on the item or a calculation using an expression that is defined in each field's **Expression** property.



By default, *FlexSim* provides four fields, as shown in the figure to the right above and defined below. Fields are either **Dynamic**

(updated as affected states change in the simulation) or not (use and maintain the current value).

- **Type** is the value of the item's label named *Type* at the time it is placed on the list (not **Dynamic**).
- **age** is how long an entry has been on the list. The default value is calculated based on the **Expression** that is defined as `time() - pushTime`,
 This means that the age of an item on this list is the current simulation time, `time()`, minus the simulation time when the entry was put onto the list, `pushTime`. Since the **Dynamic** box is checked, the value is constantly updated while the entry is on the list.
- **distance** is how far the item currently is from the object that will pull the entry from the list. The default value is calculated based on the **Expression** that determines the straight-line distance between the item and the pulling object. Since the **Dynamic** box is checked, the value is constantly updated while the entry is on the list.
- **queueSize** is how many items are in the object where the item is currently located. The default value is calculated based on the **Expression** value `up.subnodes.length`. Since the **Dynamic** box is checked, the value is constantly updated while the entry is on the list.

In the primer example, only the first two fields are needed. Therefore, the last two can be deleted. Of course, there is no problem leaving them on the **List**.

- Delete the *distance* and *queueSize* fields by using the  button next to the **Dynamic** property checkbox for that field.

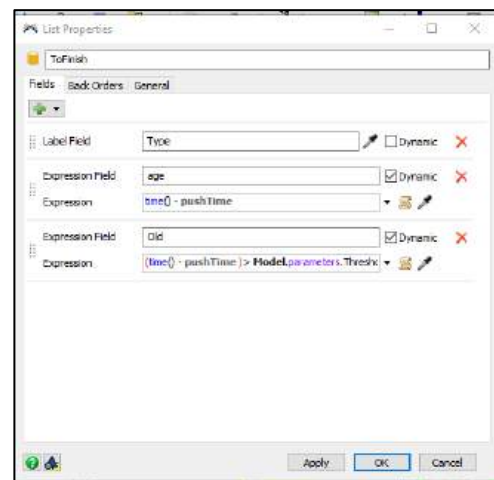
In this example, additional information beyond the default fields is needed. The routing logic is based on whether an item has waited longer than a threshold value, not just how long it has waited, which is what the *age* field contains.

- Use the  on the **List Properties** window, just under the **Fields** tab name, and select *Expression*.
This creates a new **Expression** field that will indicate if an item is “old,” i.e., has been on the **List** longer than the threshold value specified in a **Parameters Table** named *GeneralParameters*.

As shown in the figure to the right:

- Replace the default field name (*fieldname*) in the **Expression Field** with the name *Old*.
- Check the **Dynamic** box.
- Replace the 0 in the **Expression** box with the following:
(time()-pushTime) > Model.parameters.Threshold
 - The expression to the left of the > (greater than sign) can be copied and pasted from the *age* expression above. Be sure the expression is within parentheses (...).
 - The context-sensitive editing in *FlexSim* facilitates entering the reference to a model parameter.

The “Old” expression checks to see if the amount of time the entry has been on the **List** exceeds the threshold value (the **Parameter** named *Threshold* that is stored in the *GeneralParameters Model Parameters Table*). If the time on the **List** exceeds the threshold, then the comparison is *true*, and the field will have a value of 1. If the entry is not on the **List** longer than the threshold value, then the comparison is *false*, and the field will have a value of 0.



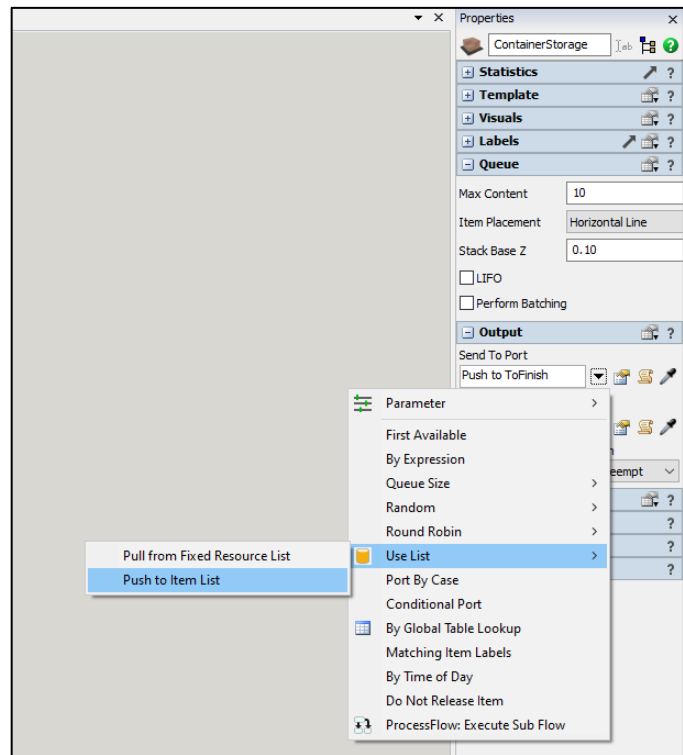
Now that the **List** is defined, the model needs to push information to the **List** and pull information from the **List** to decide which container to process next.

Push an item's information to the **List** from the **Queue ContainerStorage**, as shown in the figure to the right.

- In the **Output** pane, use the dropdown menu on the **Send To Port** trigger to select *Use List* and then *Push to Item List*. Make sure the **ToFinish** list is shown in the menu.

- While in the **Queue** object, use the outer  button to remove the **On Entry** trigger *Sort by Expression*.

When doing so, a dialogue box will appear asking, “Remove this trigger and its logic?”; press the *OK* button. The **List** logic is now controlling the item selection from the **Queue** so the **Queue** no longer needs to be sorted when items enter.



The Finishing Machines will use information in the **List** to decide which container to process next. It will:

- Query the current **List**
- Select an entry based on the specified criteria
- Pull the information from the **List**
- Pull an item from the **Queue** based on the information

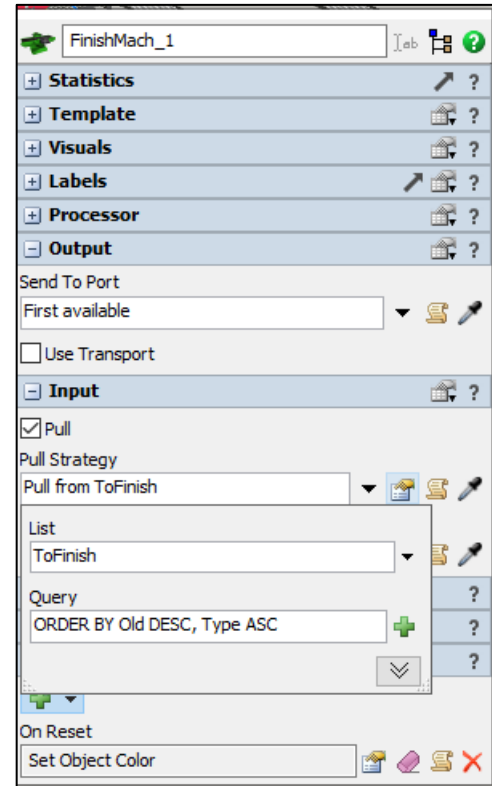
To implement the above logic, follow the instructions below and as shown in the figure to the right. The changes are made on the **Input** pane of the **Processors'** (both Finishing Machines) **Properties** window.

Follow the steps for *FinishMach_1*, then repeat them for *FinishMach_2*.

- In the **Input** section, check the **Pull** box.
- For **Pull Strategy**, use the dropdown menu to select *Use List*, then *Pull from Item List*.
Be sure the **List** property value is set to *ToFinish*.

Two sort queries are entered using the  button to the right of the textbox. The first query sorts the **List** based on the **Old** field; the second query then sorts by the **Type** field. To do this:

- Select *Order By (Sort)* from the dropdown menu, and then select *Old*.
This results in ORDER BY Old ASC
- Change *ASC* to *DESC*.
- Again, using the  button, select *Order By (Sort)* again from the dropdown menu, and then *Type*. Leave the default *ASC*.
This results in the Query command: ORDER BY Old DESC, Type ASC

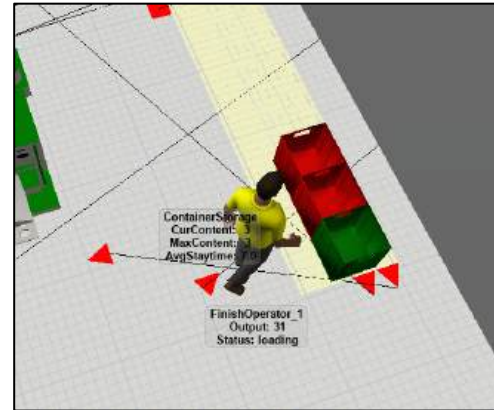


This command uses SQL (Structured Query Language) to sort the **List** and place the entries in the specified order. The **List** is first sorted in descending (*DESC*) order of the values in the field **Old**. The values with **Old=1** (indicating they have been on the **List** longer than the threshold) are placed at the top of the list, and those with **Old=0** are at the bottom of the **List**. The **List** is sorted again by the values of item's **Type** label, and they are put in ascending (*ASC*) order of **Type**. As a result of the query, any entries in the list that are beyond the threshold (**Old=1**) and a high priority (**Type=1**) will be at the top of the list. The priority here is the containers with the shortest processing time.

Each Finishing Machine pulls the flow item from the **Queue** that is associated with the entry at the top of the **List**, and the entry is then removed from the **List**.

- Be sure the above pull logic changes are repeated on the other Finishing Machine.

- **Reset** and **Run** the model. Notice, as in the figure to the right, the operator does select the intended item in the queue. The operator is moving to the red container (Type 1) for processing even though there is an item (green) in front that has waited longer.



- To view the current contents of the **List**, double-click on the *ToFinish* **List** in the **Toolbox**. As shown in the figures below, go to the **General** tab and select the **View Entries** button.

A list entry is an item that has been pushed to the list and is waiting to be pulled.

Note that in the screenshot to the left below, there are five containers waiting in FIFO order in the container storage queue. Three containers are considered “old,” so they will be selected before the others (Old=0). However, Type 1 is the higher priority, so based on our decision rule, it will be selected first. That entry is highlighted by the red box.

This result is shown in the figure to the right. Note in the figure that the Type 1 container was pulled from the List by a Finishing Machine. The pull was made as soon as Finishing Machine 2 became available; i.e., it commits to that item regardless of what happens after the pull. Also, note that the container is still in the storage area. This is because the Finishing Operator has not yet picked it up. In this case, the Operator was repairing Finishing Machine 1. Therefore, a few minutes later, the Operator will pick up the container and load it on Finishing Machine 2.

value	Type	age	Old
/ContainerStorage/Container 3	3	76.45	1
/ContainerStorage/Container 2	1	46.56	1
/ContainerStorage/Container 3	2	34.16	1
/ContainerStorage/Container 4	3	20.33	0
/ContainerStorage/Container 5	2	6.79	0

value	Type	age	Old
/ContainerStorage/Container 3	3	76.16	1
/ContainerStorage/Container 3	2	35.90	1
/ContainerStorage/Container 4	3	22.07	0
/ContainerStorage/Container 5	2	8.53	0

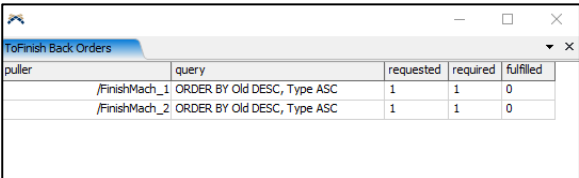
Consider another example that was observed as the model was running. In the screenshot to the left below, three containers are waiting in FIFO order in the container storage area. As per the **List Entries** table, none of the containers are considered “old,” so they will be selected according to the Type priority rule (SPT). In this case, the entry highlighted by the red box will be selected. This is actually what happens, as shown in the figure on the right below. The first red container (Type 1) was pulled by a Finishing Machine since its entry has been pulled from the list.



A **List backorder** is a puller that is waiting to pull an item from a List.,

- To view the current **List** backorders, double-click on the *ToFinish List* in the **Toolbox**. Then, go to the **General** tab and select the **View Back Orders** button.

In the figure to the right, which is from the start of a simulation, there are two **List** backorders, both Finishing Machines. They are on the list because they are idle – they want to pull items from the **List**, but there are none to pull.



If you haven’t already done so, save the model. Recall that it is good practice to save often.



Use the **Save Model As** option in the **File** menu to make a copy of the existing model to be further customized in the next section. Again, you can use any file name, but the next model is referred to as Primer_15 in the primer.

PART VI - MODELING USING PROCESS FLOW + WAREHOUSING AND AGV

This section uses *FlexSim's* powerful logic builder to complete the primer model. After introducing Process Flow, it is used to include initial component inventory in the model, implement a means to manage component inventory through a reorder-point system, and represent the order-fulfillment process. This section also introduces two additional *FlexSim* constructs that are key to modeling many operations systems – Racks and Warehousing objects and AGVs and their associated objects.

- Chapter 21 introduces Process Flow, including basic concepts and the modeling environment. It also describes some additions to the model that support the use of Process Flow.
- Chapter 22 describes several approaches to modeling the creation of an initial component inventory and explains how to implement the more general approach.
- Chapter 23 uses Process Flow to incorporate a reorder point inventory system to manage components.
- Chapter 24 introduces the Rack object and describes its use to store packed containers in the warehouse.
- Chapter 25 defines the order-fulfillment process and describes how to represent it in the model by using Process Flow. The process includes generating orders for containers, having an Order Picker gather the appropriate containers, delivering a completed order to a fulfillment area, and completing orders by updating an information system. Also, an output table is created that captures information on each order, including the contents of each order and the time it takes to fulfill orders. Charts have also been added that track the time it takes to fulfill an order and how many orders are waiting to be processed.

21 INTRODUCTION TO PROCESS FLOW

Chapter 21 provides an introduction to Process Flow, including basic concepts and the modeling environment. It also describes some additions to the model that support the use of Process Flow.

Process Flow is a major tool in *FlexSim* for developing complex logic. While *FlexSim*'s **3D Objects** include many properties and methods that can be used to develop quite sophisticated logic for modeling complex operations systems, Process Flow greatly expands those capabilities. Combining Process Flow and 3D Objects provides a highly robust and powerful simulation modeling environment. While Process Flow can be used to develop more complex processing logic within 3D Objects, it is especially valuable for modeling inter-object relationships.

Process Flow enables users to define and specify logic via a flowchart-like, drag-and-drop interface and link the logic with objects in the 3D environment. Since this primer is intended to provide a basic introduction to *FlexSim*, only a limited number of features and capabilities of Process Flow are described. However, this should provide a good starting point and a solid foundation for modeling with Process Flow.

After introducing the basic concepts of Process Flow and describing its modeling environment, the primer illustrates modeling in Process Flow via several examples. The first and simplest is to create an initial inventory of components so models do not start at an oftentimes unrealistic condition of “empty and idle,” where there is no work in process and all resources idle at the start of a simulation. The next example uses Process Flow to implement a reorder-point inventory system in the model. Later in the primer, Process Flow is used to model an order-fulfillment process. The latter two applications could be implemented via the 3D objects but it would be quite challenging. The implementation would also not be very transparent because so much of it would be implemented via custom coding in *FlexScript*. Transparency is greatly enhanced by using Process Flow since all of the logic is contained in one place, on the Process Flow interface, and not dispersed in 3D objects. In addition, the logic is more transparent because the logic is represented in an easy-to-understand flowchart-like format.

While the remainder of the primer focuses on Process Flow, the examples revisit aspects of the 3D model and introduce additional features in the 3D environment, e.g., Racks for storage and AGVs.

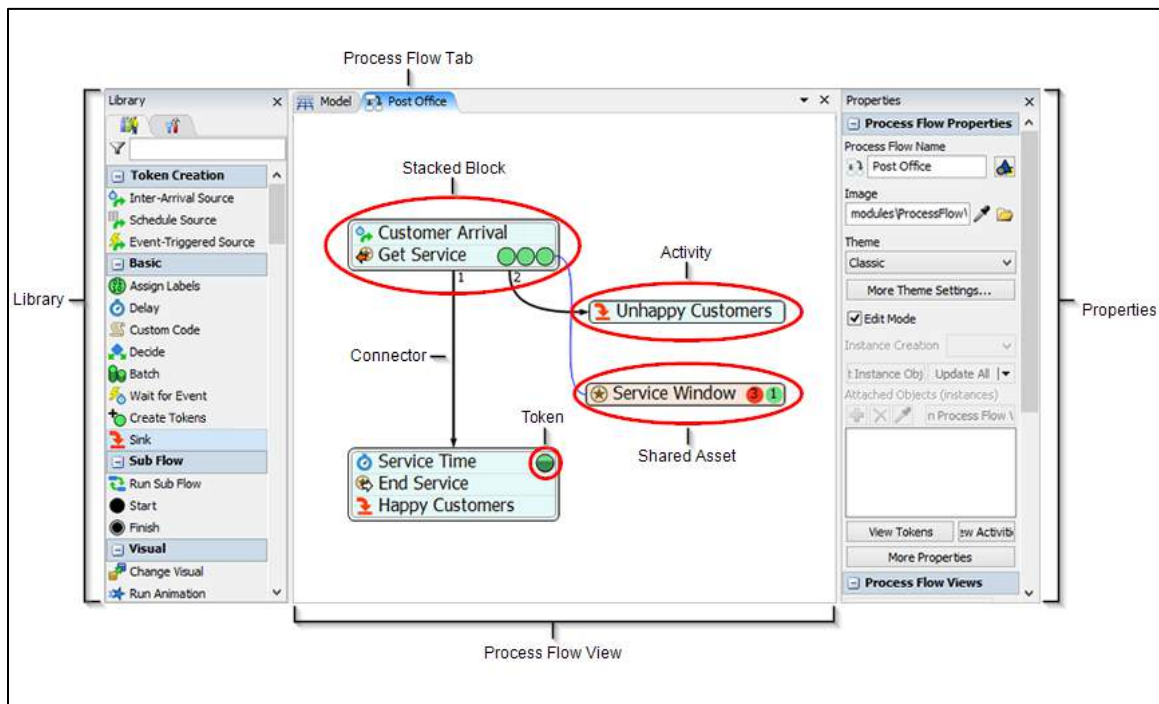
Again, this chapter introduces the basic elements of Process Flow and its modeling environment. While the interface and library of modeling tools are different than in 3D, the application is similar. As with **3D Objects**, **Process Flow** activities are dragged onto a modeling surface, connected to represent how a system behaves, and are customized by specifying relevant property values. This chapter also describes the basic operation of a reorder-point inventory system.

21.1 Basic Process Flow concepts and modeling environment

As noted above, Process Flow is a part of *FlexSim* that is used to add complex processing logic into a simulation model. The logic is used to define interactions among *FlexSim* objects and other elements as a simulation runs. While *FlexSim* provides a lot of logic-modeling capability through its drop-down menu options on 3D objects and tools, there is a limit as to how much can be pre-specified. Typically, custom logic requires scripting or writing computer code to implement the more complex logic. This can be done through *FlexSim*'s programming language, called *FlexScript*, a subset of C++. However, Process Flow reduces the need for programming by using a set of logic-building activities that can be defined and combined in a flowcharting type of environment.

As with any modeling endeavor, building the logic in steps is always best.

Process Flow has its own logic-building interface, analogous to the 3D modeling surface, and its own set of modeling objects, referred to as *activities*. The basic components of the Process Flow modeling environment and their terminology are shown in the figure below.



- The Process Flow View is the workspace where logic is defined. It is accessed via the **Process Flow** button on the **Main Menu** bar or the **Toolbox**. It is displayed as a tabbed window in the modeling environment.
- Logic is defined by selecting and combining activities from a **Library**, similar to the 3D objects in their library. A part of that Library is shown at the left side of the figure above.
- As with the 3D objects, **Properties** are shown in the window to the right in the basic interface.
- In the workspace or View, logic is defined through connected activities, connected either in stacked blocks or directed lines.
- Tokens flow through activities and trigger the logic elements.

There are different types of Process Flows - *General, Object, SubFlow, People* - but only the *General Process Flow* is used in the primer.

Activities are the building blocks of Process Flow - they are logical operations or steps in a logical process. Activities are analogous to 3D objects since they are dragged from a library onto a modeling surface or workspace. Also, like 3D objects, activities have properties that define their behavior. Activities are grouped into the following categories. This primer only addresses a subset of the activities, but the complete list is provided so that the reader is aware of the capabilities of Process Flow.

- **Token Creation** activities are analogous to a Source in 3D. Tokens are created based on inter-arrival times, schedules, and “listening” for specific events to occur in a 3D model. Listening to events in the 3D model is a key capability in Process Flow; for example, when an event occurs in the 3D model, such as an item arrives at an object, it may trigger the creation of a “token” in Process Flow which flows through activities and executes the custom logic defined in the Process Flow View.
- **Basic** activities include assigning label values, implementing a delay in the flow, waiting for another event to occur, making decisions (branching in the logic), destroying tokens (like a Sink in 3D), defining a custom or special activity, etc.
- **Sub Flow** activities are used to create separate flow logic that contains a set of activities that may be used by multiple objects or other activities. It is analogous to a function or subroutine in computer programming.
- **Visual** activities change the appearance of an object in the 3D model or trigger an animation on a 3D object.
- **Object** activities are used to create, move, or destroy objects, such as flow items, fixed resources, Task Executors, etc., in a 3D model.
- **Task Sequences** activities are used to build custom task sequences that are assigned to Task Executors in a 3D model, including travel, load, unload, delay, etc.
- **Shared Assets** activities are used to define, access, and manage the following:
 - *List* activities enable pushing and pulling tokens, flow items, Task Executors, etc. to Lists, which can be local to the Process Flow itself or tied to a Global List in a simulation model.
 - *Resources* represent limited supplies of entities that can be acquired and released. They can simulate a supply of goods, services, time, materials, employees, etc.
 - *Variables* are used to store data in a centralized location that is accessible throughout Process Flow and can be read and changed by tokens.
 - *Zones* have two main uses - to gather statistics for a group of activities and to restrict access to a set of activities based on token properties.
- **Warehousing** activities are used to find a slot or an item in a Rack object.
- **Coordination** activities manage relationships between multiple tokens in a Process Flow, including splitting, joining, and synchronizing tokens.

- **Preemption** activities manage interruptions in a token's flow.
- **Display** activities annotate a Process Flow with text, arrows, images, and containers. They do not affect the logic, just the visual appearance, which can make Process Flows better organized, clearer, and easier to understand.
- **People Activity Sets** are preconfigured activities that are bundled together to model basic People Module tasks, such as *Walk then Process*, *Escort then Process*, *Wait then Process*, etc.
- **People Basic** activities create or delete a Person object or define a People Process.
- **People Resources** are activities that facilitate acquiring and releasing People-related resources, such as Locations, Staff, Transports, and Equipment.
- **People Sub Flows** are activities tied to pre-built Sub-Flow objects, including *Walk*, *Wait in Line*, *Escort Person*, *Transport Person*, *Move Equipment*.

A token is a basic element of Process Flow logic. Analogous to flow items in the 3D environment, tokens flow through activities as a simulation runs. Tokens are typically more abstract than flow items since they usually represent logic flow rather than physical flow. Each token has a unique ID and a set of labels or characteristics. Tokens are depicted as green circles in Process Flow as a simulation runs; they may change color depending on their use and state.

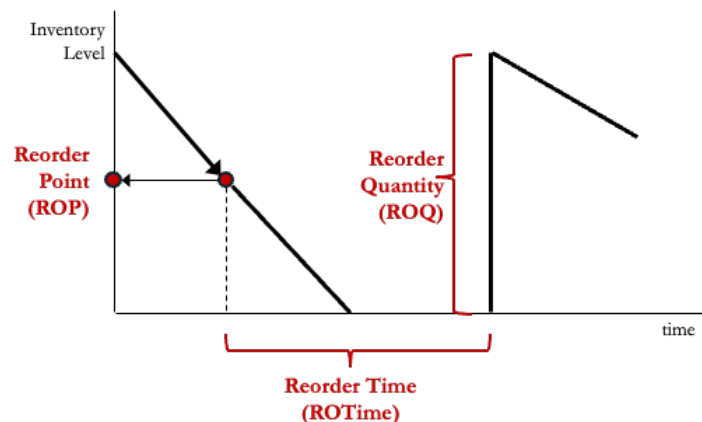
Tokens move from activity to activity either by connectors or through stacked blocks. A block is a set of activities that have been “snapped” together, i.e., activities that form a single sequence of process flow steps.

Tokens have **Labels**, which are an important part of modeling with Process Flow. Labels store information that is used in the logic. As with the 3D objects, each label has a name and value; the value can be of any data type, e.g., numbers, text, arrays, object references, etc. Similar to 3D objects, token labels are referenced in the dot-notation format token.LabelName, e.g. token.Type.

The base model for the additions described in this chapter is **Primer_14** that was saved at the end of Chapter 20. However, a copy of that file was saved as **Primer_15**; thus, we begin with that file.

21.2 Set up to use Process Flow to model inventory policy

Continuing the primer example from the previous chapter, Process Flow is used to model a reordering system for supplying components to the packing area. Rather than having components delivered in batches on a schedule, as in the current model, components are ordered when their inventory level falls below a specified reorder point. As shown in the figure below, when the inventory level falls over time below a specified reorder point (ROP), an order is triggered for a specified reorder quantity (ROQ) of the component, and that quantity is delivered as a batch after a specified time (ROTime).



Therefore, three new parameters must be specified for each component: ROP, ROQ, and ROTime.

- ROQ is analogous to batch size, which already exists in the **Parameters Table** named *ComponentBatchSize*.
- ROP is specified as a quantity, level of inventory, and not time.
- ROTime is the total time from when an order is placed until it is delivered.

Another parameter to be considered in the model is initial inventory, the number of components on hand when a simulation starts. It can be any value greater than or equal to zero. However, if the initial value is below the reorder point, there would need to be logic that would ensure an order is placed when the simulation starts. While this would not be difficult to include, for simplicity, assume the initial inventory value is above the reorder point when a simulation starts. This can be ensured by setting the lower bound of the parameter to the component's reorder point value.

Therefore, three additional **Model Parameters Tables** need to be added to the model – one each for reorder point, reorder time, and initial inventory.

Since the reorder quantity is basically the same as the batch size parameter, which is already included in the model in the **Parameters Table** *ComponentBatchSize*, batch size will represent the reorder quantity.

The parameters in the **Model Parameters Table** *ComponentFrequency* (time between batches) will no longer be used since the reorder point system will replace it. However, leave its **Parameters Table** in the model in case there is a need to return to the scheduled-order system. Also, some components might use the scheduled-order system, and some might use the reorder-point system.

For the reorder point.

- In the **Toolbox**, select **Statistics**, then *Model Parameter Table*.

As shown in the table to the right,

- Name the table *ReOrderPoint*.
- Add two parameters named *ROPComp_A* and *ROPComp_B*.
- For both parameters, set their **Value** for **Type** to *Integer*, **Lower Bound** to 1, and **Upper Bound** to 100.
- Set the current values to 6 for *ROPComp_A* and 20 for *ROPComp_B*.

The screenshot shows the 'ReOrderPoint' parameter table. It has two columns: 'Name' and 'Value'. The 'Name' column contains 'ROPComp_A' and 'ROPComp_B'. The 'Value' column contains '6' and '20'. The 'Display Units' column is empty. The 'Description' column is empty. Below the table, there are settings for 'Type' (Integer), 'Lower Bound' (1), 'Upper Bound' (100), 'Reference' (None), and 'On Set'.

Name	Value	Display Units	Description
ROPComp_A	6		
ROPComp_B	20		

Similarly, for the reorder time.

- In the **Toolbox**, select **Statistics**, then *Model Parameter Table*.

As shown in the table to the right,

- Name the table *ReOrderTime*.
- Add two parameters named *ROTimeComp_A* and *ROTimeComp_B*.
- For both parameters, keep their **Value** for **Type** as *Continuous* and set their **Value** for **Lower Bound** to 30 and **Upper Bound** to 480.
- Set the current values to 60 for both *ROTimeComp_A* and *ROTimeComp_B*.

The screenshot shows the 'ReOrderTime' parameter table. It has two columns: 'Name' and 'Value'. The 'Name' column contains 'ROTimeComp_A' and 'ROTimeComp_B'. The 'Value' column contains '60' and '60'. The 'Display Units' column is empty. The 'Description' column is empty.

Name	Value	Display Units	Description
ROTimeComp_A	60		
ROTimeComp_B	60		

Similarly, for the initial inventory:

- In the **Toolbox**, select **Statistics**, then *Model Parameter Table*.

As shown in the table to the right,

- Name the table *InitInv*.
- Add two parameters named *InitInv_Comp_A* and *InitInv_Comp_B*.
- For both parameters, set their **Value** for **Type** to *Integer* and **Upper Bound** to 100.
- For both parameters, set their **Value** for **Lower Bound** to the value of the component's reorder point. Since the reorder point is defined as a parameter, the **Lower Bound** needs to reference the reorder point, as shown in the figure above. For each parameter's **Value** cell, use the dropdown menu to select **Parameters**, then the table *ReOrderPoint*, then the applicable component name.
- Set the current values to 12 for *InitInv_Comp_A* and 30 for *InitInv_Comp_B*.

The screenshot shows the 'InitInv' parameter table. It has two columns: 'Name' and 'Value'. The 'Name' column contains 'InitInv_Comp_A' and 'InitInv_Comp_B'. The 'Value' column contains '12' and '30'. The 'Display Units' column is empty. The 'Description' column is empty. Below the table, there are settings for 'Type' (Integer), 'Lower Bound' (Model.parameters[ROPComp_A].value), 'Upper Bound' (100), 'Reference' (None), and 'On Set'.

Name	Value	Display Units	Description
InitInv_Comp_A	12		
InitInv_Comp_B	30		



If you haven't already done so, save the model. Recall that it is good practice to save often.

22 MODELING INITIAL INVENTORY

Chapter 22 describes several approaches to modeling the creation of an initial component inventory and explains how to implement the more general approach.

This chapter uses a small and simple example to introduce modeling with Process Flow. That example loads components into their storage areas when a model starts, thus providing an initial inventory of components.

The first section describes a straightforward approach to defining the logic for creating an initial inventory. However, the approach involves additional work when the model is scaled to consider more components. Therefore, the second section illustrates a more general approach to adding initial inventory, which does not require additional work when adding components to the model. While the extension is more complex and takes more time to develop, by using it, the model scales as more components are added, and no changes are needed in the Process Flow logic.

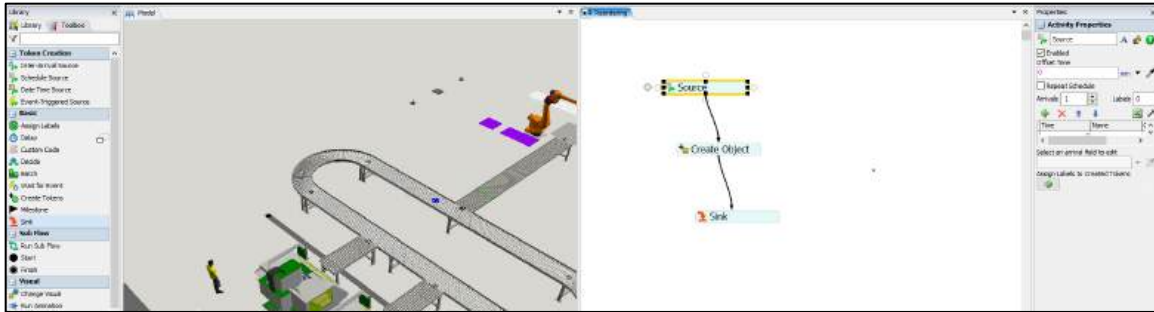
Using these two approaches illustrates that there are always multiple approaches to modeling a system. Two experienced modelers are likely to approach modeling the same system very differently. These differences are typically due to experience, background, modeling philosophy, how the model will be used and by whom, the due date for the completed model, etc.

The base model for the additions described in this chapter is Primer_15 that was saved at the end of Chapter 20. Thus, this chapter continues to add to the model Primer_15.

22.1 A simple approach

This first approach to creating an initial inventory is relatively straightforward and is a good introduction to modeling operations systems in Process Flow.

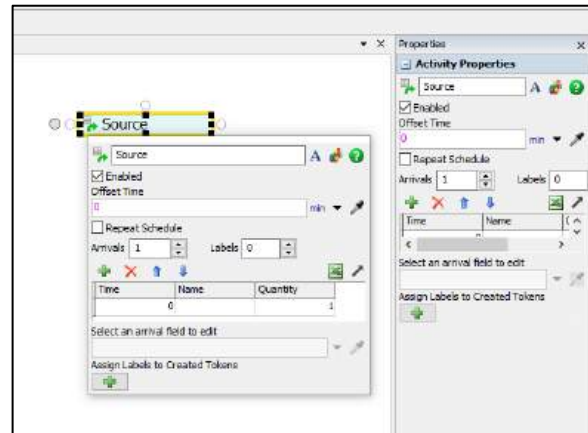
- Through the **Process Flow** button on the **Main Menu** or via the **Toolbox**, select **Add a General Process Flow**, and change its name from the default *ProcessFlow* to *Reordering*.
- As shown in the figure below, drag out three activities onto the Process Flow View or modeling surface:
 - **Schedule Source** (from the **Token Creation** section of the **Activity Library**)
 - **Create Object** (from the **Objects** section)
 - **Sink** (from the **Basic** section)



- As shown in the picture above, link the activities using the **Connector** – move the mouse over an activity until the mouse cursor changes to a chain, then holding down the left mouse button, drag the arrowed-line connector to the other activity. The activities may be positioned by selecting them with the left mouse button and dragging them to the desired position.

As shown in the figure to the right, when selecting an activity, i.e., when an activity is highlighted in yellow, handles are available to size the activity. The handles are the small black squares around the activity box.

Also shown in the figure to the right, the properties of an activity are displayed either by (1) selecting an activity and viewing its properties in the Properties window, (2) clicking on an activity's icon, or (3) double-clicking the activity.



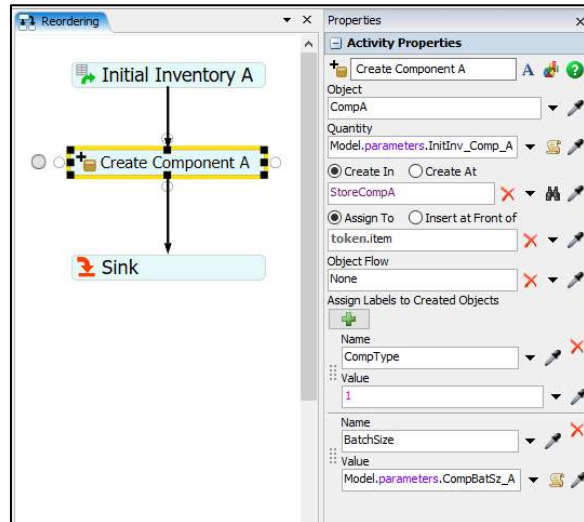
For now, the only changes being made to the Process Flow are to the properties of the **Create Object** activity.

- However, change the name of the
 - **Schedule Source** activity from *Source* to *Initial Inventory A*
 - **Create Object** activity from *Create Object* to *Create Component A*.

The **Create Object** activity creates flow items that represent the components and puts them into their **Queue** in the Packing Area. Note that the flow items are created directly in the **Queue** and do not come from the **Source**. As a result, these items do not pass through the **Queue's OnEntry** trigger. Therefore, no logic on the **OnEntry** trigger are invoked for these items. Of course, that logic could be moved to Process Flow.

Change the **Create Object** activity as described below and as shown in the figure to the right. This activity creates a specified number of components in its store location.

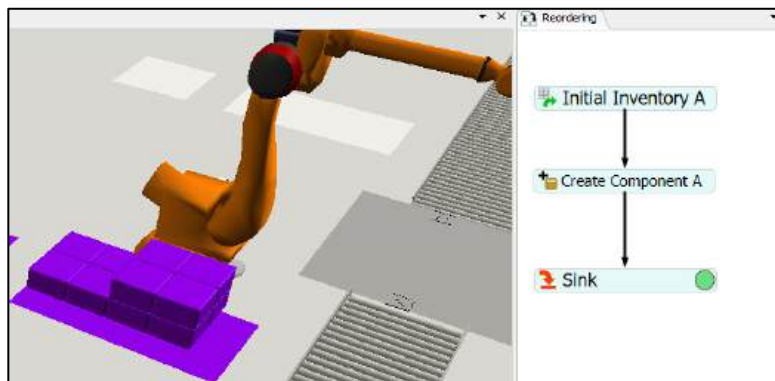
- The value of the **Object** property is obtained by using the drop-down menu, selecting *FlowItems*, and then *CompA*.
- The value of the **Quantity** property is obtained by using the drop-down menu and selecting *Parameter*, then *InitInv*, and finally *InitInv_Comp_A*. This selection generates the command `Model.parameters["InitInv_Comp_A"].value`, which *FlexSim* executes to get the value of the **Model Parameter** *InitInv_Comp_A*.
- The **Create In location** value is obtained by using the sampler (eyedropper icon) to sample the **Queue** *StoreCompA* in the 3D model.



Add two Labels that are passed to the created object; in this case, a flow item representing components. The labels and their values are added by the **Source** activity.

- Below the **Assign Labels to Created Objects** section, press the  button twice to add two labels.
- For the first label, set the name to *CompType* and **Value** to 1. The component type is 1, i.e., Component A.
- For the second label, set the name to *BatchSize* and set its **Value** by using the dropdown menu and selecting *Parameter*, then *ComponentBatchSize*, then *CompBatSz_A*.
- **Reset** the model and use the **Step** button to the right of the **Run** and **Stop** buttons on the **Execution** toolbar. As the name implies, this moves through a model's execution one step at a time. It will likely take several clicks of the **Step** button for the items to appear since *FlexSim* executes multiple events within one time period and at time zero.

At some point, clicking the **Step** button results in a token appearing in the **Initial Inventory** activity. The next click of the **Step** button moves the token to the **Create Components** activity and then to the **Sink**. When the token leaves the **Create Components** activity, the initial inventory of CompAs appears as shown in the figure to the right – there are 12 purple boxes in the **Queue** *StoreCompA* object.

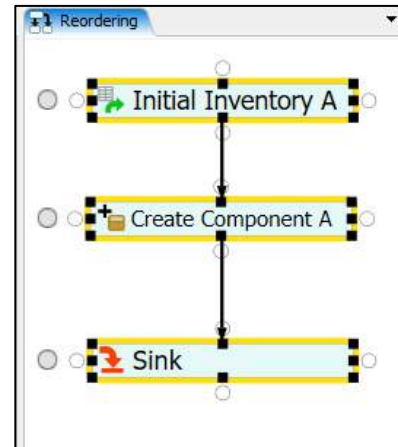


If you haven't already done so, save the model. Recall that it is good practice to save often.

The problem with this approach is that it only creates inventory for Component A since the Create Components activity is “hard-wired” for Component A. The following provides two variations to address this issue.

One approach is to create another set of activities for Component B. The modeling steps defined above could be repeated, or the set of activities for Component A can be copied and pasted, then edited for Component B’s property values. The latter approach is used here.

- Select the activities by holding down the Shift key and “lassoing” the activities, i.e., drag the mouse so that all of the activities are selected, as shown in the figure to the right.
- Use the Cntl-C keys to Copy the activities, then click somewhere to the right of Component A’s activities and use the Cntl-V keys to paste the activities.

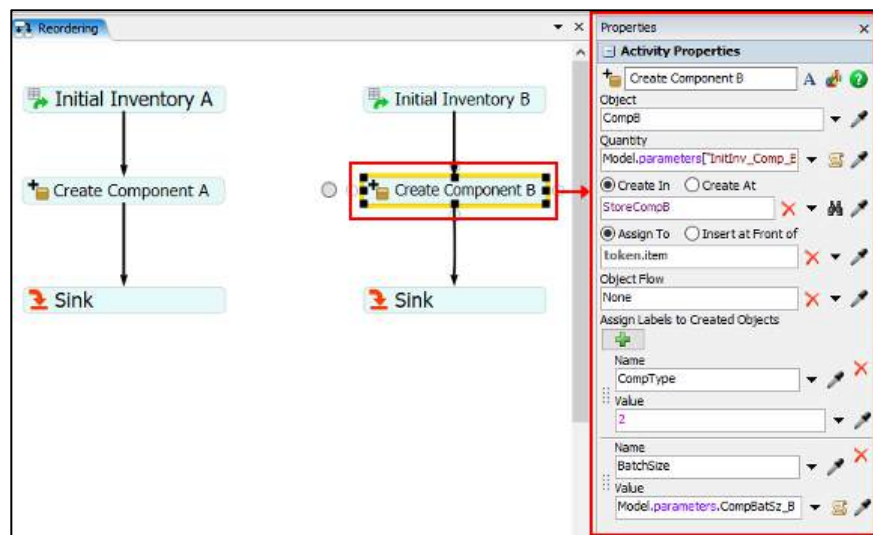


Edit the activity properties as shown in the figure to the right.

- The only change to the **Schedule Source** is to change its name to *Initial Inventory B*.

For the **Create Objects** activity:

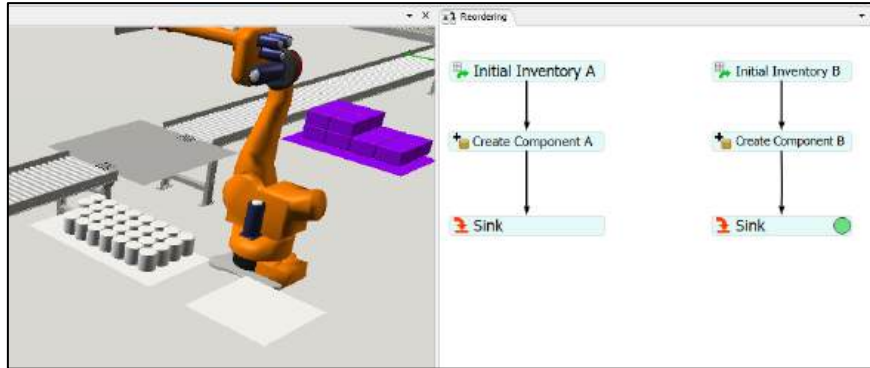
- Rename the activity *Create Component B*.
- For the **Object** property, use the drop-down menu and select *FlowItems*, then *CompB*.
- For the **Quantity** property, use the property’s drop-down menu and select *Parameter*, then *InitInv*, and finally *InitInv_Comp_B*
- The **Create In location** value is obtained by using the sampler (eyedropper icon) to sample the **Queue StoreCompB** in the 3D model.



Edit the two Labels that are passed to the created object.

- For the first label, *CompType*, set its **Value** to 2, since it is a type 2 item or Component B.
- For the second label, *BatchSize*, set its **Value** by using the dropdown menu and selecting *Parameter*, then *ComponentBatchSize_B*. Thus, this contains the batch size for Component B.

- **Reset** the model and use the **Step** button to test that the logic works properly. After several clicks of the **Step** button, the component items should appear in both storage areas, as shown in the figure to the right. Note there are 12 Component As and 30



Component Bs, as defined in the **Model Parameters Table**.

This approach works fine, but the problem is that every time a new component is added to the system, the process above needs to be repeated for each component; i.e., the activities need to be copied and pasted, and their properties edited for the new component values.

The following provides a slight variation on the approach described above.

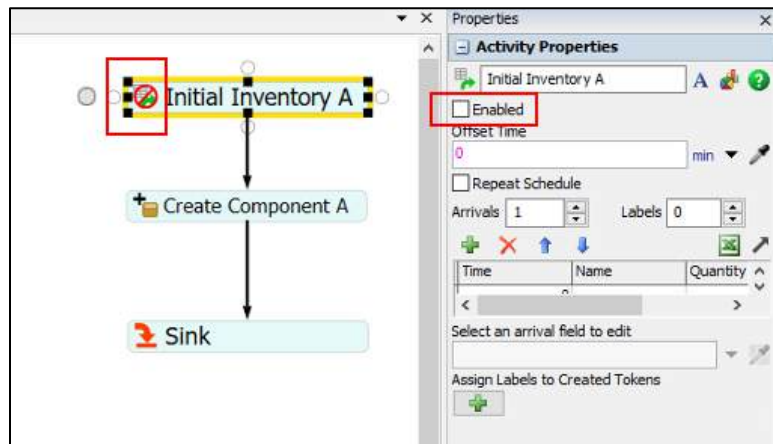
In this approach, all of the component's properties are defined in the **Schedule Source** activity. The property values are defined and stored in token labels, and then the token label values are used to create the components in the single, generic **Create Object** activity.

In this variation, the activities can either be created from scratch or they can start with a copied-and-pasted version of the previous Process Flow. This version can be created in the same Process Flow View, or a new Process Flow instance can be created.

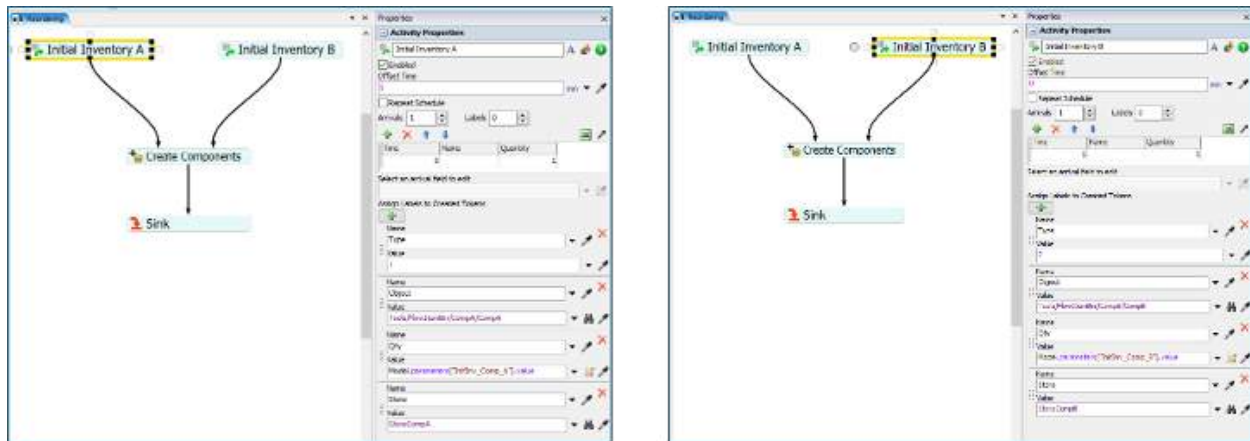
Regardless of where this version is created, if the previous version remains in the model, then the **Schedule Sources** must be made inactive. Otherwise, they will all generate components, and too many will be in the model.

To disable the **Schedule Source**, uncheck the *Enabled* box in the **Activities Properties** window, as shown in the figure to the right.

When the **Source** is disabled, a “no” or “prohibition” symbol (red-crossed circle) appears over the icon on the activity object, as shown in the figure to the right. This prevents the Source from producing tokens.

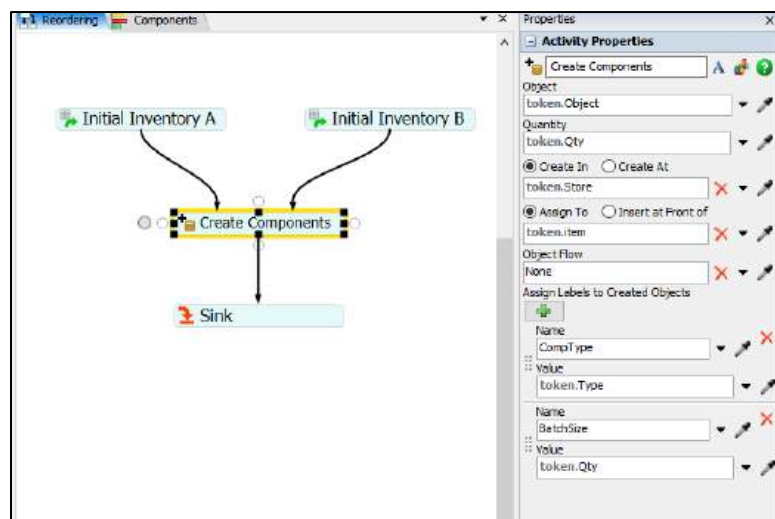


The values for Components A and B in the two **Schedule Source** activities are shown in the figure below.

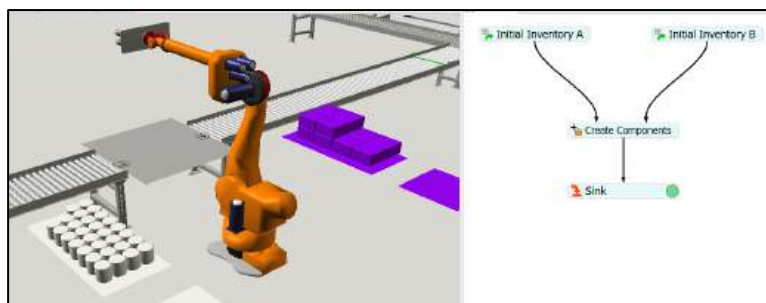


As shown in the figure to the right, the **Create Object** activity uses the token label values to create the components.

As with the previous approach, which worked fine, so does this. However, the problem is that every time a new component is added to the system, the process needs to be repeated for each component; i.e., the activities need to be copied and pasted, and their properties edited for the new component values. This is remedied in the next section.



- **Reset** the model and use the **Step** button to ensure the logic works properly. After several clicks of the **Step** button, the component items should appear in both storage areas, as shown in the figure to the right.



Again, verifying as you move through the modeling process is very important.

22.2 A more general approach

The general approach is based on maintaining all of the data on components in a single **Global Table**, as shown in the figure below. This way, when a new component is added to the model, only a row with the pertinent component information must be added to the table. No changes need to be made to the Process Flow logic and data.

Model ComponentReference						
	Name	CompType	Object	InitInv	Store	BatchSize
Row 1	Comp_A	1	/Tools/FlowItemBin/CompA/CompA	Model.parameters.InitInv_Comp_A	/StoreCompA	Model.parameters.CompBatSz_A
Row 2	Comp_B	2	/Tools/FlowItemBin/CompB/CompB	Model.parameters.InitInv_Comp_B	/StoreCompB	Model.parameters.CompBatSz_B

String
↑

Number
↑

Pointer
↑

FlexScript
↑

Pointer
↑

FlexScript
↑

Data Types

The properties stored in each column in the table are of a different data type, as noted in the figure above.

Create the data table as follows.

- Using the  button in the **Toolbox**, create a **Global Table**.
- Name the table *ComponentReference*.
- Using the table's **Properties** window, increase the number of columns to six.
In terms of rows, for now, leave the table with the default one row. The data type for each column will be changed below, and then, as rows are added, the cells will have the data type of the row above it.
- Name the headers as shown in the figure - *Name*, *CompType*, *Object*, etc.

- Right-click on the cell under each column and select **Assign Data**, then click on the following option depending on the column.
 - For *Name*, select *Assign String Data*. The cell only accepts string data.
 - For *CompType*, select *Assign Number Data*. The cell only accepts numeric values.
 - For *Object*, select *Assign Pointer Data*. The cell only accepts pointer data, i.e., a reference to an object.
 - For *InitInv*, select *Assign FlexScript Data*. The cell only accepts computer code (*FlexScript*).
 - For *Store*, select *Assign Pointer Data*. The cell only accepts pointer data, i.e., a reference to an object.
 - For *BatchSize*, select *Assign FlexScript Data*. The cell only accepts computer code (*FlexScript*).
- Using the table's **Properties** window, increase the number of rows to two.
- Enter the data in the two rows as described below.
 - For *Name*, type in the *Comp_A* and *Comp_B*.
 - For *CompType*, enter 1 and 2.
 - For *Object*,
 - In the **Toolbox**, expand the **FlowItem Bin** section so all flow items are listed in the **Toolbox**.
 - Select the cell in the table (first row of the *Object* column), then use the Sampler (eyedropper) tool to select *Comp_A* from the **Toolbox**'s **FlowItem Bin** list.
 - Repeat for the second row and *Comp_B*.
 - For *InitInv*, enter the command as shown below, which will be executed by *FlexScript* when referenced. It will return the current value of the **Model Parameter**.
 - Type the following in the first row: `Model.parameters.InitInv_Comp_A`
 - Type the following in the second row: `Model.parameters.InitInv_Comp_B`
 - For *Store*,
 - Select the cell in the table (first row of the *Store* column), then use the Sampler (eyedropper) tool to select the **Queue** object *StoreComp_A* in the 3D model.
 - Repeat for the second row and Component B.
 - For *BatchSize*, enter the command shown below, which will be executed by *FlexScript* when referenced. It will return the current value of the **Model Parameter**.
 - Type the following in the first row: `Model.parameters.CompBatSz_A`
 - Type the following in the second row: `Model.parameters.CompBatSz_B`

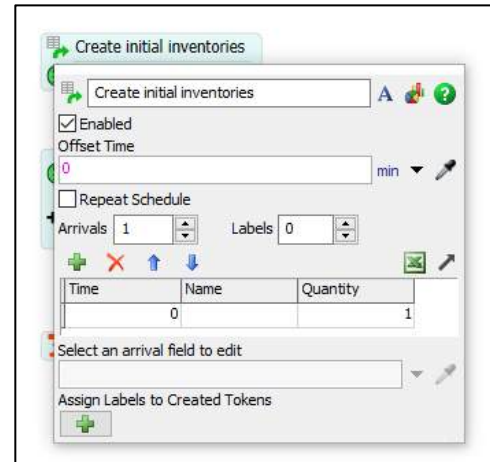
The resulting **Global Table** should look like the one in the figure presented earlier.

The construction of the Process Flow logic is described below. It will use the data from the **Global Table** created above. Basically, the logic will read information from each row in the table, i.e., read information on each component. It will loop through each row in the table until all rows have been processed. Therefore, it doesn't matter how many components are in the table - the logic will process all rows.

For now, just create and populate the activities; connections will be made later in the process.

Use a **Schedule Source** activity and modify its properties as shown in the figure to the right.

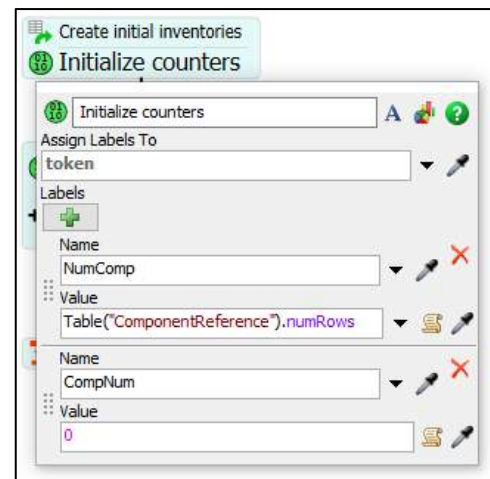
- Only the name needs to be changed.
 - Only one arrival is created at the beginning of the simulation, at Time = 0.



Use an **Assign Labels** activity and modify its properties as described below and shown in the figure to the right.

- Name the activity *Initialize counters*
- Use the  button under **Labels** to create a token label.
 - Name the label *NumComp*.
 - Type in its value as shown in the figure to the right. As the expression is typed, *FlexSim* suggests such things as the table name. Pick the name from the list rather than typing it out – it is both easier and less prone to mistakes.

The method **numRows** determines the number of rows in the specified table, i.e., how many components need to be processed.
- Use the  button under **Labels** to create a second token label.
 - Name the label *CompNum*.
 - Type in its **Value** as 0.



The label **NumComp** contains the total number of components that need initial inventory to be created, the same as the number of rows in the *ComponentReference* **Global Table**. **CompNum** is the cumulative count of how many components have been processed and the number of rows that have been processed in the table. These are used in a later activity to control looping through the table and determine when all components have been processed.

Use another **Assign Labels** activity and modify its properties as described below and shown in the figure to the right.

- Name the activity *Increment counter & Set variables*
- Use the  button under **Labels** to create a token label.
 - Name the label *CompNum*.
 - For the **Value**, select *Increment Label* from the dropdown menu.

When a token passes through this activity, the value of the label *CompNum* will increase by 1. This counts how many components have been processed.

- Use the  button under **Labels** to create a second token label.
 - Name the label *Object*.
 - For the **Value**, select *Table*, then *By Global Table Lookup* from the dropdown menus.

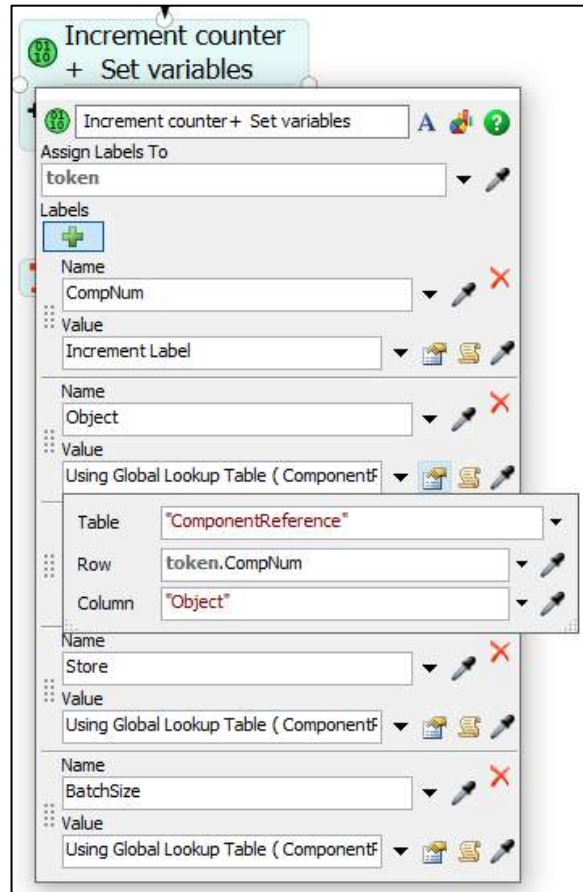
Populate the selection's properties as follows.

 - **Table** – Use the dropdown menu to select *Global Tables*, then *ComponentReference*.
 - **Row** – Type in the value or use the dropdown menu by selecting *Labels*, the *token.CompNum*.

The row in the table is the number of the component being processed.

- **Column** – type in “*Object*.”

Note that the value *3* could also be used in place of “*Object*” since the object data is in the third column of the table. However, if the layout of the table is changed, all references to the columns would need to be changed. Using the header name allows the columns to be rearranged without changing where they are referenced. Of course, if the header name is changed, the references to that column will need to be updated.



The remaining three labels are defined similarly, as described below.

- Use the  button under **Labels** to create a third token label.
 - Name the label *InitInv*.
 - For the **Value**, select *Table*, then *By Global Table Lookup* from the dropdown menus.

Populate the selection's properties as follows.

 - **Table** – Use the dropdown menu to select *Global Tables*, then *ComponentReference*.
 - **Row** – Type in the value or use the dropdown menu by selecting *Labels*, the *token.CompNum*.
 - **Column** – type in “*InitInv*” as the header name in the table.

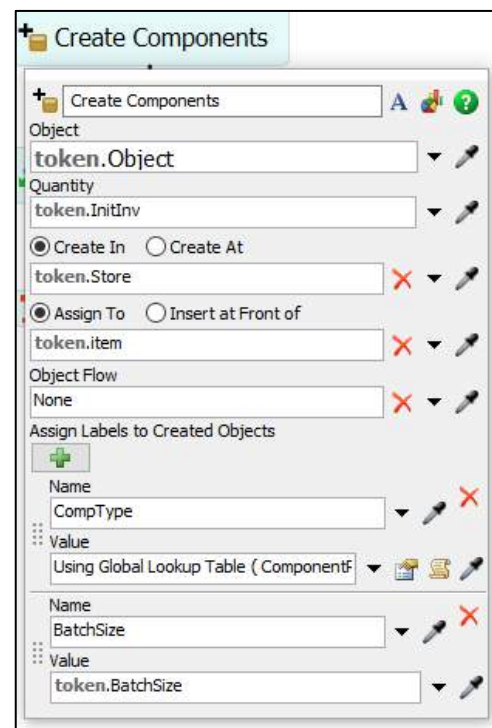
- Use the  button under **Labels** to create a fourth token label.
 - Name the label *Store*.
 - For the **Value**, select *Table*, then *By Global Table Lookup* from the dropdown menus. Populate the selection's properties as follows.
 - **Table** – Use the dropdown menu to select *Global Tables*, then *ComponentReference*.
 - **Row** – Type in the value or use the dropdown menu by selecting *Labels*, the *token.CompNum*.
 - **Column** – type “*Store*” as the header name in the table.
- Use the  button under **Labels** to create a fourth token label.
 - Name the label *BatchSize*.
 - For the **Value**, select *Table*, then *By Global Table Lookup* from the dropdown menus. Populate the selection's properties as follows.
 - **Table** – Use the dropdown menu to select *Global Tables*, then *ComponentReference*.
 - **Row** – Type in the value or use the dropdown menu by selecting *Labels*, the *token.CompNum*.
 - **Column** – type “*BatchSize*” as the header name in the table.

Use a **Create Object** activity and modify its properties as described below and shown in the figure to the right.

- Name the activity *Create Components*.
- For the **Object** property, type in the value *token.Object* or use the dropdown menu and select *Token Label*, then *Object*.
- For the **Quantity** property, type in the value *token.InitInv* or use the dropdown menu and select *Token Label*, then *InitInv*.
- For the **Create In** property, type in the value *token.Store* or use the dropdown menu and select *Token Label*, then *Store*.

Two labels are added to the created Component item.

- Use the  button under **Labels** to create an item label.
 - Name the label *CompType*.
 - For the **Value**, select *Table*, then *By Global Table Lookup* from the dropdown menus. Populate the selection's properties as follows.
 - **Table** – Use the dropdown menu to select *Global Tables*, then *ComponentReference*.
 - **Row** – Type in the value or use the dropdown menu by selecting *Labels*, the *token.CompNum*.
 - **Column** – type “*CompType*” as the header name in the table.
- Use the  button under **Labels** to create an item label.
 - Name the label *BatchSize*.
 - For the **Value**, type in the *token.BatchSize* or use the dropdown menu and select *Token Label*, then *BatchSize*.

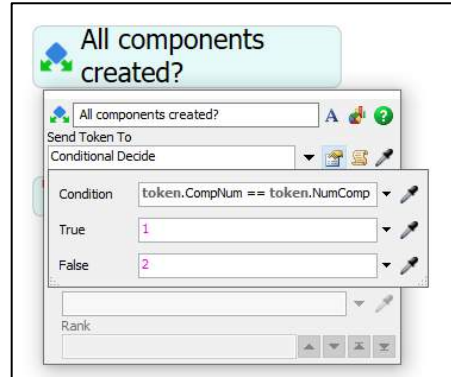


Once the initial number of items has been added to its storage location in the model, a check must be made to see if this is the last component in the table. If so, the Process Flow logic ends; if not, the items for the next component in the table are created. Therefore, a routing decision needs to be made, which is implemented using a **Decide** activity.

- Use a **Decide** activity and modify its properties as described below and shown in the figure to the right.
 - Name the activity *All components created?*
 - For the **Send Token To** property, select *Conditional Decide* from the value's dropdown menu.
 - As shown in the figure to the right, change the Condition to **token.CompNum == token.NumComp**

Note the use of the double=. This is a Comparison Operator; i.e., it compares what is on either side of the symbol. A single = is an Assignment Operator that assigns the value on the right side of the = to the variable on the left side of the =.

The **True 1** and **False 2** properties refer to the number of the activity's connectors where the token will be routed if the comparison made in the Condition is True or False. Since the activities have not yet been connected, this will be checked later.



- Use a **Sink** activity to end the token flow. When a token enters this activity, it is destroyed, much like a **Sink** object in 3D.

When a token enters this activity, it is destroyed, much like a **Sink** object in 3D.

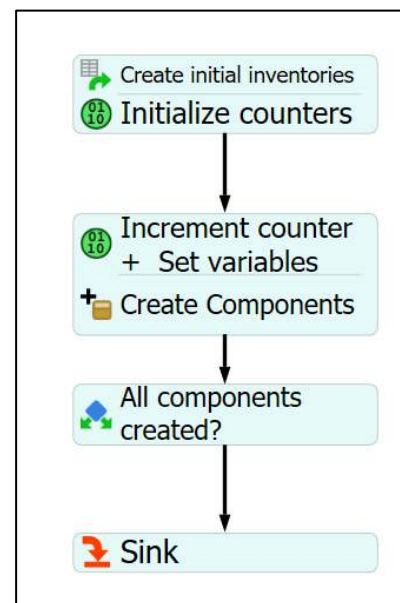
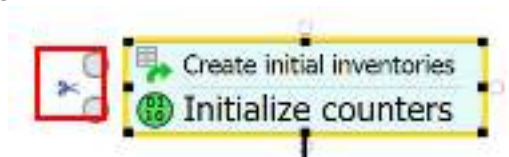
- Arrange and connect the activities as shown in the figure to the right.

This demonstrates the two ways to connect activities.

The first two activities are “snapped together” to form a block of activities. When two activities are brought together, they are automatically joined.

The second and third activities are joined by a connector – the directed line segment.

To separate the activities, click on the block of activities, then click on the scissors icon, which separates the activities on either side of the scissors. This is shown in the figure below; the scissors button is highlighted in the red box.

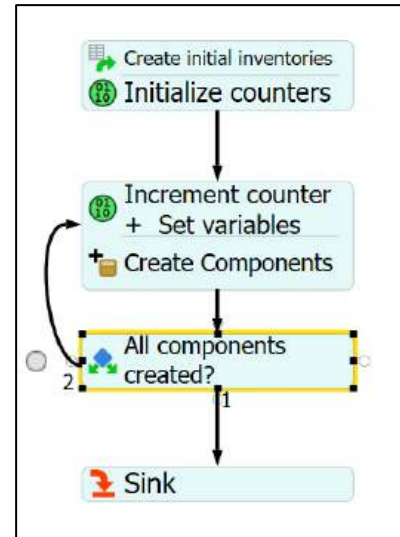


- Add the loop-back connector, as shown in the figure to the right.

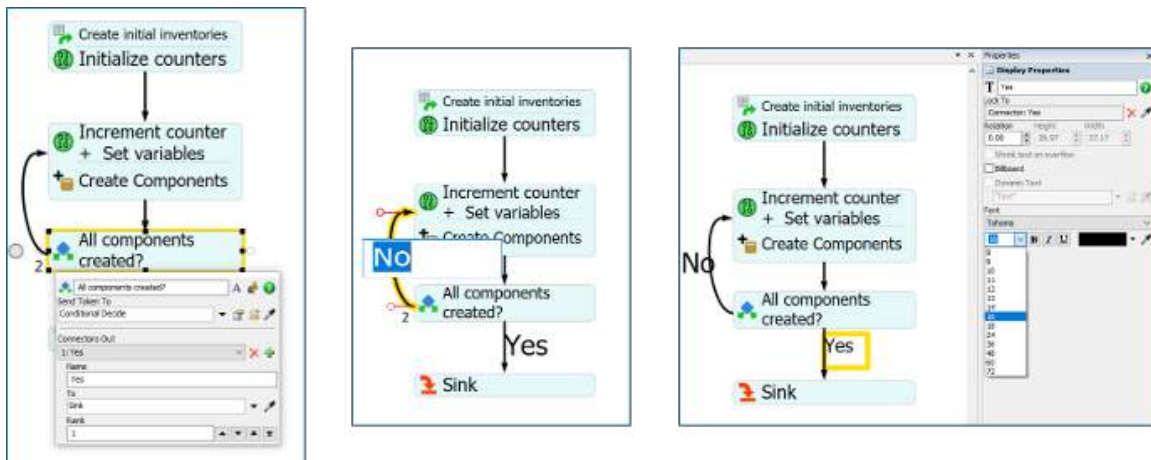
When an activity is selected, move the cursor near the edge of the activity box where the connection will be made. The cursor should change from a pointer to a chain, which signifies a connection can be started at this point.

Move the cursor to the other activity to be connected and release; the two activities should be connected. If not, try again.

Note the 1 and 2 beside the two connectors from the **Decide** activity. These are the connector numbers. Recall from a previous step that if the condition evaluated in the **Decide** activity is *True*, then the token uses Connector 1 to travel to the next activity. If the condition is considered *False*, then the token uses Connector 2.



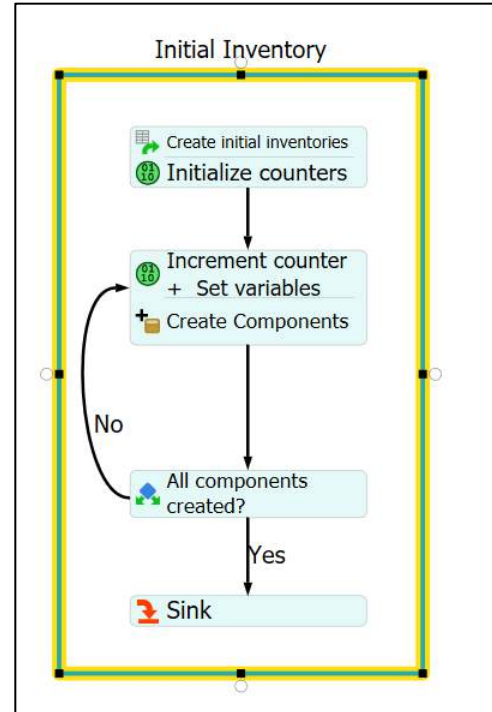
- Clicking on the **Decide** activity icon brings up the interface shown in the figure to the left below. The connector can be named here, e.g., “Yes.” The interface also shows the connection number and the object at which the connection ends.



- Double-clicking on the connector brings up a text box, which provides another means for a connector to be named. This is shown in the middle figure above.
- The middle figure also shows the handles that are available to shape the connector.
- As shown in the figure to the right above, selecting the connector label brings up some properties that allow the text to be edited and its font, color, and size changed. In this case, the font size is reduced from the default of 24 to 16.

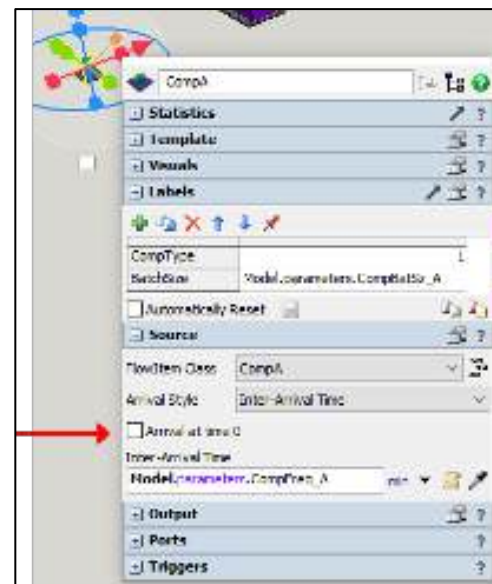
Finally, put this set of logic into a visual container, as shown in the figure to the right. This will help organize the logic segments as more are added and provide an easy means to move all activities; i.e., all activities in a container are moved together.

- In the **Display** section of the Process Flow library, select *Container*, then *Process*, and click on the workspace.
- Select and drag the container so that it encompasses the logic as shown in the figure to the right. Use the handles (small black squares on the periphery of the container) to position the object.
- Change the container's text box from *Process* to *Initial Inventory*. The textbox is a separate entity and can be moved anywhere or removed completely by just selecting it and pressing the Delete key.

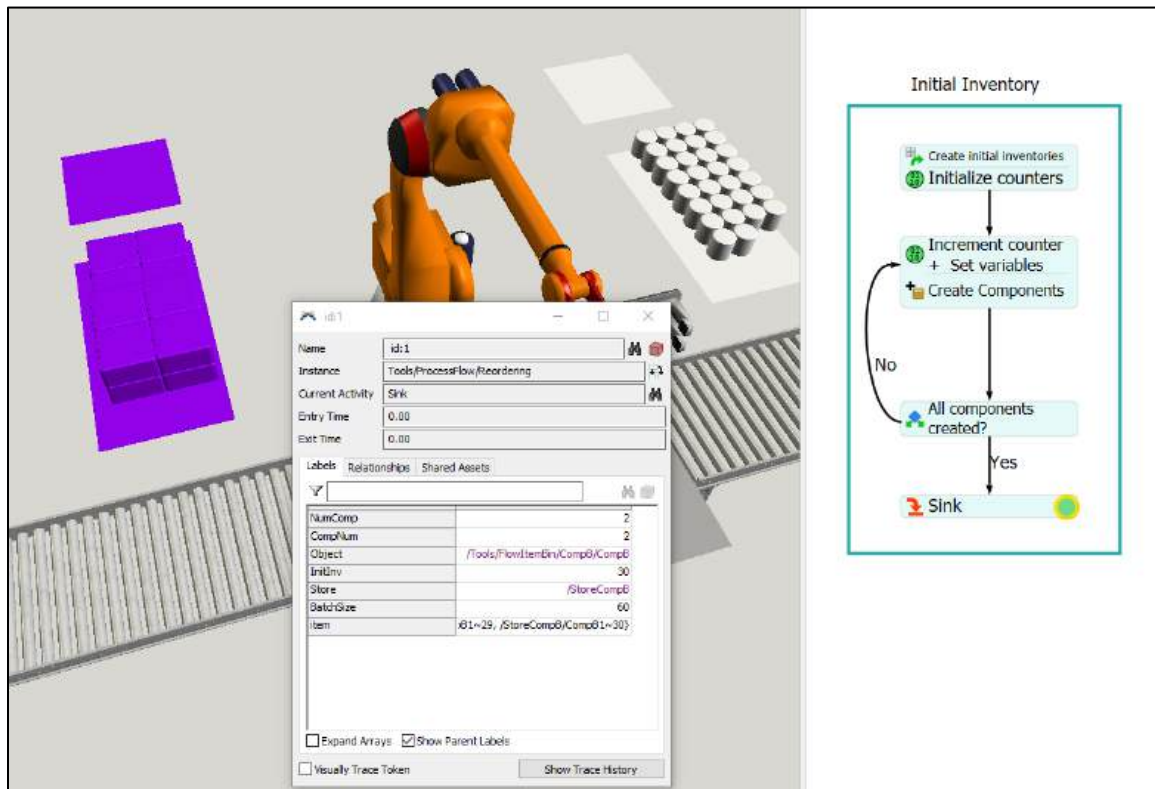


As the model stands now, two batches of each component will be delivered at the beginning of a simulation – one from the 3D Sources (CompA and CompB) and one from the initial inventory Process Flow. To rectify this:

- Uncheck the **Arrival at time 0** box on the **Source** pane of each component's **Source** object. The red arrow highlights the check box in the figure to the right.



- **Reset** and **Step** through the events, noting the token moving through the Process Flow and the initial inventory for each component created when the token hits the **Sink** activity, as shown in the figure below. In this case, the second component's initial inventory was just created.



Note the window in the center of the figure above; it shows the token's label values as it moves through the Process Flow. This information is available anytime for any token; to view it, double-click the token.

Also, note that it is assumed the Finishing Operator does not unpack the initial inventory. No logic is included in the Process Flow to do that. However, the initial inventory is likely a partial batch, which the operator unloaded before the simulation started.



If you haven't already done so, save the model. Recall that it is good practice to save often.



Use the **Save Model As** option in the **File** menu to make a copy of the existing model to be further customized in the next section. Again, you can use any file name, but the following model is referred to as Primer_16 in the primer.

23 MODELING INVENTORY REORDER POLICY

Chapter 23 uses Process Flow to incorporate a reorder point inventory system to manage components.

This section defines the reorder-point inventory system logic and describes in detail how to implement it using Process Flow. However, before doing so, a few preparatory changes are made in the 3D model. Why these changes are being made is explained as they are used in the next section.

The base model for the additions described in this chapter is **Primer_15** that was saved at the end of Chapter 22. However, a copy of that file was saved as **Primer_16**; thus, we begin with that file.

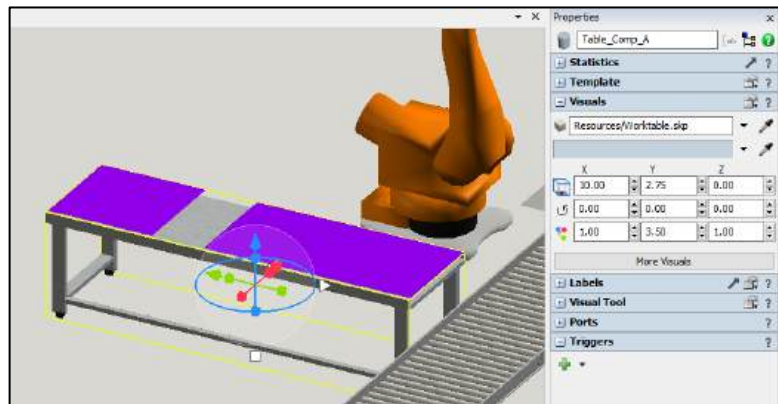
23.1 Changes in 3D objects for use in Process Flow

Before extending the Process Flow logic, this section describes some changes that prepare the model for adding Process Flow logic.

23.1.1 Add storage tables

This change is for aesthetics, as a base for the component storage areas, and will be a place to unpack batches of components. A table object is placed under the objects in the component storage area, as shown in the figure to the right.

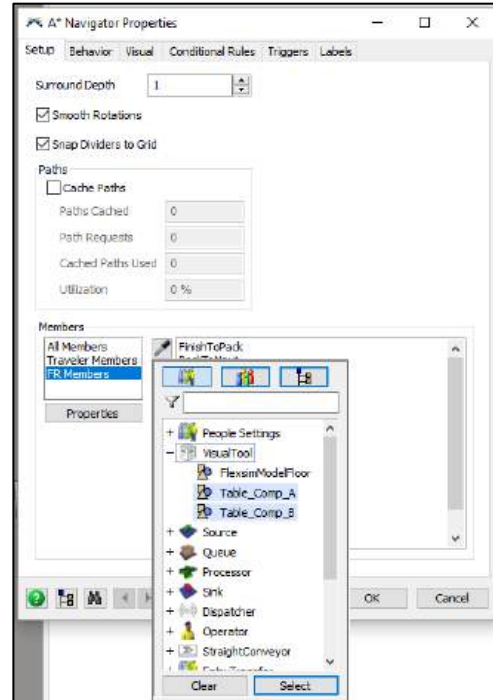
- Drag out a **Shape** object from the **Visual** pane of the **Library**; its default shape is a gray cylinder.
- Change its name from *Shape1* to *Table_CompA*.
- In the **Visuals** pane, use the dropdown menu and select *Browse...*. Browse your computer to find the supplied **Resources** folder and select the *Worktable.skp* file.
- Using the settings in the **Visuals** pane in the figure above, size and position the table under the unpacking (**Separator**) and storage (**Queue**) objects.
- Repeat for the storage area for Component B by copying and pasting the current object. Then, rename it *Table_CompB*, and position it under the unpacking and storage objects.



Since component storage tables are barriers the operator needs to avoid, they are added to the A* Navigator, as shown in the figure to the right.

- Open the **A* Navigator** tool from the **Toolbox**.
- In the **Members** section of the **Setup** tab, select *FR Members*.
- Use the  button to select *Table_Comp_A* and *Table_Comp_B* from the **Visual Tool** section, as shown in the figure to the right. Or, use the sampler tool (eyedropper) to choose the table objects in the 3D view.

Note this step would not be necessary if the added objects were connected to an object that had already been added to the **A* Navigator**. The table is just a visual object and is not connected with an A or S Connection to any other object; i.e., it has no Ports.



23.1.2 Reset storage colors

Each component's storage will be used as a visual indicator, i.e., the storage locations will be colored black when their reorder point is reached and remain black until the ordered batch is unpacked. Once unpacked, the location's color will be changed back to its basic color (Component A is purple, and Component B is white). The change-color logic is added later, but an **On Reset** trigger needs to be added to the object to set its correct color at Reset in case it is in the reorder state when a simulation ends.

- For the **Queue** *StoreCompA*, use the  button in the **Triggers** section to add an **On Reset** trigger.
- Use the  button to the right of the **On Reset** text box to select *Set Object Color*.
- On the **Set Object Color** interface, change the **Object** value to *current*, and for the **Color** value, select *Color.purple*.

Repeat the above for the second storage area.

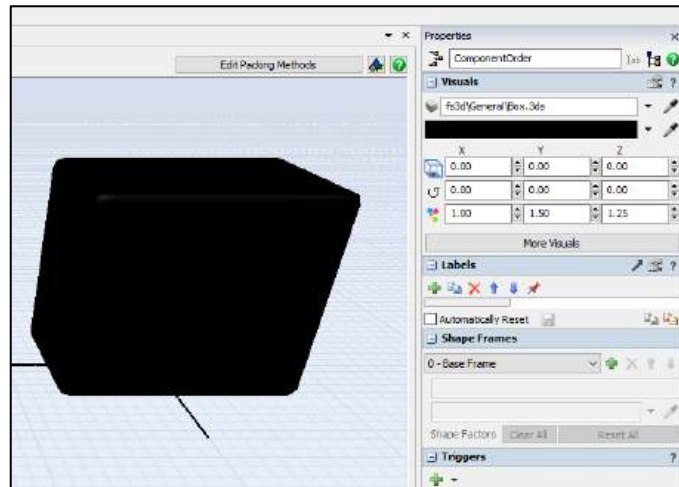
- For the **Queue** *StoreCompB*, use the  button in the **Triggers** section to add an **On Reset** trigger.
- Use the  button to the right of the **On Reset** text box to select *Set Object Color*.
- On the **Set Object Color** interface, change the **Object** value to *current*, and for the **Color** value, select *Color.white*.

23.1.3 Create a component order item

Components arrive at the facility in batches or as an order. The order item is transient in that it is only in the model long enough to be unpacked by the Finishing Operator. Therefore, any shape would be fine, such as a basic box. While a box is used, it will be its own type of item and resized.

As shown in the figure to the right and described below, create a new item in the **Flowitem Bin**.

- Select the **Box** item at the top of the list of flow items and press the **Duplicate** button just to the right of the  button, which makes a copy of the box object and adds it as the last item in the list of flow items.
- In the **Properties** window, change the name to *ComponentOrder*.
- Set the size and location values as shown in the figure above and select its color as black.



23.1.4 Remove objects for scheduled orders

As developed earlier, the model contains logic for generating batches of components on a schedule and unpacking them. However, that logic within the 3D objects will be replaced by reorder-point logic that will be developed in Process Flow.

The new process will deliver batches directly to the component's storage table and not a common batch queue. This change in delivery adds little work to the delivery resource but saves time on the more critical Finishing Operator resource.

Components currently arrive in scheduled batches. As a result, the following objects will no longer be needed:

- **Source** for each component that generates batches on a schedule.
- Common batch **Queue** that holds any arriving batch and calls a Finishing Operator to move the batch to the component's storage area and unpack components.
- **Separators** for each component that converts a batch to a quantity of components.

Therefore, these objects can be removed from the model, or they could be left in with the **Sources** disconnected from the **Queue BatchQueue**. In this case, the objects will be removed. Unnecessary objects only complicate the model and make it harder to maintain.

- Delete the five objects described above (two **Sources**, one **Queue**, and two **Separators**) by selecting them and pressing the Delete key.

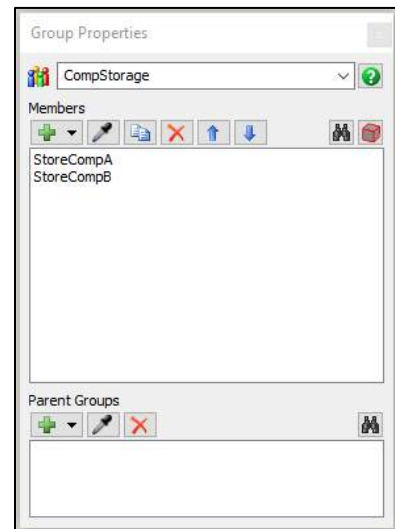
If you'd like to keep the approach to modeling batches, save the current model under a more descriptive name and remove all objects except those relative to the batching logic. The small “study” model can be filed with other small models that illustrate specific concepts. Over time, these become a valuable resource for future modeling.

23.1.5 Create a storage Group

In the Process Flow logic that will be developed soon, any use of a component in the Packing Area will trigger the reorder point logic. Therefore, all of the component storages will be monitored as a group. A group is considered to be any set of objects that are similar or related. In this case, the component storage areas are similar.

As shown in the figure to the right, the **Group** tool allows similar objects to be processed in a like manner. Create the **Group** either via the **Toolbox** or by right-clicking on an object and adding it to a current group or adding it to a new group.

- In this case, use the **Toolbox** to create a **Group**.
- Change its name from *Group1* to *CompStorage*.
- Add the two component-storage **Queues**, *StoreCompA* and *StoreCompB*, by using either the Sampler tool or by using the  button and then the *Select Objects* menu.



23.1.6 Add and update component properties in tables

The Process Flow logic will need a reference to the objects where the batches will be unpacked. Also, as a visual indicator, the storage locations will be colored black when their reorder point is reached and remain black until the ordered batch is unpacked. Once unpacked, the location's color will be changed back to its basic color (Component A is purple, and Component B is white). This additional information is stored in the **Global Table** *ComponentReference*, as shown in the table below.

Model		ComponentReference						
	Name	CompType	Object	InitInv	Store	BatchSize	UnpackLoc	Color
Row 1	Comp_A	1	/Tools/FlowItemBin/CompA/CompA	Model.parameters.InitInv_Comp_A	/StoreCompA	Model.parameters.CompBatSz_A	/Table_Comp_A	Color.purple
Row 2	Comp_B	2	/Tools/FlowItemBin/CompB/CompB	Model.parameters.InitInv_Comp_E	/StoreCompB	Model.parameters.CompBatSz_B	/Table_Comp_B	Color.white

For the unpacking location information:

- Add a column to the **Global Table** *ComponentReference* by increasing the **Columns** value from 6 to 7 in the **Properties** window.
- Name the column header *UnpackLoc*.
- Select and right-click the cells in the *UnpackLoc* column, then select **Assign Data** and then *Assign Pointer Data*. What is stored here is a pointer to the selected object.

- Select the cell in the first row, then using the Sampler tool (eyedropper), click on the *Table_Comp_A* object in the 3D view.
- Select the cell in the second row, then using the Sampler tool (eyedropper), click on the *Table_Comp_B* object in the 3D view.

For the component's color information:

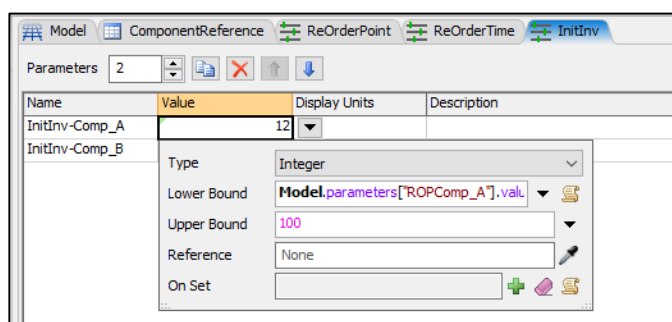
- Add a column in the **Global Table** *ComponentReference* by increasing the **Columns** value from 7 to 8 in the **Properties** window.
- Name the column header *Color*.
- Select and right-click the cells in the *Color* column, then select **Assign Data**, and then *Assign FlexScript Data*.
- Select the cell in the first row, then type Color.purple.
- Select the cell in the first row, then type Color.white.

For reorder point and reorder time information:

The Process Flow logic will also need the value of each component's reorder point and reorder time. These are specified in **Model Parameters Tables**, but for ease of reference, they will also be included in the **Global Table** *ComponentReference*. The values will still be entered in the **Model Parameters Table**, but their values are now copied into the **Global Table** to facilitate their use in Process Flow.

Since the reorder point parameters are used as a lower bound for initial inventory, their names need to be updated in that table, as shown in the figure to the right.

- Edit the parameter's name in the **Lower Bound** expression for **both** parameters in the *InitInv* **Model Parameters Table**.



Add the reorder point and time properties to the **Global Table** *ComponentReference* as shown in the table below.

	Name	CompType	Object	Color	ROP	ROTime
Row 1	Comp_A	1	/Tools/FlowItemBin/Co	Comp_A Color.purple	Model.parameters.ROPComp_A	Model.parameters.ROTimeComp_A
Row 2	Comp_B	2	/Tools/FlowItemBin/Co	Comp_B Color.white	Model.parameters.ROPComp_B	Model.parameters.ROTimeComp_B

For the reorder times, as shown above:

- Add a column in the **Global Table** *ComponentReference* by increasing the **Columns** value from 8 to 9 in the **Properties** window.
- Name the column header *ROP*.
- Select and right-click the cells in the *ROP* column, then select **Assign Data**, and then *Assign FlexScript Data*.
- In the first row in the new column, enter **Model.parameters.ROPComp_A**
- In the first row in the new column, enter **Model.parameters.ROPComp_B**

For reorder time, as shown above:

- Add a column in the **Global Table** *ComponentReference* by increasing the **Columns** value from 9 to 10 in the **Properties** window.
- Name the column header *ROTime*.
- Select and right-click the cells in the *ROTime* column, then select **Assign Data**, and then *Assign FlexScript Data*.
- In the first row in the new column, enter `Model.parameters.ROTimeComp_A`
- In the first row in the new column, enter `Model.parameters.ROTimeComp_B`



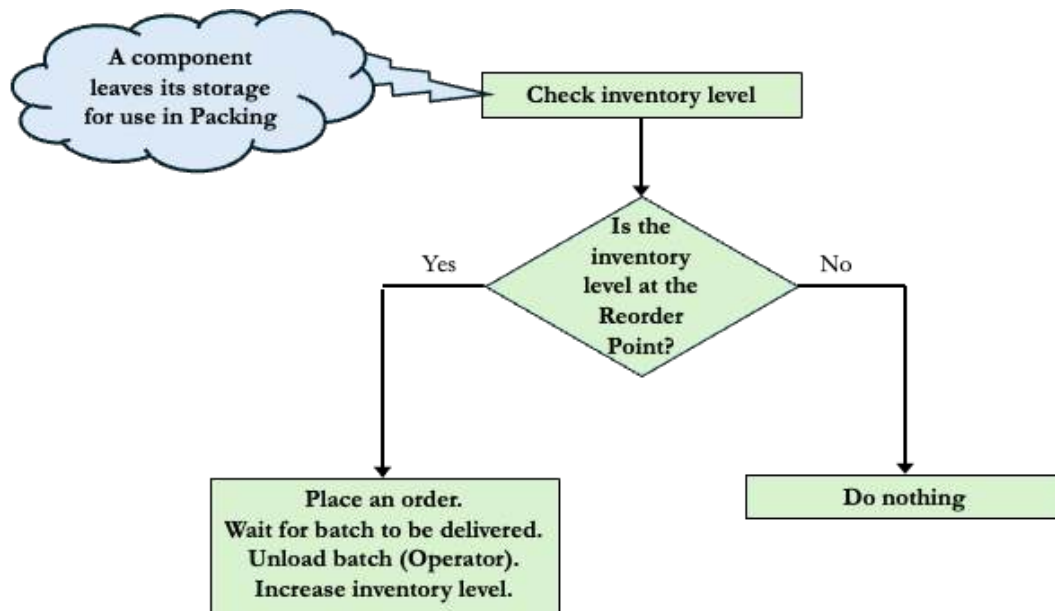
If you haven't already done so, save the model. Recall that it is good practice to save often.

Note:

Model Primer_16 is an incomplete model because the logic that replenishes components has been removed. Therefore, the only components available to the Packing Area are those in the initial inventory. The component replenishment logic is added back in the next section. However, this model is a good illustration of what happens when a system backs up – the packing spur conveyor gets full, the loop conveyor gets full, the Finishing Machines get blocked, and incoming containers are redirected since there is no space for incoming containers in the container storage area.

23.2 Implementing reorder-point inventory logic using Process Flow

The basic logic for a reorder-point inventory system is shown in the figure below. The logic is contained in three sections, as denoted by the boxes in the figure below. The logic is triggered each time a component is used in the Packing Area to fill a container. At that point, the component's inventory level is checked. No action is taken if the inventory level is not at the reorder point. However, if the inventory level equals the reorder point, the reorder quantity is ordered. There is a delay until the order is fulfilled; then, the inventory level is increased by the number of items in the reorder quantity. Orders arrive in batches, and the Finishing Operator unpacks the orders; i.e., a batch is converted into individual components.



This logic is now implemented in *FlexSim* via Process Flow in three sections:

1. Check Inventory Level. The logic is initiated whenever a component leaves its store, i.e., the **Source** activity “listens” to the storage areas and generates a token whenever an item leaves a storage queue. Subsequently, the logic determines which component type the item is and gets its reorder point value and the current contents of its storage area. It then checks to see if the current content of the storage area is equal to the type’s reorder point.
2. Do Not Reorder. If the current content of the storage area is not equal to the type’s reorder point, nothing is done, and the token is destroyed.
3. Reorder. If the current content of the storage area is equal to the type’s reorder point, then the following actions are taken.
 - Set the color of the storage area to black, indicating the component is being reordered.
 - Delay the token for the reorder time.
 - After the delay:
 - Create a batch item at the component’s storage table.
 - Call a Finishing Operator to come to the storage area to unpack the batch.
 - The Finishing Operator unpacks the batch, and the component’s storage area is increased.
 - Once all items are unpacked, the batch item is destroyed, the storage area’s color is changed from black to its color, and the Finishing Operator is released to do other work.

The base model for the additions described in the remainder of this chapter is **Primer_16** that was saved at the end of Section 23.1. However, a copy of that file was saved as **Primer_17**; thus, we begin with that file.

23.2.1 “Check inventory level” logic

The first set of logic involves three activities: an **Event-Triggered Source** activity in the **Token Creation** section of the **Library**, an **Assign Labels** activity in the **Basic** section, and a **Decide** activity in the **Basic** section.

- Drag out these three activities onto the Process Flow workspace named Reordering.

Event-triggered Source

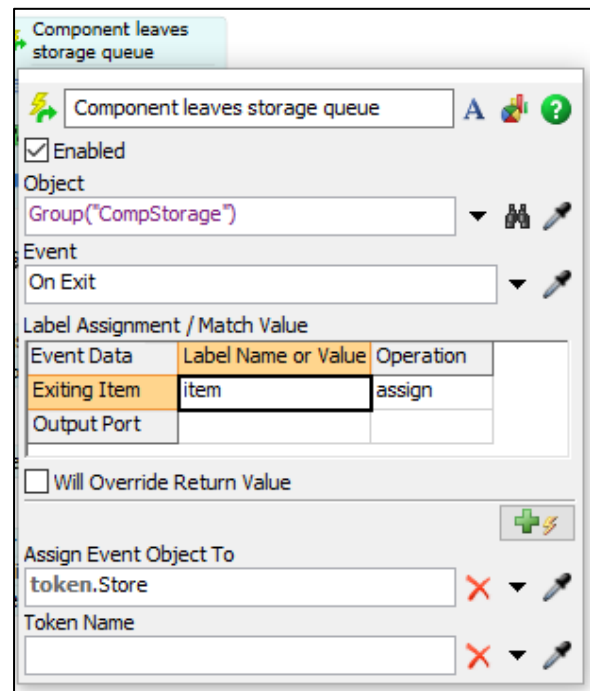
This activity is used to “listen” in the 3D model for **On Exit** events that occur in any of the **Queues** included in the **Group** named *ComponentStorage*. It also collects information on the leaving component (flow item) and the storage area (**Queue** object) from which it is leaving.

This is implemented as described below and as shown in the figure to the right.

- Name the activity *Component leaves storage queue*.
- The **Object** property is set by selecting **Group** from the drop-down menu and then *CompStorage*. The Object here is the Queue that the component just left.
FlexSim writes the command `Group("CompStorage")` that obtains the property's value.
- The **Event** property is *OnExit*, which is selected from its drop-down menu.
- In the **Label Assignment / Match Value** section
 - In the cell where the row is **Exiting Item** and the column is **Label Name or Value**, type *item*.
 - In that same row in the **Operation** column, select *assign* from the dropdown menu.

This assigns a reference to the component (flow item) that is leaving its storage area in the token's label named *Component*.

- In the **Assign Event Object To** textbox, type *token.Store*.
This assigns a reference to the storage location from where a component leaves to the token's label named *Stoe*.

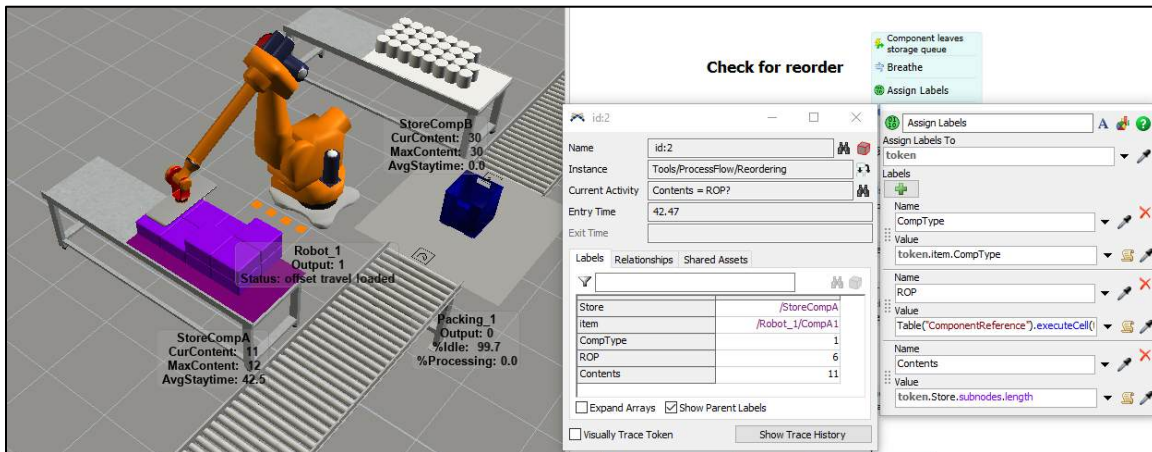


Assign Labels

The next activity, Assign Labels, is used to obtain information from the 3D model that is needed for the reorder logic. Information is obtained about the item (component) exiting storage and about the storage object (Queue) where the item exited. The information is stored as labels on the token that was just created in the activity above (the **Event-Triggered Source**). Any reference to the component is via the token label **token.item**; similarly, reference to the store where the component just exited is through **token.Store**.

The first part of the Process Flow logic tests if its inventory level is at the reorder point after the component exits the storage. If it is not, which happens most of the time, nothing else is done, and no additional information is needed. Therefore, only the information needed to conduct the test is obtained now. If the inventory level is at the reorder point, then additional information will be gathered. Of course, there is nothing wrong with collecting all the information needed in the Process Flow at this point.

The figure below shows the properties of the **Assign Labels** activity on the right and its test and verification to the left. The figure on the left shows the first container being packed after a resupply. The robot has selected a Component A for packing. The statistics for the current content at each storage show 11 for Component A (the resupply value minus the one selected by the robot) and 30 for Component B, its resupply value. The interface in the middle of the figure shows the data on the first token; its label value for **Contents** is 11.

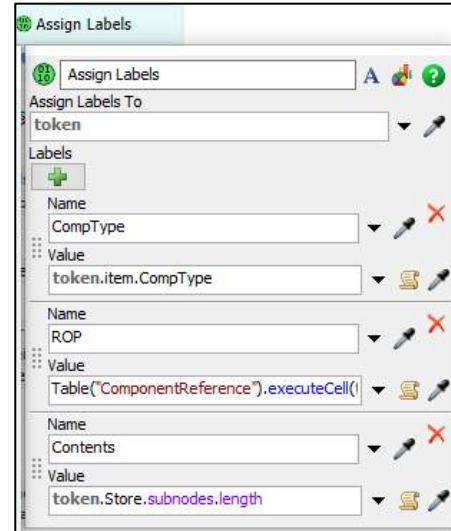


Note the Process Flow activities are linked as a “block.” Recall that activities can either be connected by a directed-line connector or by “snapping” the activities together to form a block. The choice between using the different link methods is made chiefly for readability and clarity.

- Add the **Breathe** activity between the **Source** and **Assign Labels** activities, as shown in the figure above. The **Breathe** activity delays the token for 0.0 time units to ensure the storage’s content statistic has been updated before using it in the next activity. The **Breathe** activity is often used to ensure things are happening when needed. In this case, it was noticed that the content statistic had not been updated with the on-exit event before it was needed in a label definition. At this time, no more needs to be said about this activity.

The following describes defining the labels on the *Assign Labels* activity. Each label is added by pressing the  button. All of the labels are shown in the figure to the right. Names are typed in, and values can be typed in directly, or they can be entered, at least partially, by using the dropdown menu.

- *CompType*'s **Value** is obtained from the expression `token.item.CompType` which first references the item (component) leaving storage (`token.item`) and then the value is stored in the item's label named *CompType*.



- *ROP*'s **Value** is obtained from the expression `Table("ComponentReference").executeCell(token.CompType, "ROP")`

It executes the code statement in the **Global Table** *ComponentReference*'s cell. The cell is defined by the row, which is the value of the token label *CompType*, and the column named "ROP" in the table.

- *Contents*'s **Value** is obtained from the expression `token.Store.subnodes.length` which provides the value of the current contents of the object (`subnodes.length`) that is stored in the token label name *Store*.

Decide

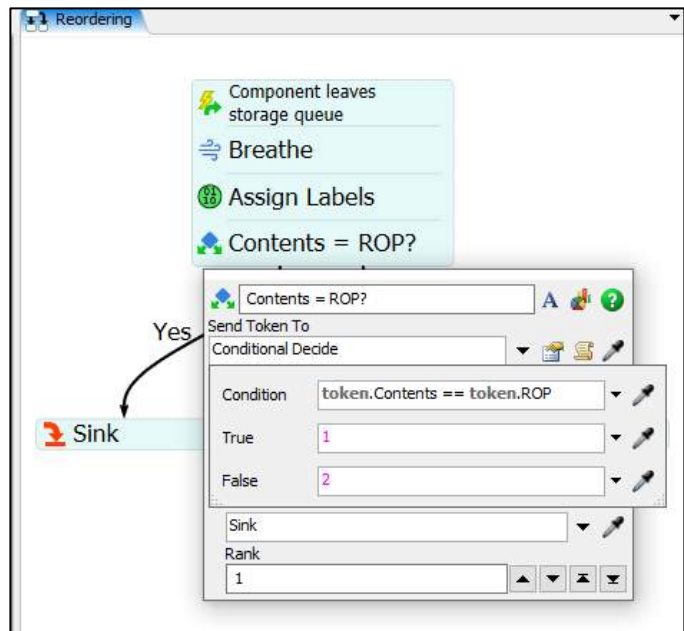
A **Decide** activity decides whether the reorder point has been reached and whether an order needs to be placed.

- As shown in the figure to the right, add a **Decide** activity to the block of activities.
- Name the activity *Contents = ROP?*
- In the **Send Token To** box, use the dropdown menu and select *Conditional Decide*.
- In the Condition box, enter as shown, **token.Contents == token.ROP**

Be sure to enter the double equal signs in the statement. As noted before, = is an assignment operator, and == is a comparison operator.

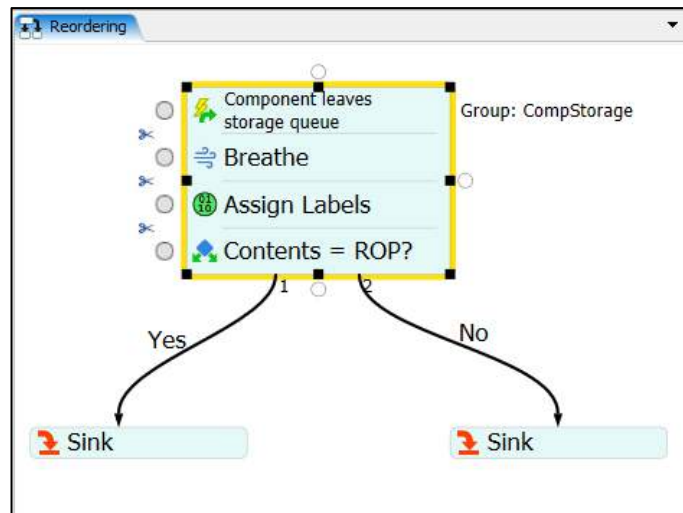
This statement compares the values of two tokens - one that contains the current contents of the storage object and the other the component's reorder point.

If the two values are equal, the statement is True, and the token is routed out of Branch 1 of the activity. If the values are not equal, the statement is False, and the token is routed out of Branch 2.



- As shown in the figure to the right, snap the activities in the order shown.
- Drag our two **Sink** activities.
- Connect the block of activities to the **Sink** on the left, which becomes Branch 1. Then, connect the block of activities to the **Sink** on the right, which becomes Branch 2.

It is important to connect the branches in this order because if the **Decide** activity's conditional statement is true, the token will exit the activity through the left branch, Branch 1.



- Double-click each branch connector and label them Yes and No, as shown in the figure.

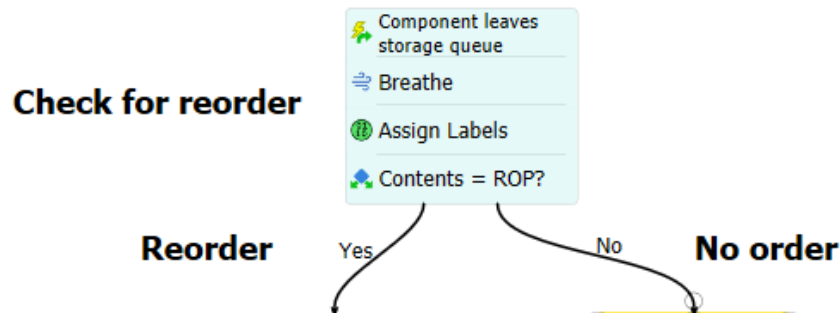
The **Sink** on the left will be replaced in the next section with the reordering logic. However, the current configuration is used to test the logic to ensure it is working properly before adding more activities.

Note in the figure above that the highlighted activity block or individual boxes in Process Flow can be resized to make the text more readable. Just drag the handles (small black squares) on the shape's perimeter to resize in that direction.

Text

Use the **Text** activity in the **Display** section of the Activity Library to label the parts of the logic as shown in the figure below.

- Drag out three Text activities and type in the text; resize and reformat as desired.



If you haven't already done so, save the model. Recall that it is good practice to save often.

23.2.2 “Reorder” logic

If a component's inventory level is at its reorder point, then:

- Additional information about the component is needed, such as order quantity, order time, etc.
- The storage area is colored black to indicate an order has been placed
- A delay is incurred that represents the order replacement time
- At the end of the order delay, an order is created and delivered to the storage table
- The Finishing Operator is called to the storage area
- The Finishing Operator loads, moves, and unloads each component in the order to its storage location
- When all components are unloaded, the Finishing Operator is freed to perform other tasks, and the color of the storage object is set back to its original color

The logic below implements this process. The additional information needed in the logic is obtained through an **Assign Labels** activity.

Assign Labels

Create the token labels shown in the figure to the right.

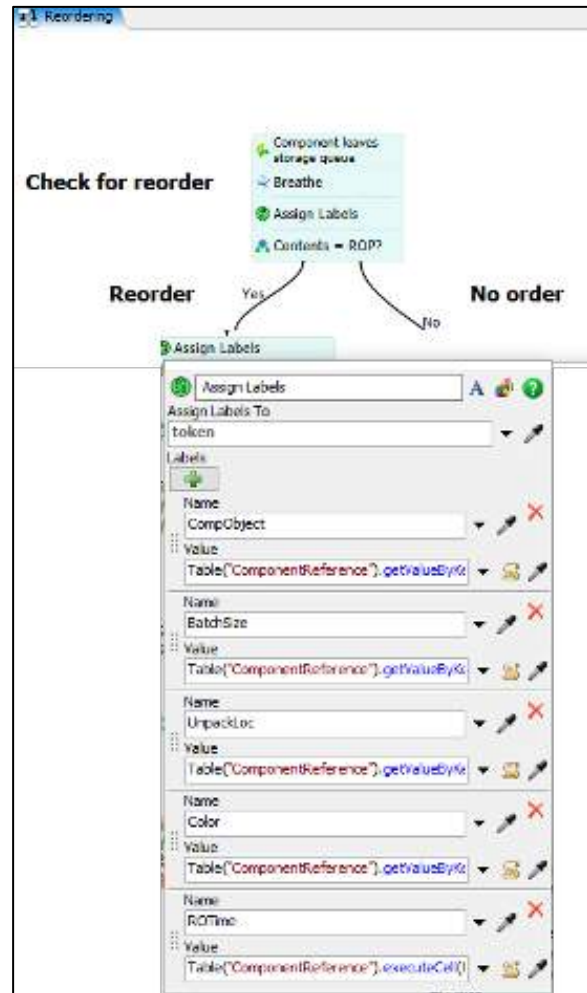
Each label is added by pressing the  button. Names are typed in and values can be typed in directly, or they can be entered, at least partially, by using the dropdown menu.

- *CompObject*'s **Value** is obtained from a powerful expression that searches the **Global Table ComponentReference** for specific information.

In this case, it searches the table for the component's object, which is the component that will be created when an order is unpacked.

To get the object, the expression finds the row in the table where the value in the column named *CompType* matches the value of the token label **CompType**. In that same row, it gets the value in the *Object* column and stores it in the token label *CompType*. (token.CompType)

The full expression is shown below.



Table("ComponentReference").getValueByKey(token.CompType, "Object", CompType)

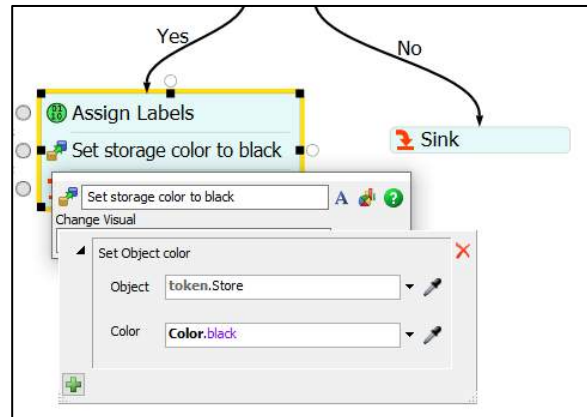
The next three labels obtain their values in a similar manner through a lookup in the **Global Table ComponentReference**.

- token.BatchSize uses the expression
Table("ComponentReference").getValueByKey(token.CompType, "BatchSize", CompType)
The batch size parameter is the reorder quantity.
- token.UnpackLoc uses the expression
Table("ComponentReference").getValueByKey(token.CompType, "UnpackLoc", CompType)
- token.Color uses the expression
Table("ComponentReference").getValueByKey(token.CompType, "Color", CompType)
- The last label, ROTime is obtained the same way as the ROP value defined in the first Assign Labels activity. It uses the expression
Table("ComponentReference").executeCell(token.CompType, "ROTime")
which executes the code statement in the **Global Table ComponentReference**'s cell in the row given by the value of the token label *CompType* and column named "ROTime" in the table.

Change Visual

The **Change Visual** activity is used to change the color of the component's storage location to black to signify the component has been reordered. It uses the token label **Storage** to reference the location. Add the activity as shown in the figure to the right and described below.

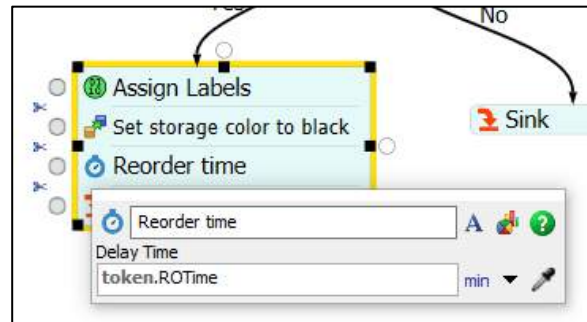
- Select the **Change Visual** activity from the **Basic** section of the Process Flow Library.
- Name the activity *Set storage color to black*.
- For the **Change Visual** property, using the  button, select the **Set Object Color** option, then
 - Set the **Object** property to *token.Store*.
 - Set the **Color** property to *Color.black* from the drop-down menu.



Delay

Delay the token the length of the reorder time, as described below and shown in the figure to the right.

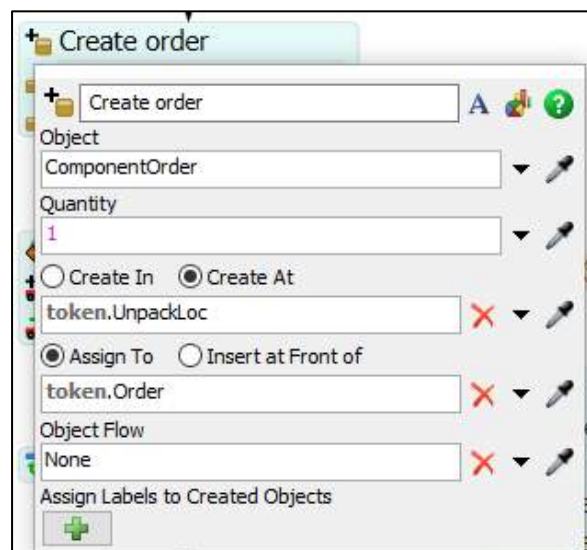
- Select the **Delay** activity from the **Visual** section of the Process Flow Library.
- Name the activity *Reorder time*.
- For the **Delay Time** property, use the dropdown menu to select **Token Label**, then *ROTime*.



Create Object

Once the order time has passed, a component order is created by the **Create Object** activity at the component's storage table. Add the activity shown in the figure to the right and described below.

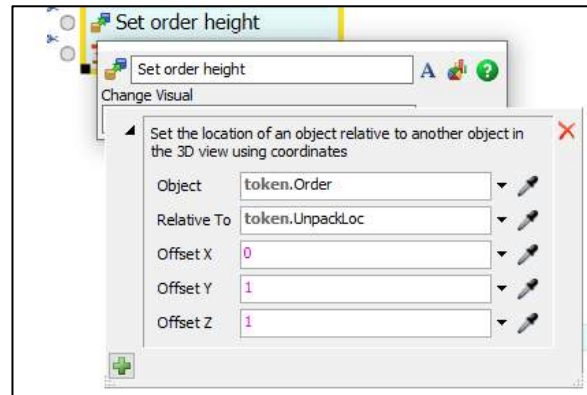
- Select the **Create Object** activity from the **Objects** section of the Process Flow Library.
- Name the activity *Create order*.
- For the **Object** property, use the dropdown menu to select **Flowitems**, then *ComponentOrder*.
- Leave the default value of 1 for **Quantity**.
- For the **Create At** property, use the dropdown menu to select **Token Label**, then *UnpackLoc*.
- For the **Assign To** property, type *token.Order*.



Change Visual

The order will come into the table object on the floor, so it needs to be raised to table level. This activity is for aesthetics and does not affect system performance. Add the activity shown in the figure to the right and described below.

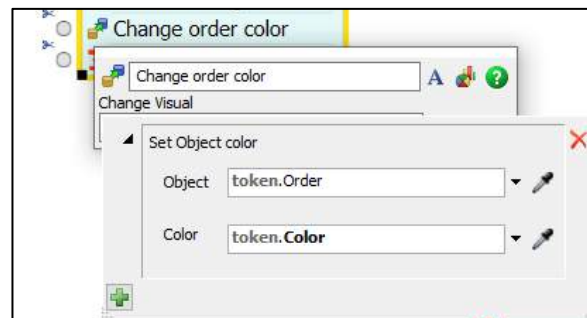
- Select the **Change Visual** activity from the **Visual** section of the Process Flow Library.
- Name the activity *Set order height*.
- In the **Change Visual** textbox, using the dropdown menu, select *Set Location to Object Location*, then set the following values
 - **Object** to *token.Order* either by typing it or using the dropdown menu and selecting **Labels**, then *token.Order*.
 - **Relative** to *token.UnpackLoc* either by typing it or using the dropdown menu and selecting **Labels**, then *token.UnpackLoc*.
 - Offset **X** to 0.
 - Offset **Y** to 1.
 - Offset **Z** to 1.



Change Visual

Another **Change Visual** activity is used to change the color of the order to the component's color. Again, this activity is for aesthetics and does not affect system performance. Add the activity shown in the figure to the right and described below.

- Select the **Change Visual** activity from the **Visual** section of the Process Flow Library.
- Name the activity *Set order color*.
- In the **Change Visual** textbox, using the dropdown menu, select *Set Object color*, then set the following values
 - **Object** to the *token.Order* either by typing it or using the dropdown menu and selecting **Labels**, then *token.Order*.
 - For the **Color** property, type *token.Color*.

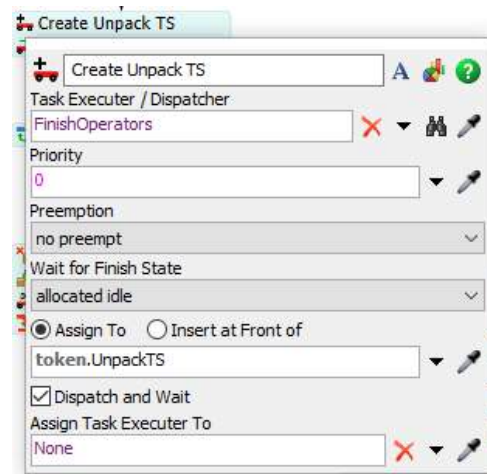


Many of the following activities involve getting a Finishing Operator to the Component Storage Area and unpacking an order to resupply a component's inventory. All of the operator's tasks, such as travel, load, and unload, are grouped into a "task sequence" so the operator is not interrupted by other tasks. If the individual tasks are not part of a defined task sequence, then if another task arises for the operator, the operator will perform that task. Thus, the task sequence will retain the operator in the storage area until all those tasks are complete. Of course, if another task has preemption privileges, the sequence would be interrupted. However, that case is not considered here.

Create Task Sequence

Create a task sequence for a Finish Operator to unpack the components as described below and shown in the figure to the right.

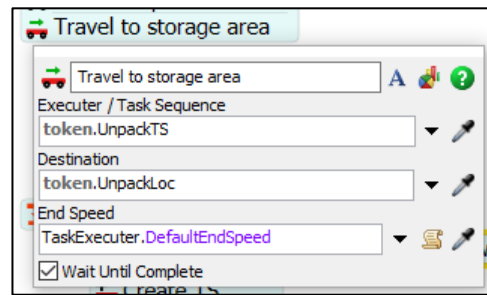
- Select the **Create Task Sequence** activity from the **Task Sequences Assets** section of the Process Flow Library.
- Name the activity *Create Unpack TS*.
- In the **Task Executer / Dispatcher** textbox, using the dropdown menu, select *Dispatcher* then *FinishOperator*.
- In the **Assign To** textbox, change *token.taskSequence* to *token.UnpackTS*
- For **Apply Task Executer To**, use the  button to delete the default value; this should result in the value *None*.



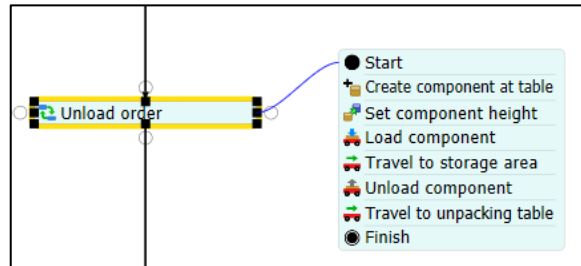
Travel

The **Travel to Object** activity sends the operator to the storage area; thus, add the activity described below and shown in the figure to the right.

- Select the **Travel To Object** activity from the **Task Sequences Assets** section of the Process Flow Library.
- Name the activity *Travel to unpacking table*.
- Set the **Task Executer / Dispatcher** value to *token.UnpackTS*.
- Set the **Destination** value to *token.UnpackLoc*.



Once the operator is at the storage area, the components are unpacked individually. Thus, the unpacking activities will be in a loop, i.e., the logic will loop through a set of unpacking activities for each component. This is modeled in Process Flow using the **Run Sub Flow** activity. The activities in the loop are bounded by the **Start** and **Finish** activities. The completed logic is shown in the figure to the right, but its construction is explained step-by-step below.

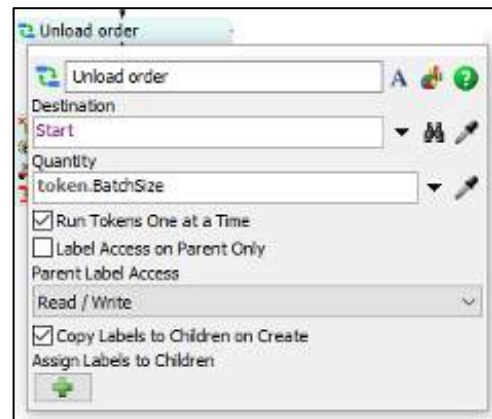
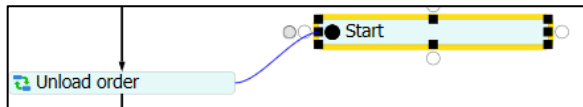


Note that each activity is attached to the previous one, forming a block.

Run Sub Flow

The **Run Sub Flow** activity creates “children” tokens that loop through a defined set of activities. The activity defines where the looping starts, how many times the loop is executed, and instructions on how the loop operates. Add the activity described below and shown in the figure to the right.

- Select the **Run Sub Flow** activity from the **Sub Flow** section of the Process Flow Library.
- Select the **Start** activity from the **Sub Flow** section of the Process Flow Library and place it near and to the right of the Run Sub Flow activity, as shown in the figure below.



- Define the **Run Sub Flow** activity properties as below and in the figure above.
 - Name the activity *Unload order*.
 - For **Destination**, use the sampler tool to select the **Start** activity added above. A blue line should now connect the two activities.
 - For **Quantity**, use the dropdown menu to select the token label named *BatchSize*; thus, the value should be *token.BatchSize*.
 - Check the box **Run Tokens One at a Time**
 - Uncheck the box **Label Access to Parent Only**
 - Check the box **Copy Labels to Children on Create**

The sub-flow activities are defined below.

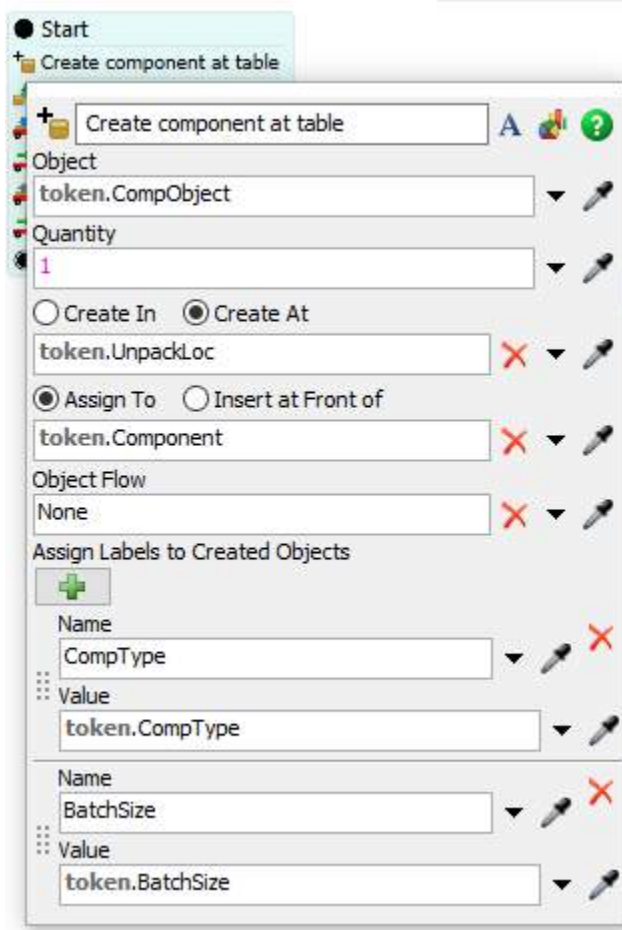
Create Object

The component to be unloaded is created at the storage table using the activity described below and shown in the figure to the right.

- Select the **Create Object** activity from the **Objects** section of the Process Flow Library and attach it to the **Start** activity.
- Name the activity *Create component at the table*.
- Set the **Object** value to *token.CompObject*. The object being created is a component flow item.
- The **Quantity** value remains *1*.
- The **Create At** value is *token.UnpackLoc*. The component is created at the storage table.
- The **Assign To** value is *token.Component*. This provides a reference to the component that was just created.

In the **Assign Labels to Created Objects** section, create the two labels on the component item. The labels are the container type and its batch size.

- Use the  button to add a label and assign the values as follows.
 - Name the label *CompType*.
 - The value *token.CompType*.
- Use the  button to add a label and assign the values as follows.
 - Name the label *BatchSize*.
 - The value *token.BatchSize*.

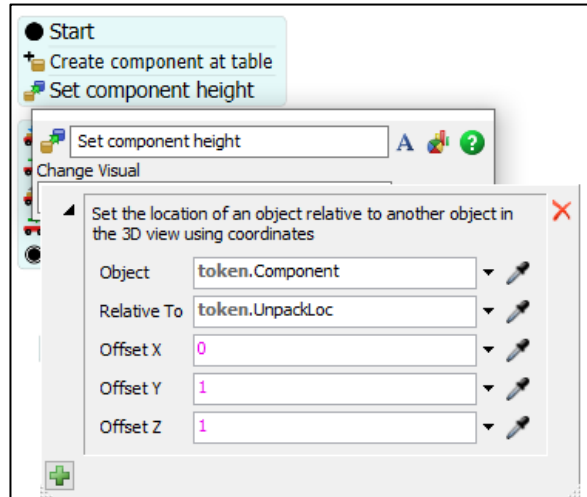


The screenshot shows the configuration window for the 'Create component at table' activity. The 'Object' field is set to 'token.CompObject' and the 'Quantity' is '1'. The 'Create At' radio button is selected, with the location set to 'token.UnpackLoc'. The 'Assign To' radio button is selected, with the target set to 'token.Component'. The 'Object Flow' is set to 'None'. In the 'Assign Labels to Created Objects' section, two labels have been added: 'CompType' with value 'token.CompType' and 'BatchSize' with value 'token.BatchSize'. Each label entry includes a red 'X' icon for removal and a pencil icon for editing.

Change Visual

Similar to what was done with the order flow item, by default, the component order will come into the table object on the floor, so it needs to be raised to table level, as shown in the figure to the right and described below.

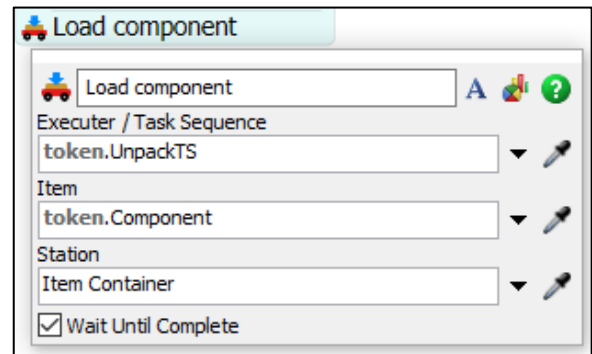
- Select the **Change Visual** activity from the **Visual** section of the Process Flow Library and attach it to the **Create Object** activity.
- Name the activity *Set component height*.
- In the **Change Visual** textbox, using the dropdown menu, select *Set Location to Object Location*, then set the following values
 - **Object** to *token.Component* either by typing it or using the dropdown menu and selecting **Labels**, then *token.Component*.
 - **Relative** to *token.UnpackLoc* either by typing it or using the dropdown menu and selecting **Labels**, then *token.UnpackLoc*. This positions the item relative to the table object.
 - **Offset X** to 0.
 - **Offset Y** to 1.
 - **Offset Z** to 1.



Load

The Finishing Operator picks up (loads) the component, which takes 0.05 minutes (3 seconds). The load time was defined previously as a property of the **Task Executer** object *FinishingOperator_1*. The activity is defined below and shown in the figure to the right.

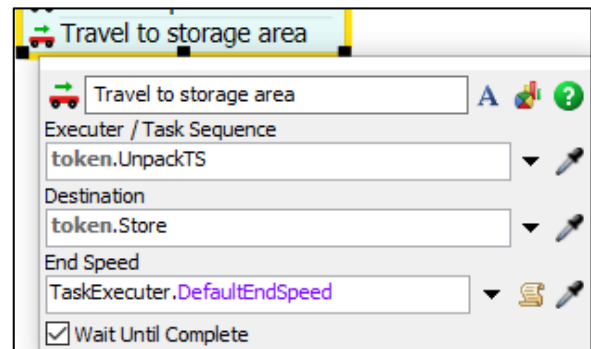
- Select the **Load** activity from the **Task Sequences** section of the Process Flow Library and attach it to the **Change Visual** activity.
- Name the activity *Load component*.
- Set the **Task Executer / Dispatcher** value to *token.UnpackTS*.
- Set the **Item** value to *token.Component*. This is what the Task Executer loads.



Travel

After loading the component, the Finishing Operator travels to the component's storage area. In this case, it is very nearby. The activity is defined below and shown in the figure to the right.

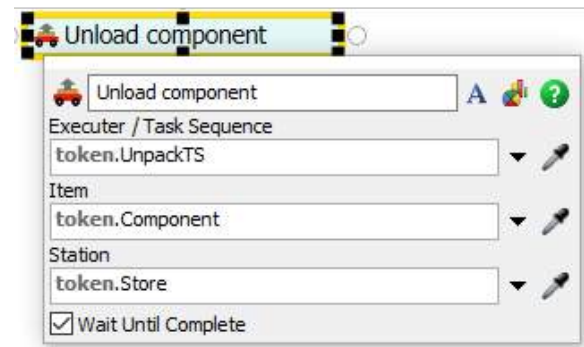
- Select the **Travel To Object** activity from the **Task Sequences** section of the Process Flow Library and attach it to the **Load** activity.
- Name the activity *Travel to storage area*.
- Set the **Task Executer / Dispatcher** value to *token.UnpackTS*.
- Set the **Destination** value to *token.Store*.
- Set the **End Speed** value to *TaskExecuter.DefaultEndSpeed*.
- Check the **Wait Until Complete** checkbox.



Unload

Once the Finishing Operator reaches the storage area, the component is unloaded, which takes 0.05 minutes (3 seconds). The unload time was defined previously as a property of the **Task Executer** object *FinisingOperator_1*. The activity is defined below and shown in the figure to the right.

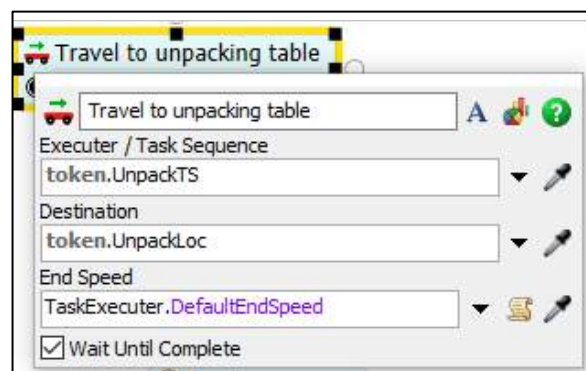
- Select the **Unload** activity from the **Task Sequences** section of the Process Flow Library and attach it to the **Travel** activity.
- Name the activity *Unload component*.
- Set the **Task Executer / Dispatcher** value to *token.UnpackTS*.
- Set the **Item** value to *token.Component*. This is what is unloaded.
- Set the **Station** value to *token.Store*. This is where the item is unloaded.



Travel

After unloading the component, the Finishing Operator returns to the location of the order object, which is at the unpacking table. The activity is defined below and shown in the figure to the right.

- Select the **Travel To Object** activity from the **Task Sequences** section of the Process Flow Library and attach it to the **Unload** activity.
- Name the activity *Travel to unpacking table*.
- Set the **Task Executer / Dispatcher** value to *token.UnpackTS*.
- Set the **Destination** value to *token.UnpackLoc*.
- Set the **End Speed** value to *TaskExecuter.DefaultEndSpeed*.
- Check the **Wait Until Complete** checkbox.



Finish

- Select the **Finish** activity from the **Sub Flow** section of the Process Flow Library and attach it to the **Travel** activity.

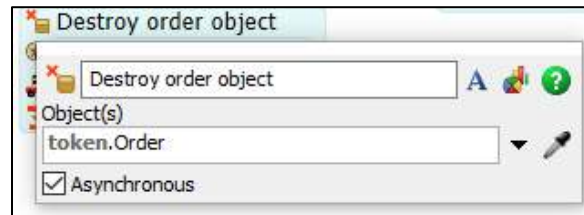


Continue the main logic stream, which is executed once all components are unpacked.

Destroy Object

The order object is no longer needed. It is removed from the model as described below and in the figure to the right.

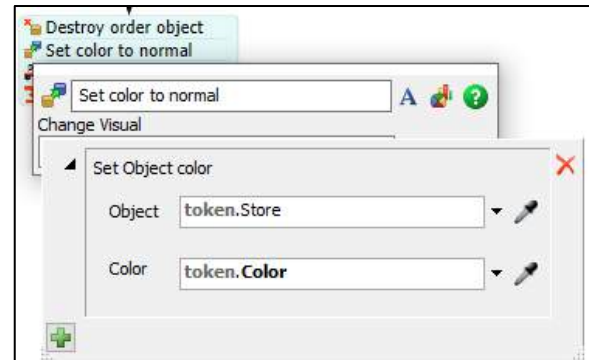
- Select the **Destroy Object** activity from the **Objects** section of the Process Flow Library.
- Name the activity *Destroy order object*.
- In the **Object** textbox, select or enter *token.Order*



Change Visual

The component's storage area color must be returned to its original color once the reorder has been fulfilled. This activity is for aesthetics and does not affect system performance. Add the activity as shown in the figure to the right and described below.

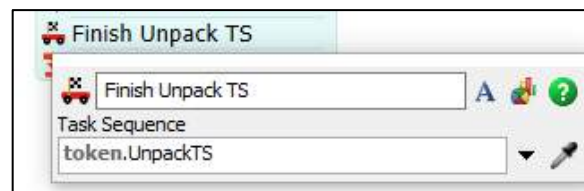
- Select the **Change Visual** activity from the **Visual** section of the Process Flow Library.
- Name the activity *Set color to normal*.
- In the **Change Visual** textbox, using the dropdown menu, select *Set Object color*, then set the following values.
 - **Object** to the *token.Store* either by typing it or using the dropdown menu and selecting **Labels** then *token.Store*.
 - For the **Color** property, type *token.Color*.



Finish Task Sequence

This concludes all of the tasks in the task sequence. It is concluded as described below and in the figure to the right.

- Select the **Finish Task Sequence** activity from the **Task Sequences Assets** section of the Process Flow Library.
- Name the activity *Finish Unpack TS*.
- Set the **Task Sequence** to *token.UnpackTS*.



Sink

The definition of the logic flow is complete, and the reorder token is destroyed.

- Use the **Sink** previously used to terminate the first branch or select the **Sink** activity from the **Basic** section of the Process Flow Library.



23.2.3 “No Reorder” logic

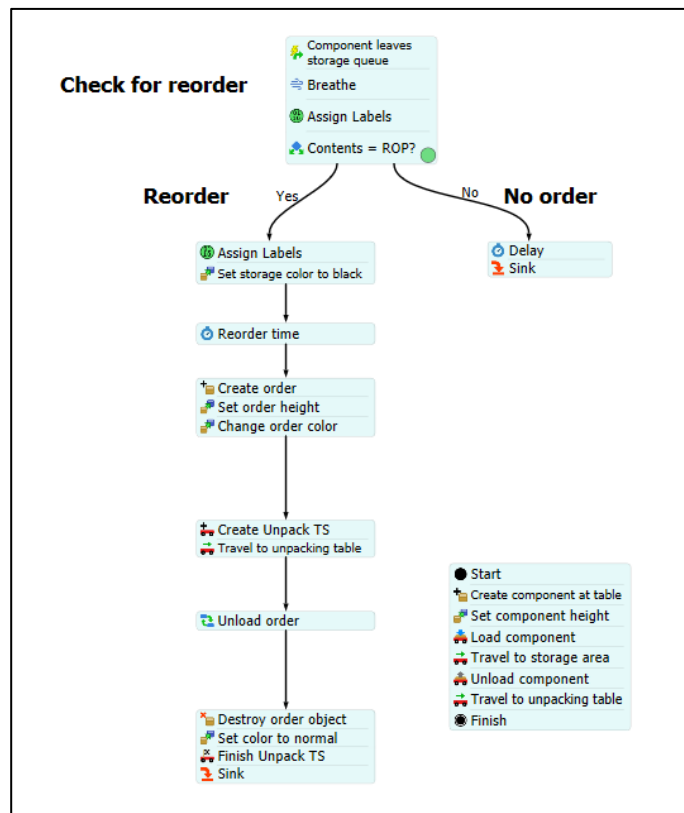
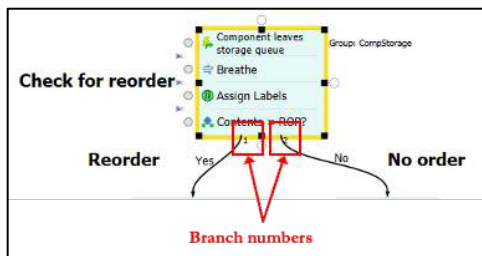
This branch includes the activities that are executed by a token when the use of a component in the packing area does not trigger a reorder event. The branch could be composed of just a **Sink**, but an arbitrary 10-minute delay is added so the modeler can see the token route to this branch and verify the model is working as expected. There is no effect on the operation or performance of the model.

- Drag out a **Delay** activity and set the **Delay Time** property to 10.
- Drag the activity to just above the **Sink** activity, and it should automatically connect to above the **Sink** activity, as shown in the figure to the right.



The final Process Flow logic should look like the figure to the right.

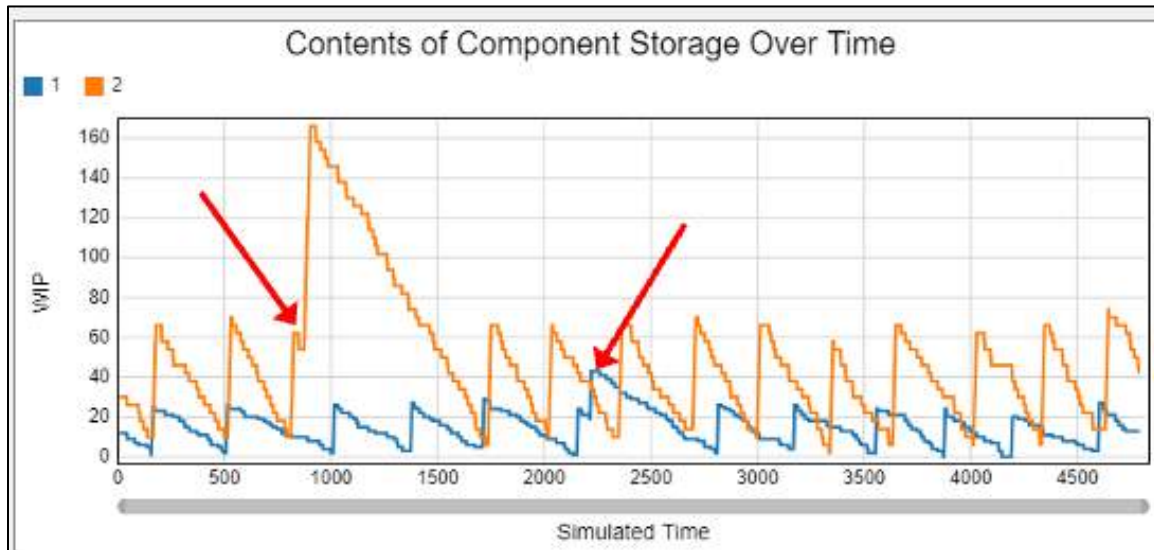
- Check to be sure the branches from “**Check for Reorder**” logic are still Branch 1 to *Yes* and Branch 2 to *No*, as shown in the figure below when the block is selected.



23.3 Validation & Modification

Use the **Dashboard** named *Components* that was created earlier in the primer as a means to test that the logic is correct.

- **Reset** and **Run** the model and examine the Dashboard, as shown in the figure below.



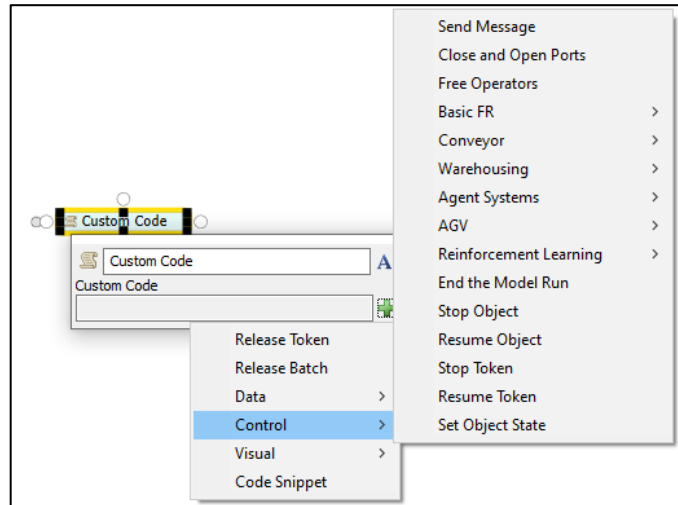
Generally, the graph looks as expected, with cyclic inventory levels. However, each component has one peak inventory level that is higher than expected. These are indicated by the red arrows in the figure.

After some investigation, here is what is happening. As the Finishing Operator unpacks an order, the inventory in the storage area increases to the reorder point, and then above. If an item is withdrawn for packing during this time interval, the reorder logic is triggered again. Of course, this would not happen in the real system, so the logic needs to be modified. As with many things in modeling, there are various ways to remedy this.

Upon further viewing of the model in action, it becomes apparent for safety reasons that the robot should not be operate while the operator is unpacking components. Therefore, the robot will be stopped when order when the operator arrives to unpack an order.

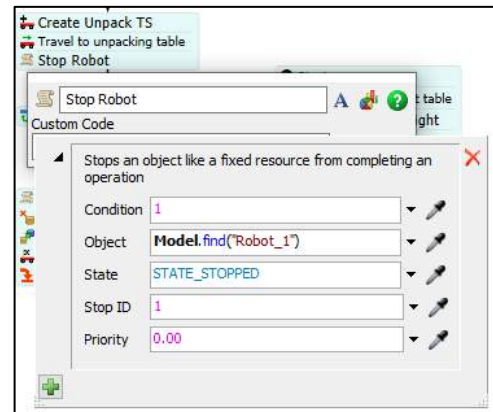
A **Custom Code** activity is used to implement this. However, that does not mean code actually has to be written. Many tasks are preprogrammed in the activity, as shown in the figure to the right.

There are several options in each of the categories: **Data**, **Control**, and **Visual**. The options in the **Control** section are shown in the figure since this is what is used to stop and start the robot.



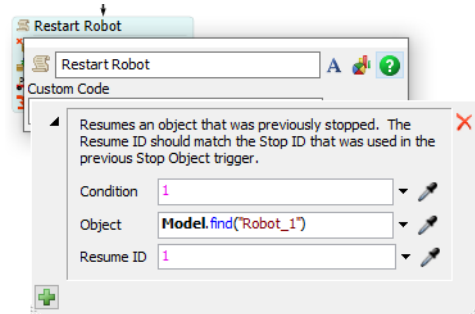
To stop the robot

- Select the **Custom Code** activity from the **Basic** section of the Process Flow Library.
- Name the activity *Stop Robot*.
- Use the  button to the right of the **Custom Code** textbox to select **Control**, then *Stop Object*.
- The only property that needs to change is the **Object**. Use the sampler tool to select the **Robot** object in the 3D view, or use the dropdown menu and select **Objects**, then *Model.find("Robot_1")*.
- Place the activity after the *Travel to unpacking table* activity and before the **Run Sub Flow** activity *Unload order*.

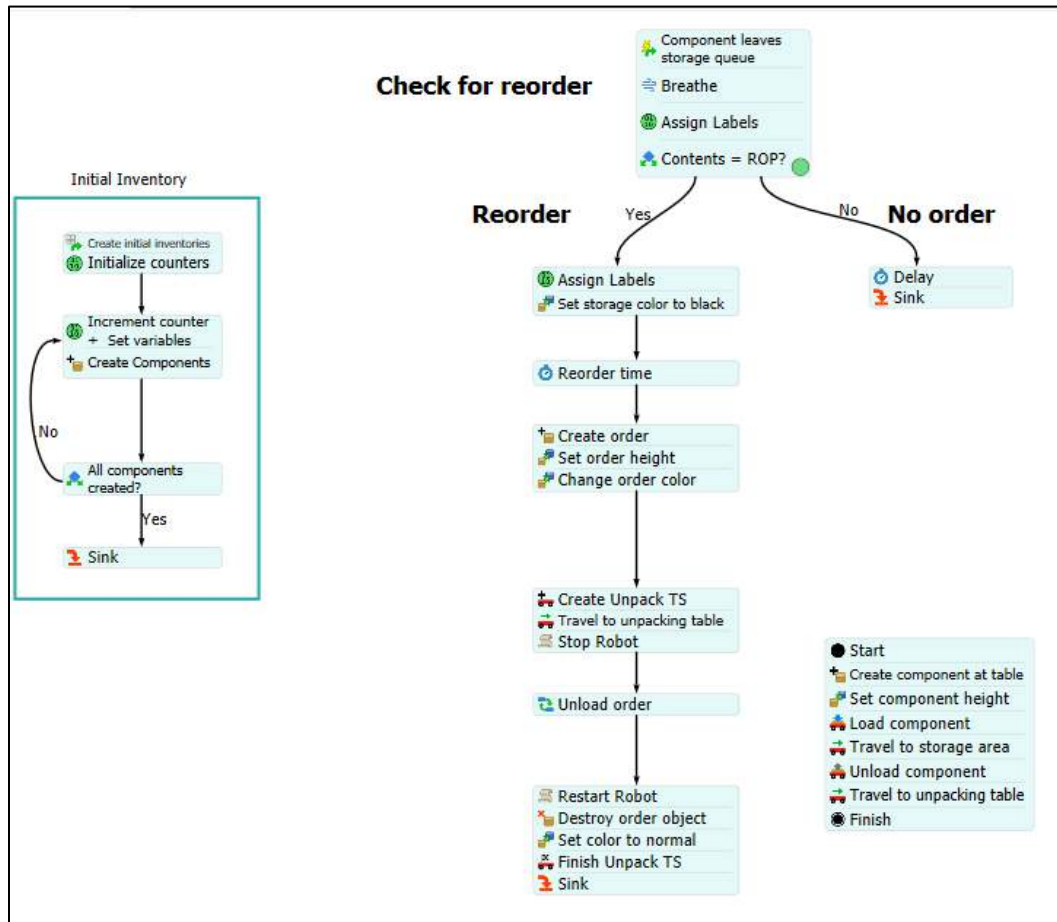


To restart the robot

- Select the **Custom Code** activity from the **Basic** section of the Process Flow Library.
- Name the activity *Restart Robot*.
- Use the  button to the right of the **Custom Code** textbox to select **Control**, then *Resume Object*.
- The only property that needs to change is the **Object**. Use the sampler tool to select the **Robot** object in the 3D view, or use the dropdown menu and select **Objects**, then *Model.find(Robot_1)*.
- Place the activity before the *Destroy order object* activity and after the **Run Sub Flow** activity *Unload order*.

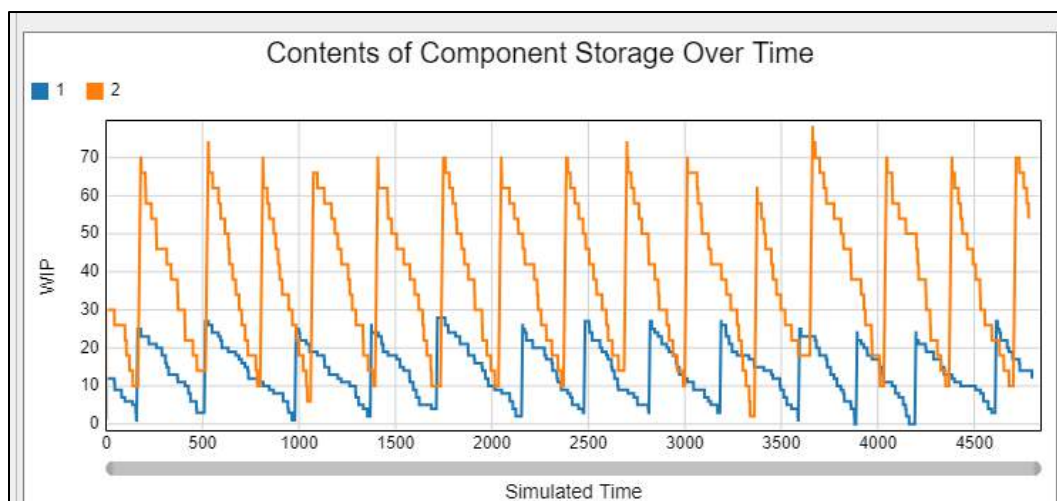


The overall *Reordering Process Flow* should resemble the figure below.



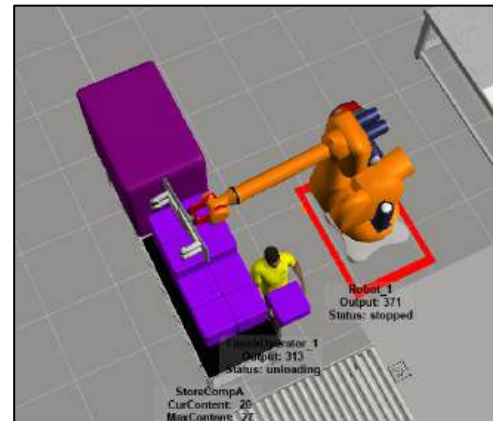
Again, use the **Dashboard** named *Components* to test that the logic is correct.

➤ **Reset** and **Run** the model and examine the Dashboard, as shown in the figure below.



Now, the graph looks as expected, with cyclic inventory levels.

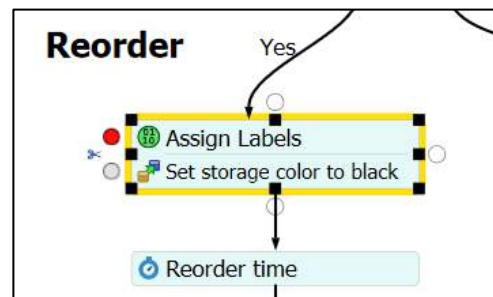
Also, note in the figure to the right that the robot is stopped when the Finishing Operator is unpacking components. The red square at the base of the robot indicates the object is stopped.



Two other means within Process Flow that help validate a model are introduced here – adding breakpoints to activities and coloring the tokens.

Breakpoints

As shown in the figure to the right, when selected each activity has a small gray circle to its left. If that circle is selected, it turns red, which means it is designated as a “breakpoint.” Clicking the circle when it is red, removes the breakpoint.



A breakpoint is basically a stopping point. When the simulation arrives at an activity that is marked as a breakpoint, *FlexSim* goes into “debug” mode, which limits some of model access and enables some debugging tools. The key point here is that the model runs until it hits a break and then stops.

When *FlexSim* hits a breakpoint, control is passed to a set of buttons, as shown in the figure to the right. The two buttons of interest here are:

- Step Token steps through the model one activity at a time. The current activity is denoted by the yellow arrow to the left of the activity, which is highlighted by the red circle in the figure to the right.



- Continue lets the model run as usual, but it will stop at the next breakpoint activity.

To stop checking using a breakpoint:

- Select the activity
- Click on the red circle, which will turn back to gray
- Press the Continue button

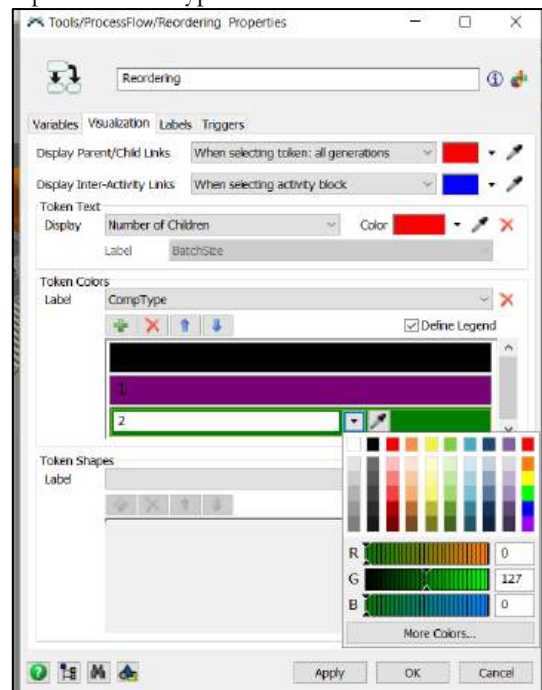
The control panel disappears and the model runs as normal.

Colored tokens

Using color is a great way to validate a model, such as in the 3D view, changing an object's color based on its state. In Process Flow, by default the tokens are all green; therefore it is difficult to know what type it is, other than double-clicking the token to display its label values.

In this example, the token color for the reorder Process Flow represents the type of token.

- Click anywhere on the Process Flow workspace, then select the **More Properties** button in the **Properties** window.
- On the resulting window, click the **Visualization** tab, which is shown in the figure to the right.
- In the **Token Colors** pane of the window, check the Define Legend box.
- Also in the **Token Colors** pane, select the **Label** dropdown menu and select **CompType**. This label contains a numeric reference to the type of component.
- Use the button to add two color bars which will be the color of the token based on the value of **CompType** (in this case, 1 or 2).



Note the first bar is the default and is colored black. A token will be black if, in this case, a **CompType** value is greater than 2. As component types are added to a model, their token colors must be updated here.

- Click on the color bar that has a value of 1 and use the dropdown menu to display the color palette as shown in the figure to the right. Select purple.
- Click on the color bar that has a value of 2 and use the dropdown menu to display the color palette as shown in the figure to the right. Select white.

Now, in Process Flow, tokens for Component A (CompType = 1) will be purple and tokens for Component B (CompType = 2) will be white.



If you haven't already done so, save the model. Recall that it is good practice to save often.



Use the **Save Model As** option in the **File** menu to make a copy of the existing model to be further customized in the next section. Again, you can use any file name, but the next model is referred to as Primer_18 in the primer.

24 USE OF RACKS TO STORE CONTAINERS IN THE WAREHOUSE

Chapter 24 introduces the Rack object and describes its use to store packed containers in the warehouse.

The model's next extension is adding a small Warehousing Area that follows the Packing operation. That is, containers that have been finished and packed with components flow to racks in a warehouse, where they are stored until they are used to fulfill demand. This introduces *FlexSim*'s Warehousing Module. For now, packed containers flow down a conveyor after packing and go directly into a storage rack in the warehouse. In a later chapter, the containers will travel to the warehouse using an AGV (automated guided vehicle).

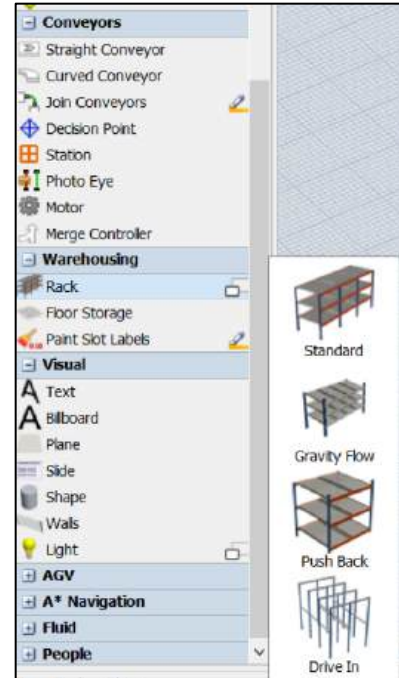
FlexSim provides a robust set of tools for modeling warehouse operations. These tools can represent complex aspects of warehousing and storage systems. Since this is a primer, the warehousing capabilities are only introduced through a simple example. However, this should provide the basic concepts and a foundation for exploring the topic further. In any case, as has been stressed throughout the primer, it is always best to start simple, have a clear definition of the problem being addressed, and have well-defined operational objective(s) for developing the simulation model.

Objects are accessed through the **Warehousing** section of the **Object Library**. The most commonly used object, and the only one discussed in this primer, is the **Standard Rack** object, hereafter referred to as a **Rack**. There are other types of racks in the Library for handling specific systems, e.g., **Gravity Flow**, **Push Back**, and **Drive In** racks. While these differ from the **Standard Rack**, they are structured similarly.

The first time a Warehouse object is dragged into a model, a **Storage System** object is added to the **Toolbox**. Most of the properties in the **Storage System** are for use in more advanced models, such as custom addressing schemes, special visualization, and triggers.

The base model for the additions described in this chapter is **Primer_17** that was saved at the end of Chapter 22. However, a copy of that file was saved as **Primer_18**; thus, we begin with that file.

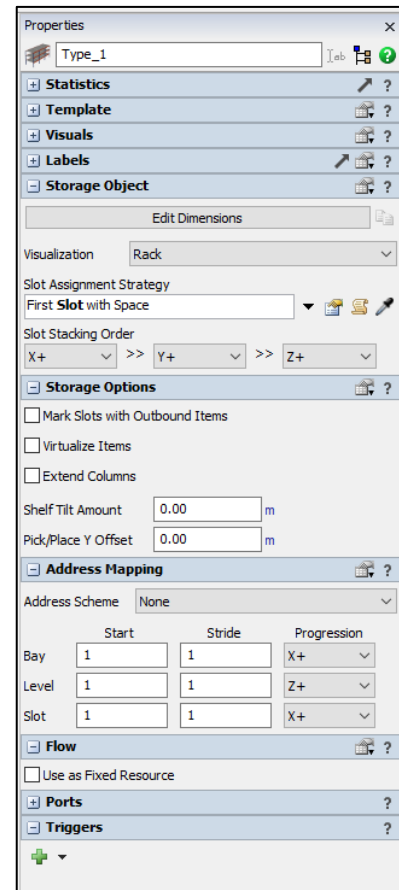
- Select the **Standard Rack** from the submenu in the **Warehousing** section of the object library, as shown in the figure to the right.
- Drag out the object into the model near the end of the conveyor from packing.



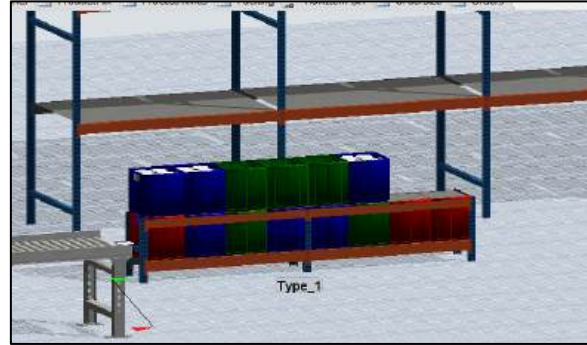
The **Properties** window for the **Rack** object is shown in the figure to the right. It has four unique panes: **Storage Object**, **Storage Options**, **Address Mapping**, and **Flow**. These are in addition to the standard panes, such as **Statistics**, **Visuals**, **Triggers**, etc.

This primer just uses the properties in the **Storage Object** pane. All of the defaults are used except for **Edit Dimensions**, which is described below. However, the other properties are briefly defined here. The following refers to “slots” in the rack; this term is defined below.

- **Visualization** provides a dropdown menu of options on how the **Rack** looks in the 3D view.
- **Slot Assignment Strategy** provides a dropdown menu of options for determining where an incoming item is stored. The default is *First Slot with Space*.
- **Slot Stacking Order** defines how incoming items are stacked within an individual slot. The default is to fill the rack's horizontal (x) direction first, then the depth (y-direction), and finally stacking in the z-direction.

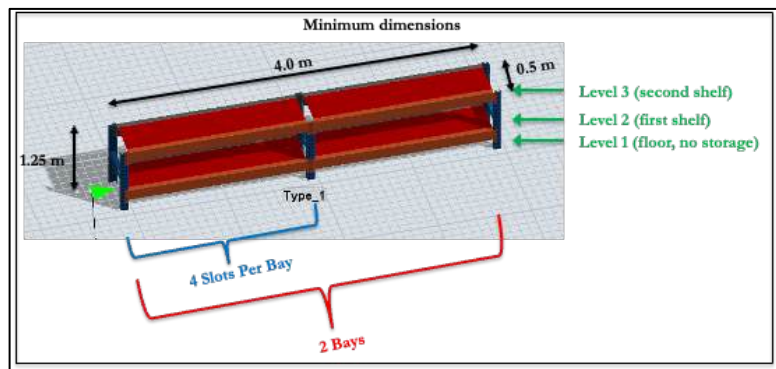


In the screenshot to the right, the default **Rack** object is in the background, and the customized **Rack** used in this model is in the foreground. Among other things, the **Rack** will be resized for the primer model. Also, note in the figure that the **Rack** stores all three types of containers; in the primer example, each type will be stored on its own rack.



The minimum rack size for DPI is shown in the figure to the right. It also shows the object's dimensions and a few key rack properties.

Bays are sections of a **Rack** that represent storage areas along the horizontal axis. Bays are subdivided horizontally into **Slots**. Bays also have **Levels**, which are storage areas in the vertical dimension. Each bay/slot/level combination is referred to as a **Cell**. Each cell may be considered either storable or not. In this case, a **Rack** has two bays, each with four slots, and there are three levels, but the lower level is not storable.

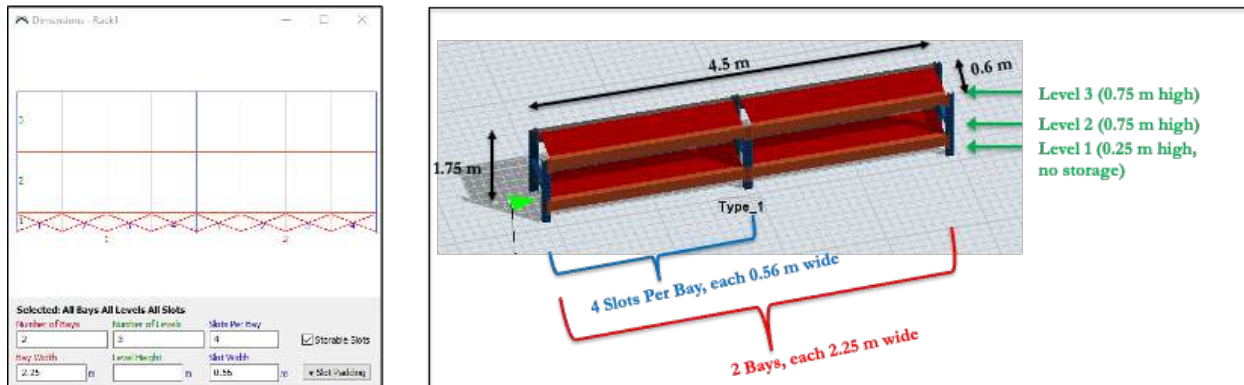


Recall that each container is a 0.5-meter cube and assume DPL wants to store containers only one unit deep on the rack. Therefore, the minimum width of the Rack in the x-direction is 4.0 meters (0.5 meters/container * 4 slots (container)/bay * 2 bays/Rack). The minimum depth in the y-direction is 0.5 meters ((0.5 meters/container * 1 container deep). The minimum height in the z-direction is 1.5 meters (3 levels * 0.5 meters/level); however, no containers will be stored on the bottom level. If the lower level is only 0.25 meters high, then the minimum rack height is 1.25 meters, as shown in the figure above. However, these calculations assume no space between containers or between the containers and shelves, which would make handling difficult. Thus, additional space will be added later.

The capacity of a rack is the product of the number of items stored in the x, y, and z directions. In this case, the rack's capacity is 16 (8*1*2).

The **Edit Dimensions** button in the **Storage Object** pane defines information on a rack's bays, slots, and levels. The number of bays and bay width automatically set the rack's size in the x-direction (width), and the number of levels determines the object's size in the z-direction (height). The size in the y-direction (depth) is set by the y-value in the **Visuals** pane, and it determines how many items deep the containers can be stored.

The **Edit Dimensions** button in the **Storage Object** pane opens the interface shown in the left part of the figure below. The screenshot on the right below shows the final object with dimensions.

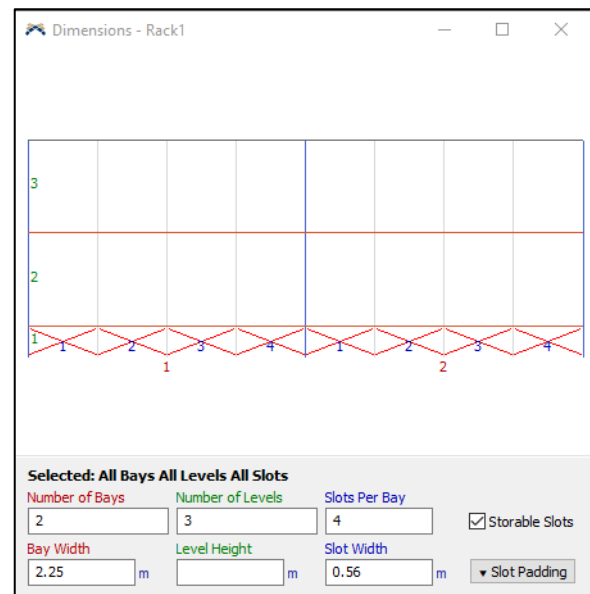


The following steps customize the **Standard Rack** to the **Type_1** object used in this example.

- Rename the rack object *Type_1*.
- Using the dropdown menu in the **Visuals** pane, set the object's color to red.

The settings for the **Edit Dimensions** interface are shown in the figure to the right and explained below.

- As shown in the figure, update the **Rack** properties for *Type_1*.
 - Set the **Number of Bays** to 2.
 - Set the **Bay Width** to 2.25. The additional 0.25 meters of space above the minimum provides space in the bay so the containers are not directly next to each other
 - Set the **Number of Levels** to 3.
 - Set the **Level Height** to 0.5. This will make each level 0.5 high, but the value will be adjusted later.
 - Set the **Slots Per Bay** to 4.
 - Set the **Slot Width** to 0.56. Again, the additional 0.06 meters (about 2.34 inches) will provide space between the stored containers.



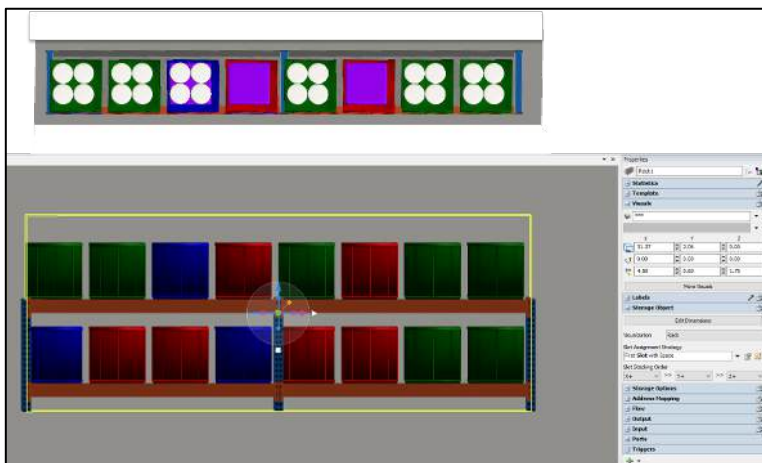
- In the **Visuals** pane, the size properties should be $x = 4.5$, $y = 0.6$, and $z = 1.5$. The x and z values are set automatically based on the entries in the **Edit Dimensions** interface. The y value is set manually and is based on how many items deep the rack will contain. In this case, containers will be stored only one deep so the depth is 0.6 meters, which provides 0.1 meters (about 3.9 inches) additional space.

In this case, the bottom level is not storable and is only 0.25 meters high. Each cell can be customized by selecting one or more cells in the diagram on the **Edit Dimensions** interface to highlight them. While highlighted the **Slot Width** and/or **Level Height** properties can be changed.

- Select all slots on the bottom row/level and set **Level Height** to 0.25.
- Also, uncheck the **Storage Slots** box since the lower level is not for storage; it is just to keep the items off the floor. Of course, no containers would be stored there anyway since a container's height exceeds the height of the cells in that level.
- Select the top two levels and set the **Level Height** to 0.75.

The loaded rack is shown in the figure to the right in terms of the top and side views. Note the spacing between containers and between the container and the next level.

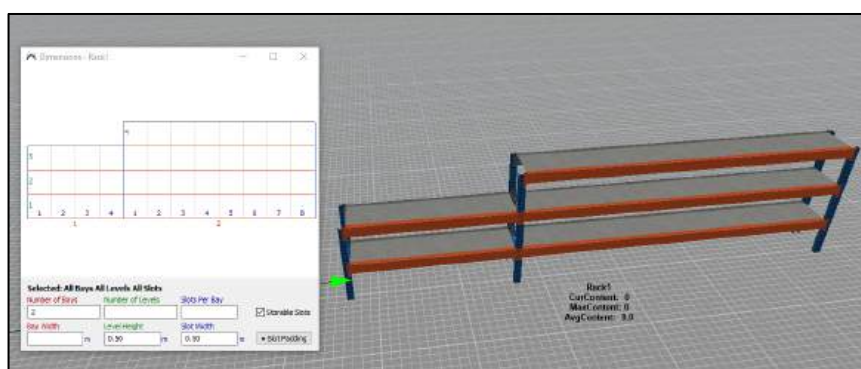
Of course, each rack will hold only one type of container. That will be handled via routing logic, which will be added later. This rack will be the template for the others.



Items entering a rack are assigned a slot. The assignment is made before the item is moved from the input object. If no space is available in the rack, the item waits in the input object.

The slot assignment method used in this case is the default. The method is defined on the **Rack's Storage Object** pane. The default method places containers in the rack starting at Bay 1, Slot 1, Level 2 (first storable level). They are filled as follows: all slots in Bay 1 Level 2, then all slots in Bay 1 Level 3, then all slots in Bay 2 Level 2, and finally all slots in Bay 2 Level 3.

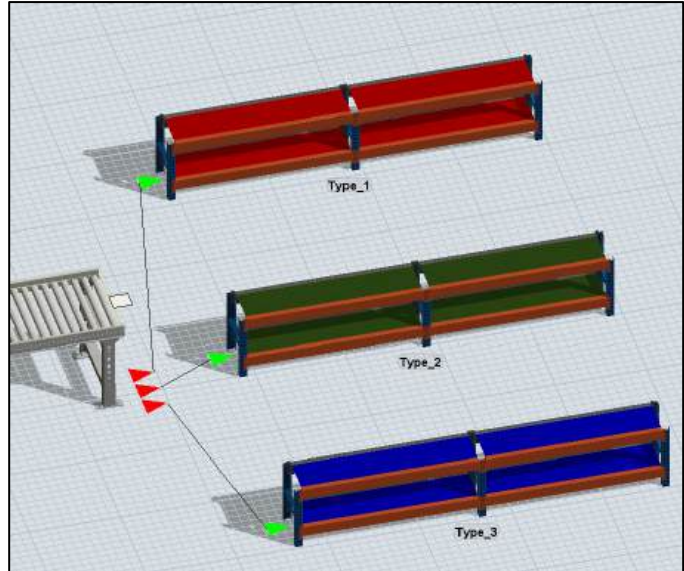
Rack configurations do not need to be symmetrical; *FlexSim's* warehousing module offers much flexibility. For example, the rack in the figure to the right has bays of different sizes, varying slots per pay, and various levels.



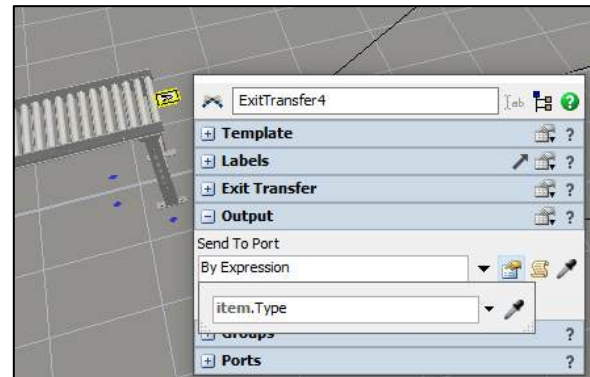
There are several ways to set a rack's color. For example, a label value on each rack could be used to set its color. Similarly, a rack's color could be set based on the label value of an entering item where the color is set by the **Rack's On Entry** trigger. This would work in the primer case since similar items, i.e., of the same type, are routed to the same rack. However, in this example, the color is set manually on each rack.

Set up the Warehousing Area of the model as shown in the figure to the right and explained below.

- Copy the *Type_1* Rack object twice for the other two types of container.
 - Rename one *Type_2* and set its color to green.
 - Rename one *Type_3* and set its color to blue.
- Position the racks on the layout and connect the **Conveyor's Exit Transfer** to each **Rack** in the order of *Type*; i.e., **Rack *Type_1*** should be connected to the **Exit Transfer's Port 1**, ***Type_2*** to **Port 2**, etc.



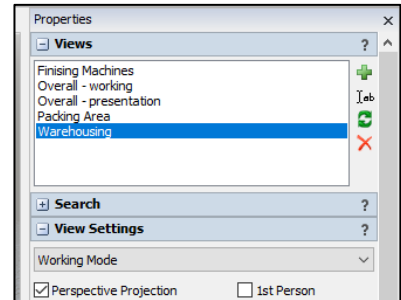
- Change the **Send to Port** property on the **Exit Transfer** to *By Expression* using the drop-down menu. Use the default value *item.Type* to transfer components from the **Conveyor** to a **Rack** based on the value of the container's label named *Type*.



Since there is no **Sink** in the flow, when the model runs, the racks will fill, containers will back up on the conveyors, blocking the Packing Area and the Finishing Machines, and finally, the containers' **Queue** at the beginning of the model. At that point, all subsequent entering containers will be redirected elsewhere. The next chapter adds the logic to represent order fulfillment, where containers are removed from the **Racks** to meet demand.

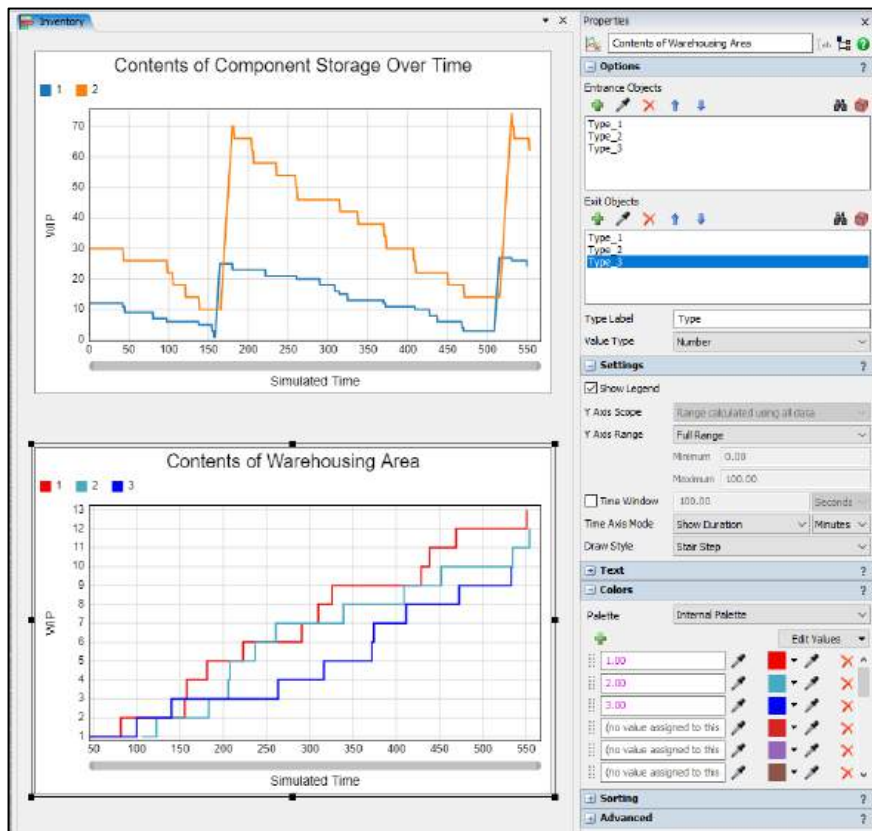
Create a view of the warehousing area

- Position the 3D view for a close-up view of the area, similar to the figure above.
- In the **Views** pane of the **Properties** menu, use the  button to add a view, then edit the name to *Warehousing*.



Create a graph of the contents of each rack in the warehousing area, as shown in the figure to the right and explained below.

- Select the existing **Dashboard** named *Components*.
- Change the name to *Inventory*.
- Since the existing chart on the dashboard is a **WIP by Type** chart, copy and paste it below the existing chart, which tracks the inventory level of the components in the Packing Area. A new chart could be created, but in this case, the copied chart is edited.
- Rename the chart *Contents of Warehousing Area*.



- For the **Entrance Objects** in the **Options** pane, select the two store objects and use the  button to remove them. Then, use the  button to add the three **Racks**.
- For the **Ext Objects** in the **Options** pane, select the two store objects and use the  button to remove them. Then, use the  button to add the three **Racks**.
- Change the **Type Label** from *CompType* to *Type*. The container's type is stored in its label named Type.

- Change the line colors in the graph to correspond to the **Rack** colors by using, as shown in the figure to the right, the **Colors** pane.
 - Use the colored square's dropdown menu to select the appropriate color for the entries 1.00, 2.00, and 3.00, which are the values for the container's Type label.



If you haven't already done so, save the model. Recall that it is good practice to save often.



Use the ***Save Model As*** option in the **File** menu to make a copy of the existing model to be further customized in the next section. Again, you can use any file name, but the following model is referred to as Primer_19 in the primer.

25 ORDER FULFILLMENT SUBMODEL

Chapter 25 defines the order-fulfillment process and describes how to represent it in the model using Process Flow. The process includes generating orders for containers, having an Order Picker gather the appropriate containers, delivering a completed order to a fulfillment area, and completing orders by updating an information system. Also, an output table is created that captures information on each order, including the contents of each order and the time it takes to fulfill orders. Charts have also been added that track the time it takes to fulfill an order and how many orders are waiting to be processed.

Packed containers are pulled from the Warehousing Area to meet demand. Demand for containers of different types is based on incoming orders. The following summarizes a simple representation of the ordering and fulfillment processes. This is followed by a description of how to model this in *FlexSim*.

25.1 Definition of the order-fulfillment process

Arriving orders are placed in a queue, and an Order Picker operator fulfills each order in the order in which it arrives. Only one order is processed at a time. However, if demand requires more than one operator, each operator processes one order at a time.

Orders are represented as a flat receptacle that holds an order's containers. For now, all receptacles are the same size regardless of the order size. A model enhancement could be that orders are packed in receptacles that depend on the order size. However, for high-level design and planning, a single size of receptacle is fine.

The Picking Operator gathers the requisite number and type of containers from their Racks in the Warehousing Area and places them on the order receptacle. When all of the order's containers are collected, the time to complete an order is triangularly distributed with a minimum of 0.75 minutes, a maximum of 1.5 minutes, and a most likely value of 1.25. This results in an average processing time of 1.17 minutes. The Picking Operator transports completed orders to a fulfilled-orders area, which is the end of the model; i.e., what happens to orders beyond this point is not considered in the model. The operator enters order information into a computer before processing the next order. The information entry time is assumed to be 15 seconds (0.25 minutes).

The Order Pickers follow the same break schedule as the Finishing Operator.

The size of each order is a random variable. It is assumed there is a request for one container in an order 10% of the time, two containers 15% of the time, three containers 50% of the time, four containers 15% of the time, and five containers 10% of the time. This is referred to as the order-size distribution. Based on this distribution, an order consists of one to five containers, and the average order size is 3.0 containers.

The product mix of ordered containers is assumed to be the same as those used to produce the containers. Recall that the product mix is defined as an **Empirical Distribution** named *ProductMix* where 30% of the containers are Type 1 (red), 35% Type 2 (green), and 35 Type 3 (blue).

The time between arrivals to the Finishing Area was assumed to also be an **Empirical Distribution** named *TimeBetweenArrivals* which was represented as a Weibull probability distribution with a maximum of 10 minutes and a maximum of 35 minutes. The distribution results in an average time between arrivals of about 20 minutes. It is assumed that the time between the arrival of orders is similar. Of course, this is the time between the arrivals of containers not orders.

Therefore, if the average time between arrivals of containers is 20 minutes and there are an average of 3.0 containers per order, then the average time between orders is about 60 minutes. The distribution of the order arrivals is assumed to be the product of the two empirical distributions or

Empirical["*TimeBetweenArrivals*"] * **Empirical**["*OrderSize*"]

Of course, once the products are being produced and sold, a more representative distribution of the order-arrival process can be determined. Still, for designing the production system, this should suffice.

As with most modeling, this aspect of the model could be modeled in various ways. A previous version of this primer used the logic within 3D objects to represent the order-fulfillment process. However, in this version, Process Flow is used to define the logic.

25.2 Implementation of the order-fulfillment process

The base model for the additions described in this chapter is **Primer_18** that was saved at the end of Chapter 24. However, a copy of that file was saved as **Primer_19**; thus, we begin with that file.

As has been the practice throughout the primer, each step in modeling the order-fulfillment process will be described in detail; however, the following is an overview of what will be discussed.

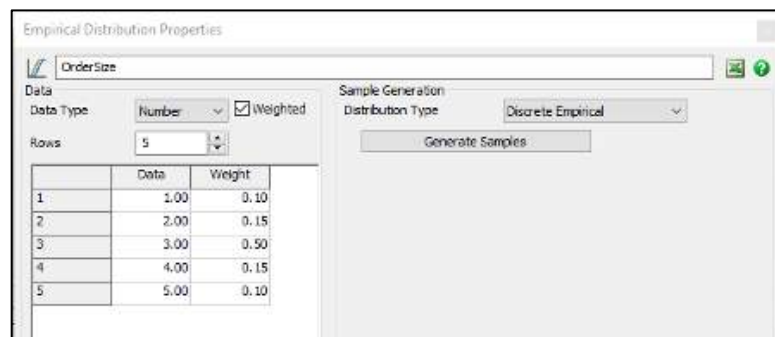
- Represent the order size distribution as an **Empirical Distribution**.
- Import the **Empirical Distribution** from *Excel*.
- Create a new **flowitem** that represents the order. This item is used to accumulate and transport the order's containers.
- Create a **Global Table** to output info on each order, such as the order number, composition of the order, time the order arrived into the system, and how long it took to fulfill the order.
- The Order-Fulfillment Area includes an input **Queue** for incoming orders, a table object for processing the orders, a place to output the fulfilled orders (**Sink**), and a home location for the Order Pickers, which is also where the operators enter order information.

- Use a **Dispatcher** and **Operator** to transport orders through the process and to move containers from the Warehouse Area to the Order-Fulfillment Area. The **Operator** uses the **A* Navigator** to guide its movement through the system and follows the same **Time Table** for breaks as the Finishing Operator.
- Add “virtual” **Queues** in front of each rack to help size the **Racks**. These objects will hold containers from the Packing Area if their designated **Rack** is full.
- For use in Process Flow, define a **Global Variable** to increment the order number.
- Also, for use in Process Flow, set up a **List** to manage the inventory of containers and order fulfillment. Containers will push information to a **List** when they enter a **Rack** and will be pulled from the **List** when they are needed to fulfill an order.
- Develop the **Process Flow** logic that represents the order-fulfillment process. This involves the mostly involves modeling the steps described in this list.
- Provide the capability to include initial inventory in the **Racks** at the beginning of a simulation. The initial inventory amount is specified in a **Model Parameters Table**, and the logic is defined in Process Flow.
- Define additional performance measures in the **Performance Measures Table** – the number of orders waiting to be processed and the total time it takes to fill an order.
- Create a **Dashboard** of charts to track the new performance measures - the number of orders waiting and the time to fulfill an order represented both over time and as a histogram.

Order-size distribution

The order-size distribution is modeled as an **Empirical Distribution**. To illustrate *FlexSim*’s interface with, the distribution is specified in an *Excel* file, which is read into the model when the model is **Reset**., if the *Excel* file has changed since the last run.

- Add an **Empirical Distribution** from the Statistics section of the **Toolbox** and name it *OrderSize*, as shown in the figure to the right.
- Check the **Weighted** box.
- Using the dropdown menu, set the **Distribution Type** to *Discrete Empirical*.
- Expand the table to 5 rows.
- Enter the values in the **Data** and **Weight** columns, as shown in the figure to the right.



Input the order-size distribution from Excel

To have the distribution read in from Excel:

- Create an *Excel* file in the same directory on your computer as the *FlexSim* model, or copy the *Excel* file named *OrderSize* from the primer's **Resources** folder. If you create the file:

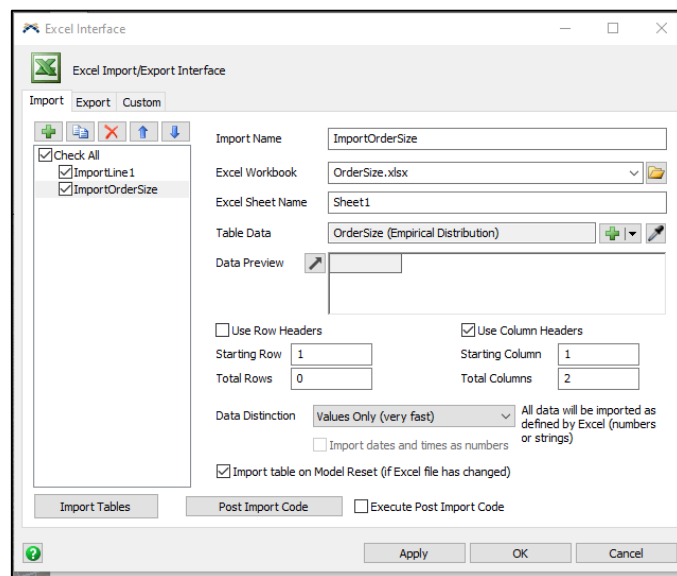
- Name the file *OrderSize*.
- As shown in the figure to the right, the spreadsheet should look like the table just completed above.
- Note that the first row is a header that defines what is stored in each column.

	A	B	C
1	Qty	Weight	
2	1	0.10	
3	2	0.15	
4	3	0.50	
5	4	0.15	
6	5	0.10	
7			

- Select the *Excel* icon in the upper right portion of the **Empirical Distribution** interface.

- Make the following changes to the *Excel Interface* window, as shown in the figure to the right.

- Since multiple imports from *Excel* may exist, set **Import Name** to *ImportOrderSize*.
- Use the folder icon to the right of the **Excel Workbook** property and browse your computer to find and select the *OrderSize.xlsx* file.
- For **Table Data**, use the dropdown menu to select *Empirical Distribution*, then select *OrderSize*.
- Check the **Use Column Headers** box.
- Check the **Import table on Model Reset (if the Excel file has changed)** box.



- Since this is the only import defined for this model, any other entries in the window, such as *ImportLine1* in the figure, can be deleted. To delete the entry, ensure only the one to be deleted is checked, then press the  button above the list.

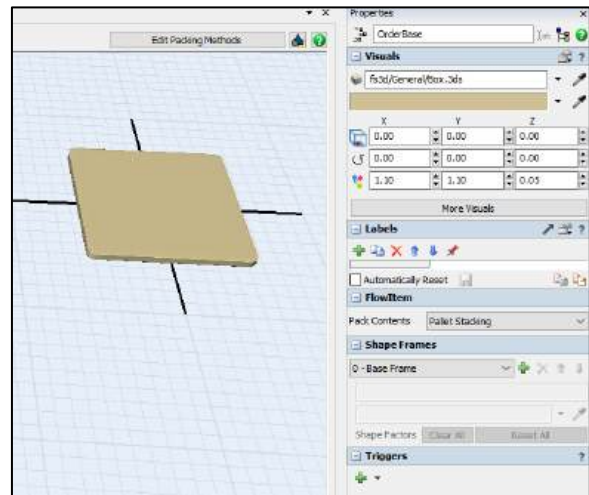
The import can be tested by changing a value in the *Excel* file, saving it, pressing the **Import Tables** button on the *Excel Interface* window, and checking the values in the **Empirical Distribution** *OrderSize*.

Whenever a model is **Reset**, *FlexSim* checks to see if the associated *Excel* file has changed. If it has, then it will automatically import the *Excel* data into the *FlexSim* table, in this case, the one for the **Empirical Distribution**

Flowitem for orders

As described in the introduction, the items (containers) for an order are placed on a new item as the order is processed.

- Create the item by duplicating the **Pallet** item in the **Flowitem Bin**; i.e., select the **Pallet** item and press the **Duplicate** button next to the  button. The **Pallet** and **Tote** are container-type flow items since they can contain other items.
- Change the properties as shown in the figure to the right and as described below.
 - Name the item *OrderBase*.
 - Using the dropdown menu, select *Box* to change the shape from a pallet to a box.
 - Set the *x*, *y*, and *z* dimensions to *1.10*, *1.10*, and *0.05*, respectively.
 - Set the *x*, *y*, and *z* center locations to *0.0*, *0.0*, and *0.0*, respectively.



Output table with information on orders

This table stores information on each order, which is an output from the model. Each time an order arrives, a row is added to the table, and the order information is added to that row. An example is shown in the figure to the right.

- Create a 6-column, 0-row **Global Table** named *Orders*.
- Name the column headers as shown in the figure (*OrderNum*, *TimeIn*, *TimeToFulfill*, *Type_1*, *Type_2*, and *Type_3*). These represent the order number, the time the order arrived, how long it took to fulfill the order, and the contents of the order in terms of the number of Type 1, 2, and 3 containers.
- In the **Properties** window, use the dropdown menu for **On Reset** to select *Delete All Rows*. This deletes all rows in the table when the model is **Reset**.

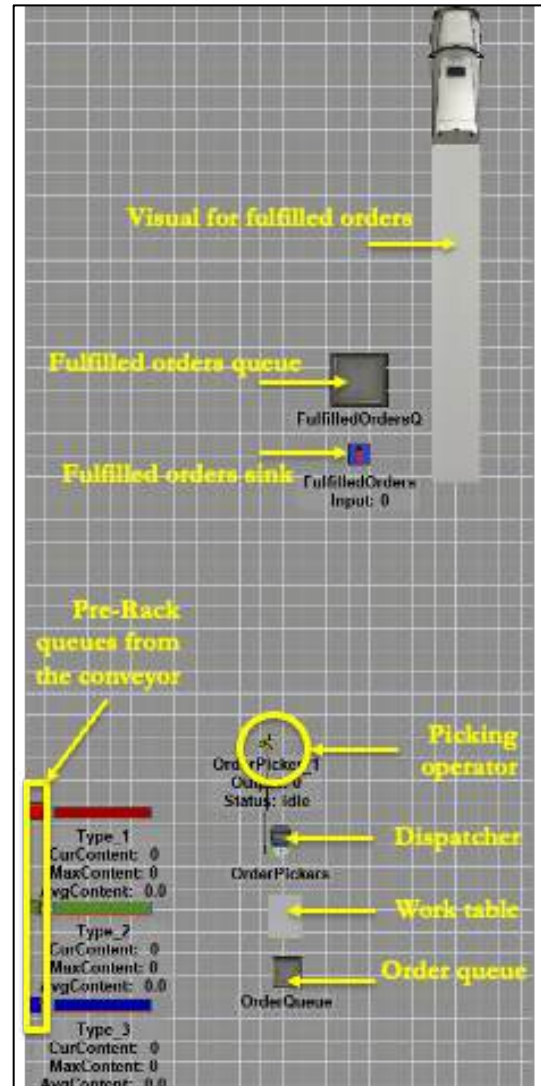
Orders						
OrderNum	TimeIn	TimeToFulfill	Type_1	Type_2	Type_3	
1	1.20	3.99	1	1	2	
2	1.85	3.36	0	1	1	
3	2.66	4.71	2	1	0	
4	3.53	3.44	1	0	1	
5	4.79	104.95	0	1	2	
6	6.33	83.34	1	0	1	
7	8.15	9.03	1	0	2	
8	9.29	20.12	1	2	1	
9	9.85	34.29	0	1	1	
10	10.51	85.32	1	1	1	
11	11.83	90.16	1	0	2	
12	13.46	28.35	1	1	0	
13	14.18	53.09	0	1	1	
14	15.40	3.91	0	3	1	
15	15.95	4.37	2	1	2	
16	17.02	3.32	1	0	1	

Layout the Order-Fulfillment Area

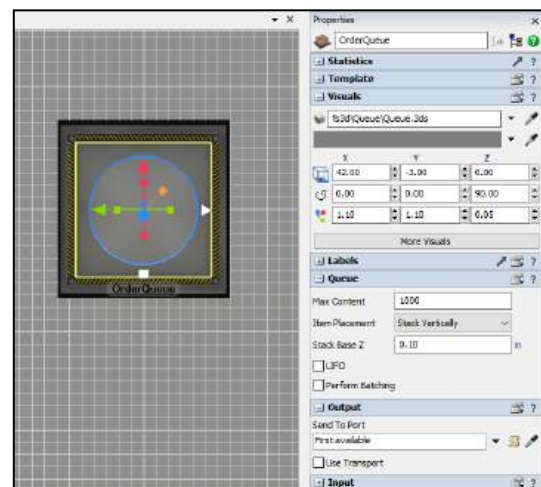
The area is shown in the figure to the right and is explained in detail below. It is located to the right of the Packing Area with enough space for the AGV network, which is discussed in the next chapter. For now, a single Picking Operator should be sufficient for the Order-Fulfillment Area.

Therefore the following objects are used:

- **Queue** for incoming orders.
- **Shape** for a work table to process orders
- **Dispatcher** for order pickers
- **Operator** for order pickers
- **Queue** for fulfilled orders
- **Sink** for fulfilled orders
- **Shape/Transporter** to hide Sink and Queue
- Three “virtual” **Queues** prior to **Racks**



- Drag out a **Queue** object and name it *OrderQueue*.
- Update the **Visuals** information and set the **Item Placement** to *Stack Vertically*, as shown in the figure to the right.
 - x , y , and z locations to 42.0, -3.0, and 0.0, respectively.
 - x , y , and z rotations to 0.0, 0.0, and 90.0, respectively.
 - x , y , and z sizes to 1.1, 1.1, and 0.05, respectively.

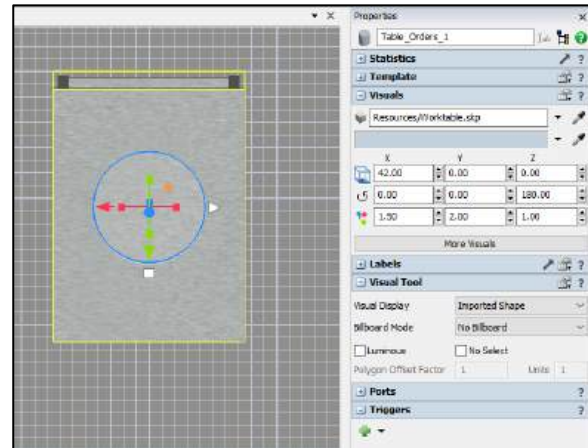


➤ Drag out a **Shape** object from the **Visual** part of the object **Library**.

➤ In the **Visuals** pane of the object's **Properties** window, use the **Browse** option in the dropdown menu to select the *Worktable.skp* file in the **Resources** folder.

➤ Update the **Visuals** information as shown in the figure to the right.

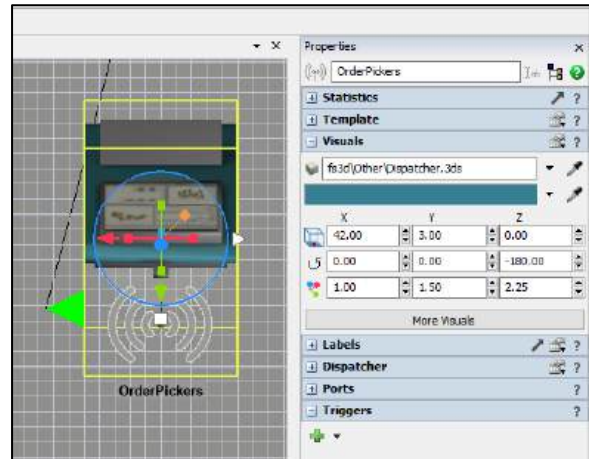
- x , y , and z locations to 42.0, 0.0, and 0.0, respectively.
- x , y , and z rotations to 0.0, 0.0, and 180.0, respectively.
- x , y , and z sizes to 1.5, 2.0, and 1.0, respectively.



➤ Drag out a **Dispatcher** object from the **Task Executors** part of the object **Library**.

➤ Update the **Visuals** information as shown in the figure to the right.

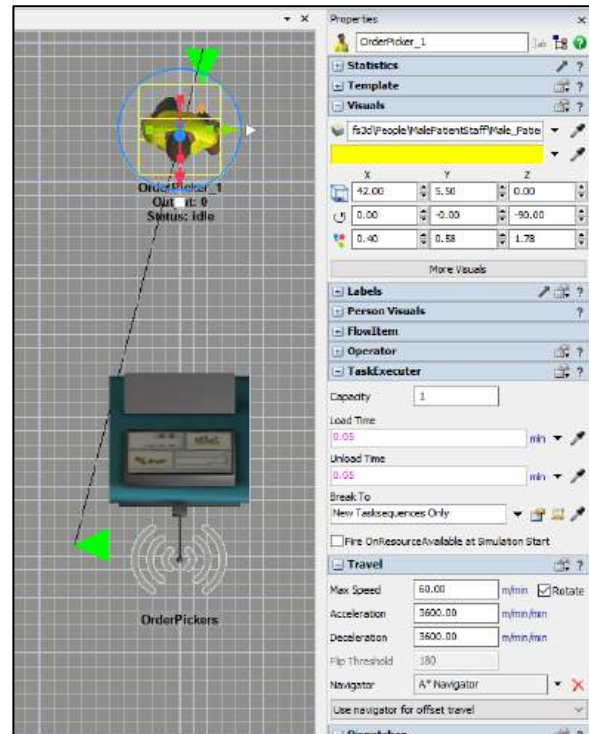
- x , y , and z locations to 42.0, 3.0, and 0.0, respectively.
- x , y , and z rotations to 0.0, 0.0, and -180.0, respectively.
- x , y , and z sizes to 1.0, 1.5, and 2.25, respectively.



Recall that the **Dispatcher** is used since multiple Order Pickers may be needed in the full system design.

Center-port or S connections between the **Dispatcher** and associated objects are not used because the relationships will be managed through custom task sequences in Process Flow.

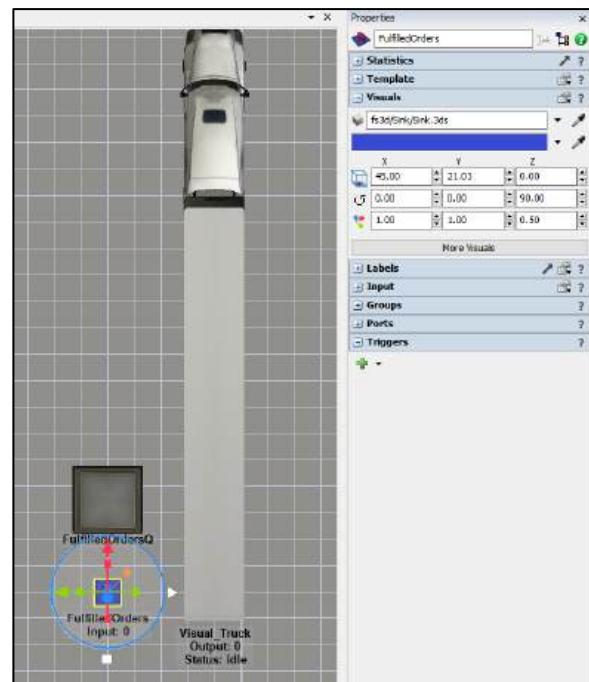
- Drag out an **Operator** object from the **Task Executors** part of the object **Library**.
- Update the **Visuals** information as shown in the figure to the right.
 - x , y , and z locations to 42.0, 5.5, and 0.0, respectively.
 - x , y , and z rotations to 0.0, 0.0, and -90.0, respectively.
 - x , y , and z sizes to 0.4, 0.58, and 1.78, respectively.
- Change the **Load Time** and **Unload Time** to 0.05 in the **Task Executor** pane.
- In the **Travel** pane, change:
 - **Max Speed** to 60.0
 - **Navigator** to A* Navigator.
- Make an A-connection from the **Dispatcher** to the **Operator**.



As shown in the figure to the right, there are three objects in the area where completed orders are taken. This is a model boundary or the last part of the system that the model represents; i.e., the model does not consider what happens in the real system beyond this point.

Only one object is required, the **Sink**, where orders go once they are fulfilled.

- Drag out a **Sink** object.
- Name it *FulfilledOrders* and update the **Visuals** information as shown in the figure to the right and described below.
 - x , y , and z locations to 45.0, 21.0, and 0.0, respectively.
 - x , y , and z rotations to 0.0, 0.0, and 90.0, respectively.
 - x , y , and z sizes to 1.0, 1.0, and 0.5, respectively.



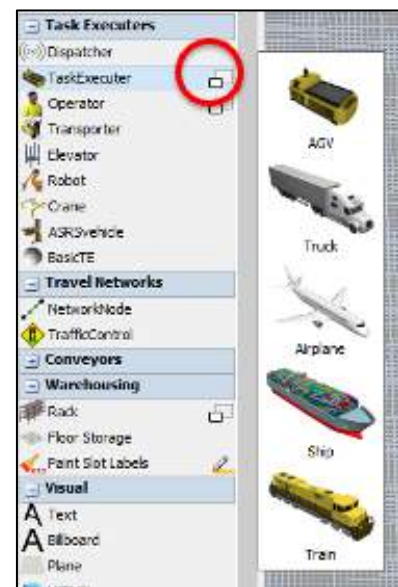
No connections need to be made to other objects because movement of the item from the packing table to the sink is handled by Process Flow and not the 3D objects.

The **Queue** is used to temporarily replace the **Sink** for verification and validation. Fulfilled orders build up in the **Queue** so they can be checked to be sure the orders are as expected and that the output data table contains the correct information on the completed orders. Once confirmed that the model is working as expected, the **Sink** replaces the **Queue**. The **Queue** object could be deleted or retained in case a change is made to the model that needs to be verified.

- Drag out a **Queue** object.
- Name it *FulfilledOrdersQ*.
- Change the **Item Placement** property to *Horizontal Line*.
- Update the **Visuals** information as shown below.
 - x, y , and z locations to 45.0, 23.0, and 0.0, respectively.
 - x, y , and z rotations to 0.0, 0.0, and 90.0, respectively.
 - x, y , and z sizes to 1.5, 1.5, and 0.1, respectively.

The truck shape hides the **Queue** and **Sink** and makes the model more visually appealing. It can be added to the model in two ways. First, use a **Shape** object from the **Visual** pane and then browse for the truck image. Second, the method used here is to use the **Truck** option from the **TaskExecuter** object. None of the **Task Executer** properties are used, just the shape.

- As shown in the figure to the right in the red circle, use the view expander in the **TaskExecuter** section of the **Task Executer** pane to drag out a **Truck** object.
- Name it *Visual_Truck*.
- Update the **Visuals** information as shown below.
 - x, y , and z locations to 45.0, 31.5, and 0.0, respectively.
 - x, y , and z rotations to 0.0, 0.0, and 90.0, respectively.
 - x, y , and z sizes to 22.0, 2.6, and 4.2, respectively.

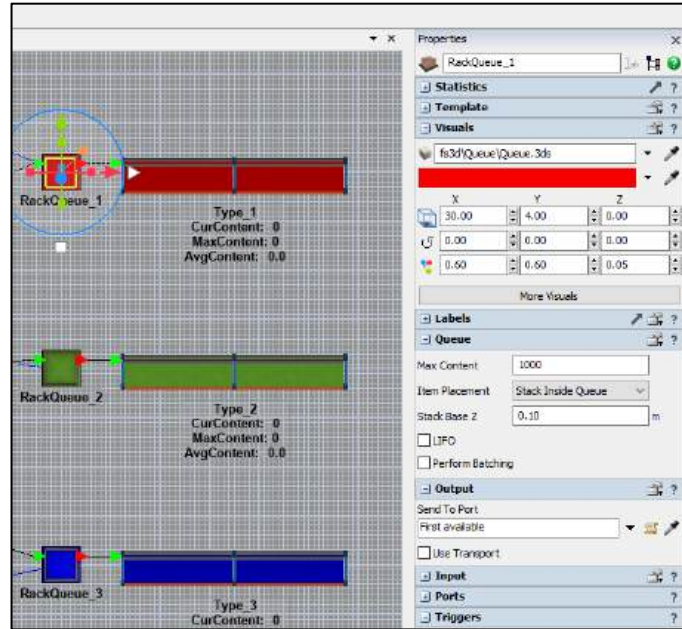


The **Queues** prior to the **Racks** in the Warehouse Area are temporary objects that help decide the capacity of the **Racks**; i.e., they are “dummy” objects – they will not exist in the real system; they are only used size the **Racks**.

Currently, each **Rack** can hold 16 containers. However, this may change as the system is being designed. The **Queues** will hold items from the Packing Area if there is no space in the container’s **Rack**. Without the **Queues**, if there is no space in a **Rack**, the item is held on the segment of conveyor after packing until space is available. If that conveyor gets full, the Packing Area becomes blocked and containers will continue to back up towards the Finishing Area.

The **Queues** are arranged as shown in the figure to the right. The figure also shows the properties for Type 1 containers.

- Drag out a **Queue** object.
- Name it *RackQueue_1* and change its color to *red*.
- Update the **Visuals** information as shown in the figure to the right and below.
 - *x*, *y*, and *z* locations to 30.0, 4.0, and 0.0, respectively.
 - *x*, *y*, and *z* sizes to 0.6, 0.6, and 0.05, respectively.



The remaining two **Queues** are created by copying and pasting the previous **Queue**.

- Copy and paste the *RackQueue_1* object.
- Change its name to *RackQueue_2* and change its color to *green*.
- Update the **Visuals** information as shown below.
 - *y* location to 0.0.
- Copy and paste the *RackQueue_1* object.
- Change its name to *RackQueue_3* and change its color to *blue*.
- Update the **Visuals** information as shown below.
 - *y* location to -4.0.

The connections from the **Conveyor** from packing to the warehouse must be changed so containers are routed to the **Queues**, not to the **Racks**.

- Disconnect the Packing Area's **Conveyor's** (*PackToNext*) **Exit Transfer** (*ExitTransfer4*) from the three **Racks**.
- For model readability, rename the **Exit Transfer** *ExitTransfer_PackToWhse*.
- Connect the Packing Area's **Conveyor's** (*PackToNext*) **Exit Transfer** (*ExitTransfer_PackToWhse*) to the three **Queues** in the following order: *red* (*Type_1*), *green* (*Type_2*), and *blue* (*Type_3*). The order is important since the **Send To Port** trigger in the **Exit Transfer** is *By Expression*, which will route the container to its **Rack** based on the value of its label *Type*.

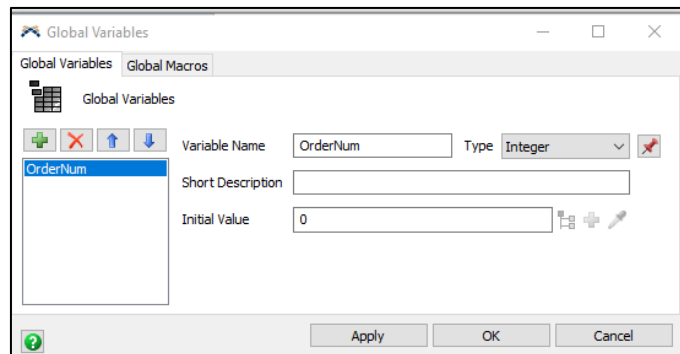
Variable to track order number

A **Global Variable** will be used in the order fulfillment logic in Process Flow. The variable will store the value of the order number, which is incremented with each arriving order. Global Variables are, as the name suggests, a variable that can be accessed anywhere in the simulation model.

- To add a **Global Variable**, click the  button in the **Toolbox**, then select **Modeling Logic**, and then select *Global Variable* from the dropdown menus.

As shown in the figure to the right,

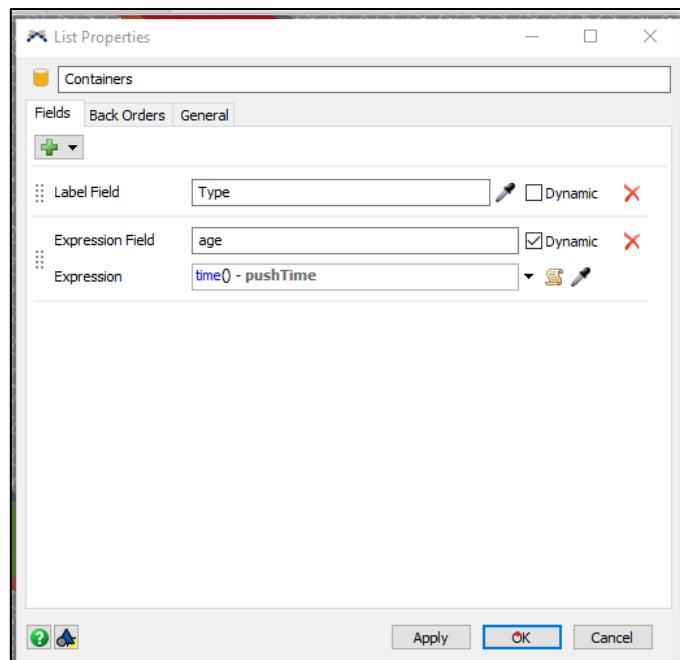
- Change the **Variable Name** to *OrderNum*.
- Using the picklist, set the **Type** to *Integer*.
- Set **Initial Value** to 0.



List to manage inventory

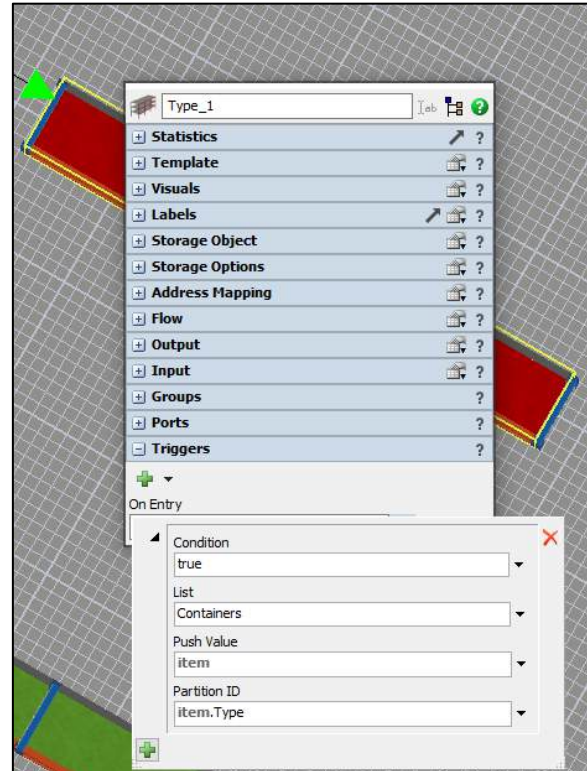
Also, for use in Process Flow, a **List** tool is used to manage the inventory of containers and order fulfillment. Containers will push information to a **List** when they enter a **Rack** object, and they will be pulled from the **List** when they are needed to fulfill an order. This logic is implemented in Process Flow. As shown in the figure to the right, create a new List and update the following properties.

- Add a **Global List** by clicking on the  button in the **Toolbox**, then select **Global List**, and then select *Item List* from the dropdown menus.
- Name the **List** *Containers*.
- Since all of the default fields are not needed, use the  button on the **Fields** tab to delete the last two **Fields** - *distance* and *queueSize*.

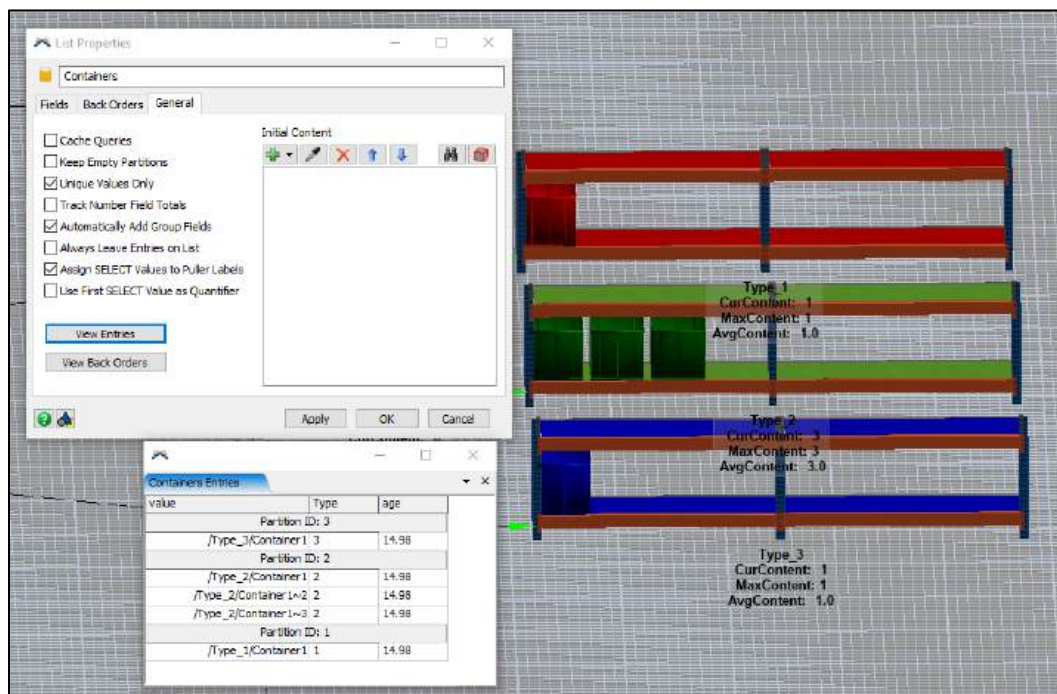


When a container enters a **Rack**, the item's information is pushed to the **List Containers**. This is shown in the figure to the right and described below.

- In the **Triggers** pane of the **Rack Type_1**, use the  button to open a list of possible triggers, and select *On Entry*.
- For the On Entry trigger, use the  button to open a list of possible actions - select *Lists* then select *Push To List*.
- On the resulting interface, use the dropdown menu to select *Containers* for the **List** property.
- Also on the interface, use the dropdown menu to select *Labels* then *item.Type* for the **Partition ID** property.



This **List** is partitioned by the value in the item/container's **Type** label; i.e., the list is divided into sections where information on all of the Type 1 containers are grouped together, all Type 2s, etc. Partitioning is invoked on the **Pull From List** operation in Process Flow. An example of a partitioned list is shown in the figure below.



The interface in the upper left portion of the figure above is the **List** tool's *General* tab. It contains a button to view the current entries on a **List** and to view backorders. Backorders are the objects that are waiting to pull something from a **List**. In the figure, the *View Entries* button is pushed and the current entries in the **List** are shown in the bottom left portion of the figure. Currently, the **List** contains one Type 1 container, three Type 2 containers, and one Type 3. The screenshot in the righthand portion of the figure above verifies this.



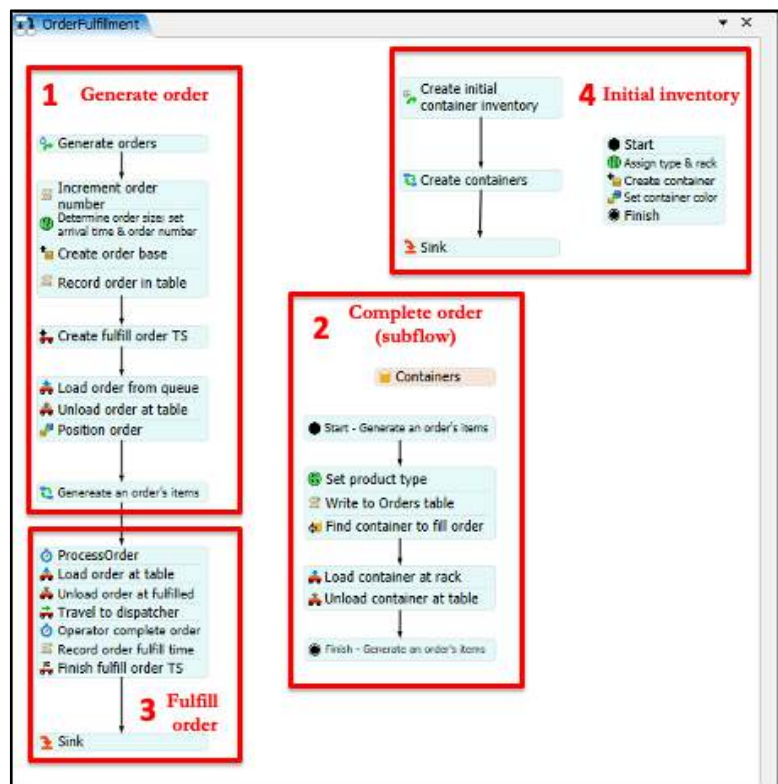
If you haven't already done so, save the model. Recall that it is good practice to save often.

25.3 Modeling order-fulfillment in Process Flow

The operational logic for the order-fulfillment process is modeled in Process Flow, as is the logic to include initial inventory in the **Racks** at the beginning of a simulation. An overview of the processes is shown in the figure to the right. That process is broken into four main parts to facilitate defining and discussing modeling the operation – generating, completing, and fulfilling orders and providing an initial inventory of containers in racks.

The Process Flow activities do not need to be arranged precisely as shown in the figure. This arrangement is based on the author's desire for structure and clarity and may vary depending on the modeler's preferences and style.

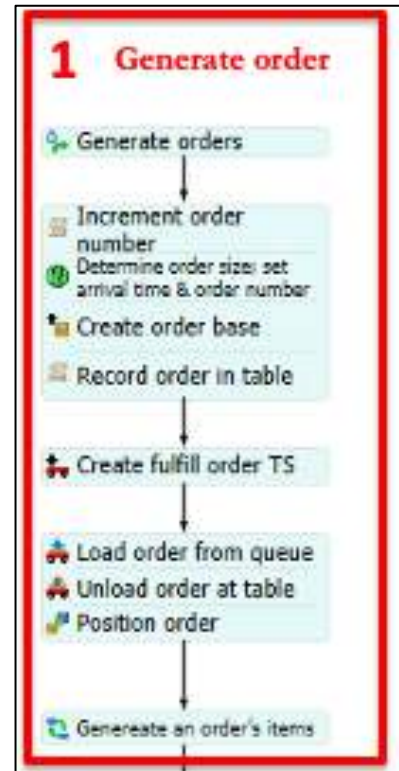
The following is a detailed discussion of how to model the operation in Process Flow.



- From the **Main** menu, select **Process Flow** and then *Add a General Process Flow*.
- Name the workspace *OrderFulfillment*.

25.3.1 Order-Fulfillment Process Part 1 - Generate Orders

The first part of the Process Flow logic used to define the order-fulfillment process is shown in the figure to the right and discussed in detail below. This part of the process generates orders and brings them into the system at an order queue.



Inter-Arrival Source

This activity randomly generates orders. As discussed earlier, the times between orders are generated from the product of samples from two empirical probability distributions – the time between arrivals of containers to the Finishing Area and order size.

- From the **Token Creation** pane, drag an **Inter-Arrival Source** activity onto the workspace.

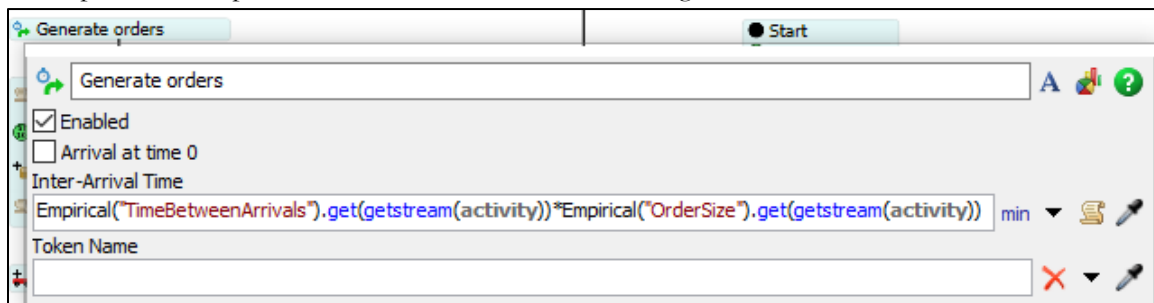
As shown in the figure below

- Name the activity *Generate orders*.

You do not need to type in the long statement for the Inter-Arrival Time.

- Use the dropdown menu to select **Statistical Distribution**, then *Empirical*. This provides an interface for entering **Distribution** and **Stream**.
- For the **Distribution**, use the dropdown menu to select **Empirical Distribution**, then *TimeBetweenArrivals*. This results in `Empirical("TimeBewteenArrivals").get(getstream(activity))`
- Use the **Cntl-C** keys to copy the expression.
- After the `)),` type in `*`
- After the `*`, use the **Cntl-V** keys to paste the expression.
- In the pasted expression, change *TimeBewteenArrivals* to *OrderSize*.

Once completed, the expression should be the same as in the figure below.

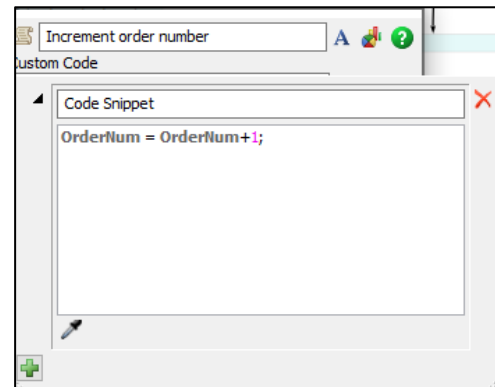


Custom Code

This activity will increment the **Global Variable** *OrderNum* each time an order arrives. As the name implies, this variable can be accessed (read or edited) anywhere in *FlexSim*, i.e., in 3D objects, Process Flow, and *FlexScript*.

The **Custom Code** activity is a bit of a misnomer. For many operations, you do not have to write code; you can just select from various options. In that regard, it is more of a “miscellaneous” activity. However, for this activity, you will need to write one line of *FlexScript* code.

- From the **Basic** pane, drag a **Custom Code** activity onto the workspace.
- Name the activity *Increment order number*.
- Use the **+** to the right of the **Custom Code** textbox to access the dropdown menu, then select *Code Snippet*.
- In the resulting textbox below **Code Snippet**, enter the following as shown in the figure to the right.
`OrderNum = OrderNum + 1;`



Assign Labels

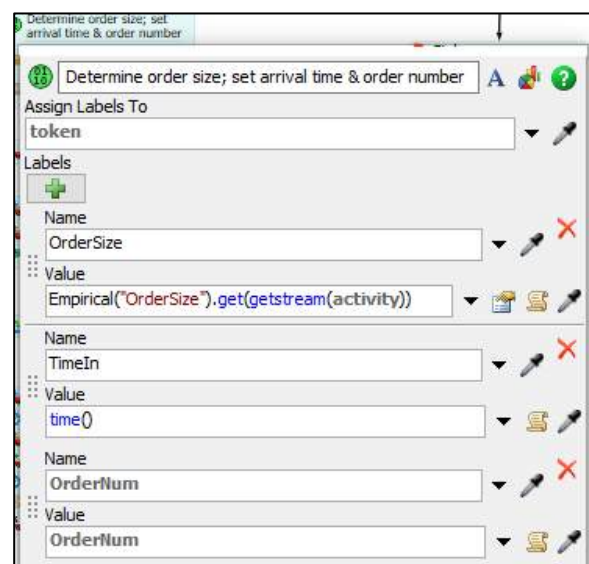
The next activity, **Assign Labels**, obtains information from various sources in the model and places that information on token labels. This activity stores three bits of information in the token's labels – order number, order size, and the current simulation time. The following describes defining the labels on the **Assign Labels** activity.

- Name the activity *Determine order size; set arrival time & order number*.

Each label is added by pressing the **+** button. All of the labels are shown in the figure to the right. Names are typed in, and values can either be typed in directly or entered, at least partially, by using the dropdown menu.

OrderSize's **Value** is obtained from a random sample from an **Empirical Distribution**.

- For **Value**, use the dropdown menu and select **Statistical Distribution**, then *Empirical*.
- In the resulting interface, for **Distribution**, use the dropdown menu to select **Empirical Distributions**, then *OrderSize*.



TimeIn's **Value** is the current simulation time.

- For **Value**, type in `time()`, a *FlexSim* command that gets the current time.

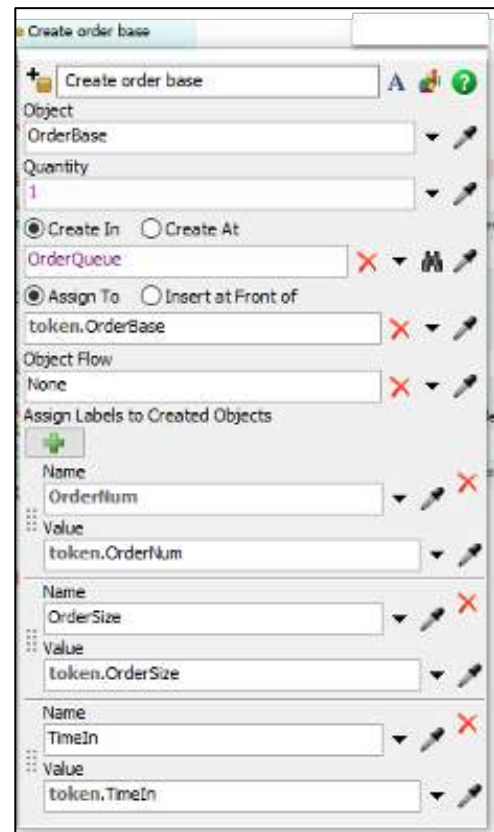
OrderNum's **Value** is the current order number that was just incremented in the previous activity.

- For **Value**, type in *OrderNum*, the current value of the **Global Variable OrderNum**.

Create Object

The **Create Object** activity generates orders into the model via the incoming order queue. The activity is shown in the figure to the right and defined below.

- Select the **Create Object** activity from the **Objects** section of the Process Flow Library.
- Name the activity *Create order base*.
- For the **Object** property, use the dropdown menu to select **Flowitems**, then *OrderBase*. This is the object being created, the order base flow item.
- Leave the default value of 1 for **Quantity**.
- For the **Create In** property, use the dropdown menu to select **Token Label**, then *OrderQueue*.
- For the **Assign To** property, type `token.OrderBase`. A reference to the created item is stored in this token label.
- Use the  button to add the three labels to the incoming order item (*OrderBase*).
For each label **Value**, use the dropdown menu to select **Token Label**, then *OrderNum*, *OrderSize*, and *TimeIn*, respectively.



Custom Code

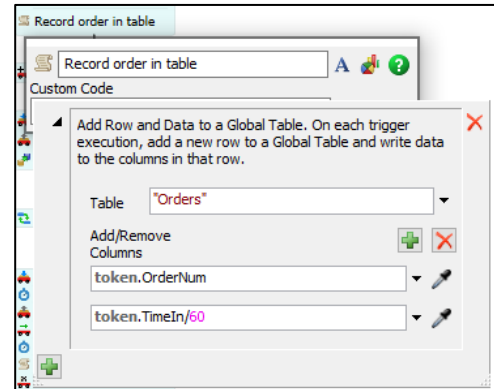
This activity adds a row to the Table Orders and populates values for two of the columns.

This is a case where the **Custom Code** activity is a bit of a misnomer in that you do not need to write code; you just select an operation from a variety of options. As mentioned earlier, in many cases, the **Custom Code** activity is more of a miscellaneous activity.

- From the **Basic** pane, drag a **Custom Code** activity onto the workspace.
- Name the activity *Record order in table*.
- Use the  to the right of the **Custom Code** textbox to access the dropdown menu, then select **Data** and then *Add Row and Data to Global Table*.

The resulting interface is shown in the figure to the right and its updates are described below.

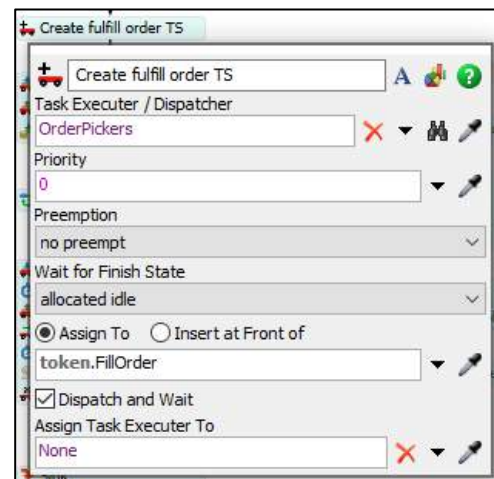
- For **Table**, use the dropdown menu to choose *Orders*.
- For **Add/Remove Columns**, use the  button to add:
 - *token.OrderNum* adds the value of the order number to the first column in the table.
 - *token.TimeIn/60* adds the time the order entered the system. Since the model units are minutes, the /60 converts the value to hours.



Create Task Sequence

Create a task sequence for the order-picking operator to process an order as described below and shown in the figure to the right. The operator processes one order at a time, and the tasks include moving orders from the incoming orders queue to the worktable, moving containers from racks to the order, moving fulfilled orders to an output sink, and entering information about the completed order.

- Select the **Create Task Sequence** activity from the **Task Sequences Assets** section of the Process Flow Library.
- Name the activity *Create fulfill order TS*.
- In the **Task Executer / Dispatcher** textbox, using the dropdown menu, select *Dispatcher* then *OrderPickers*. The task sequence is assigned to the **Dispatcher**.
- In the **Assign To** textbox, change to *token.FillOrder*. This is the name of the task sequence.
- For **Assign Task Executer To**, use the  button to delete the default value; this should result in the value *None*.



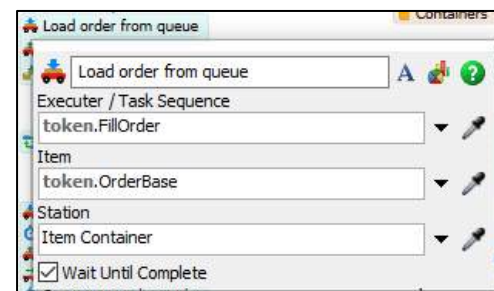
Load

After traveling to the order queue, the Picking Operator loads an order, which takes 0.05 minutes (3 seconds). The load time was defined previously as a property of the **Task Executer** object *OrderPicker_1*.

The **Load** activity includes traveling from the **Task Executer's** current location to where the item is located. Thus, a separate **Travel** activity is not needed.

The activity is defined below and shown in the figure to the right.

- Select the **Load** activity from the **Task Sequences** section of the Process Flow Library.
- Name the activity *Load order from queue*.
- Set the **Executer / Task Sequence** value to *token.FillOrder*.
- Set the **Item** value to *token.OrderBase*.



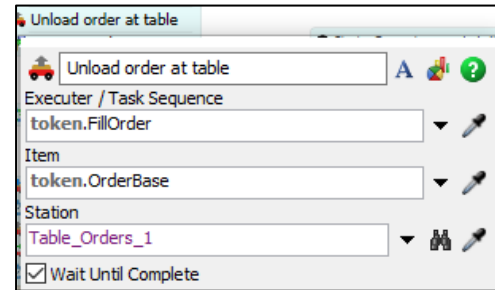
Unload

Once the Picking Operator reaches the worktable, the order is unloaded, which takes 0.05 minutes (3 seconds). Previously, the unload time was defined as a property of the **Task Executer** object *OrderPicker_1*.

Similar to the **Load** activity, **Unload** includes traveling from the **Task Executer's** current location, in the case where the order was loaded, to the unload location. Thus, a separate **Travel** activity is not needed.

The activity is defined below and shown in the figure to the right.

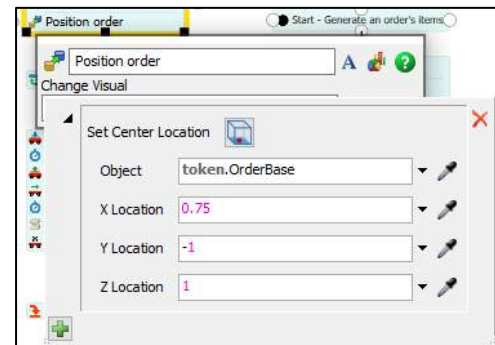
- Select the **Unload** activity from the **Task Sequences** section of the Process Flow Library.
- Name the activity *Unload order at table*.
- Set the **Executer / Task Sequence** value to *token.FillOrder*.
- Set the **Item** value to *token.OrderBase*. This specifies the item to be unloaded.
- Set the **Station** value to *Table_Orders_1*. This is where to unload the item.



Change Visual

The order will come into the table object on the floor, so it needs to be raised to table level. This activity is for aesthetics and does not affect system performance. Add the activity as shown in the figure to the right and described below.

- Select the **Change Visual** activity from the **Visual** section of the Process Flow Library.
- Name the activity *Position order*.
- Use the  button to the right of the **Change Visual** textbox, to select the *Set Location* option, then set the following values
 - **Object** to *token.OrderBase* either by typing it or using the dropdown menu and selecting **Labels** then *token.OrderBase*.
 - Set **X Location** to 0.75.
 - Set **Y Location** to -1.
 - Set **Z Location** to 1.



Run Sub Flow

The **Run Sub Flow** activity creates “children” tokens that loop through a set of activities. The activity defines where the looping starts, how many times the loop is executed, and instructions on how the loop operates.

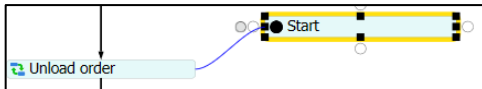
In this case, the sub flow generates the items for an order.

Add the activity described below and shown in the figure to the right.

- Select the **Run Sub Flow** activity from the **Sub Flow** section of the Process Flow Library.



- Select the **Start** activity from the **Sub Flow** section of the Process Flow Library and place it near and to the right of the Run Sub Flow activity, as shown in the figure below.



- Name the **Start** activity *Start – Generate an order's items*.

Define the **Run Sub Flow** activity properties described here and shown in the figure above.

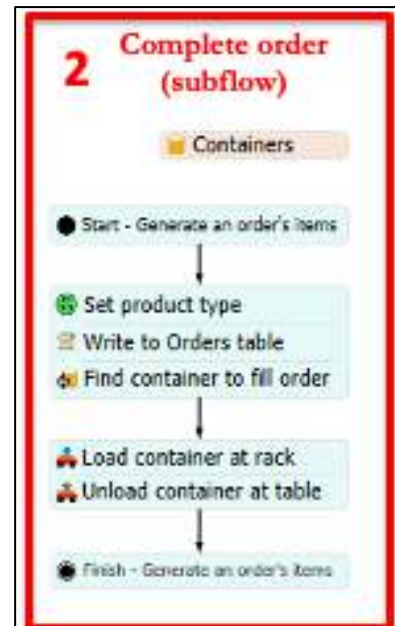
- Name the activity *Generate an order's items*.
- For **Destination**, use the sampler tool to select the **Start** activity added above. A blue line should now connect the two activities.
- For **Quantity**, use the dropdown menu to select the token label named *OrderSize*; thus, the value should be *token.OrderSize*.
- Check the box **Run Tokens One at a Time**
- Uncheck the box **Label Access to Parent Only**
- Check the box **Copy Labels to Children on Create**



If you haven't already done so, save the model. Recall that it is good practice to save often.

25.3.2 Order-Fulfillment Process Part 2 - Complete Orders

The second part of the Process Flow logic used to define the order-fulfillment process is shown in the figure to the right and discussed in detail below. This part deals with completing orders, i.e., gathering all the required containers specified in the order. This part of the process defines the sub flow associated with the **Run Sub Flow** activity that was defined at the end of the previous section. Also defined in that section was the Start activity that begins the sub flow.



Assign Labels

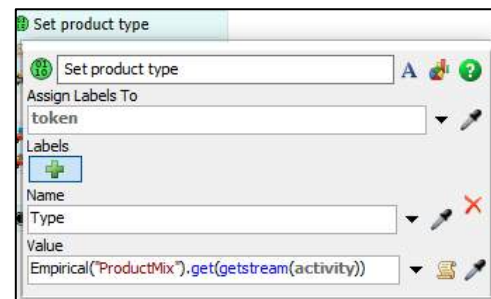
As shown in the figure to the right and as described below, this **Assign Labels** activity sets the product type for the order item.

- Name the activity *Set product type*.

Add the label by pressing the  button.

Type's Value is obtained from a random sample from an **Empirical Distribution**.

- For *Value*, use the dropdown menu and select **Statistical Distribution**, then *Empirical*.
- In the resulting interface, for **Distribution**, use the dropdown menu to select *Empirical Distributions*, then *ProductMix*.



Custom Code

This activity increments the number of containers in the order that is of the type determined in the previous activity. This is another case where the **Custom Code** activity is used to model a miscellaneous operation, in this case, writing to a table.

- From the **Basic** pane, drag an **Custom Code** activity onto the workspace.
- Name the activity *Record order in table*.
- Use the **+** to the right of the **Custom Code** textbox to access the dropdown menu, then select **Data** and then *Write to Global Table*.

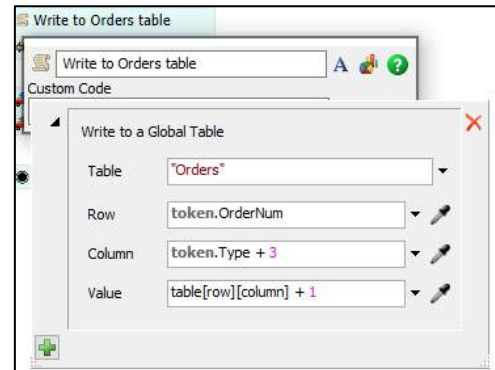
In the resulting interface, change the properties as shown in the figure to the right and described below.

- For **Table**, use the dropdown menu to choose **Table** and then *Orders*.
- For the **Row** value, use the dropdown menu to select **Labels**, then *token.OrderNum*.
- For **Column** value, use the dropdown menu to select **Labels**, then *token.Type*. Edit the value by adding +3.

This skips the first three columns in the table since they contain the order number, the time the order arrived, and a place to record the total time to fulfill an order. For example, if the container's **Type** value is 2, then the number of this type of container in this order is in column 5.

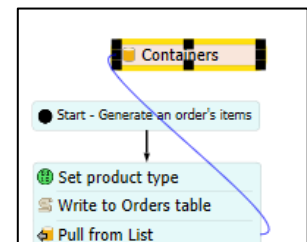
- For **Value**, use the dropdown menu to select *table[row][column] + 1*.

This increments the current value in a table cell by 1. That table cell is defined by the row and column specified in the previous two actions.



Recall that information on each container is pushed to a **List** when it enters a **Rack**, which indicates the container is available to fulfill an order. The **List** maintains information on all containers that are available to fill orders. When a container is needed to fulfill an order, the list is searched for a match, e.g. a container is of the same type as that requested by an order. If there is a match, its entry is “pulled” from the **List**, and the entry’s information is used in the Process Flow logic. If no match is found, the Process Flow token waits in this activity until there is a match between the order and an item in a **Rack**. All of this is accomplished through the **Pull from List** activity.

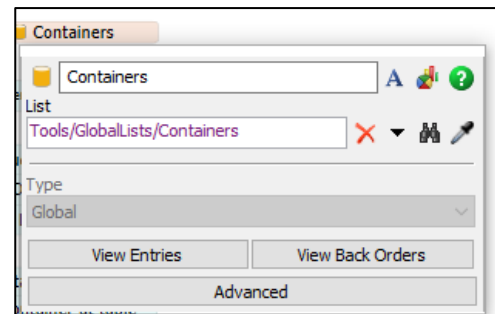
A **List** is a shared activity. In this case, the **List** is shared between three **Rack** objects and a Process Flow activity. Thus, it is referred to as a “shared resource”. In Process Flow, the **Pull from List** activity needs a reference to the correct **List**. This is specified in the **List** activity named *Containers*. Once the reference is made in the **Pull from List activity**, a blue line denotes the connection, as shown in the figure to the right.



List

Define the activity as shown in the figure to the right and as described below.

- From the **Shared Assets** pane, drag a **List** activity onto the workspace.
- Name the activity *Containers*.
- Use the dropdown menu to the right of the **List**'s textbox to select **Global List**, then *Containers*.



Pull from List

- From the **Shared Assets** pane, drag a **Pull from List** activity onto the workspace.

Define the activity as shown in the figure to the right and as described below.

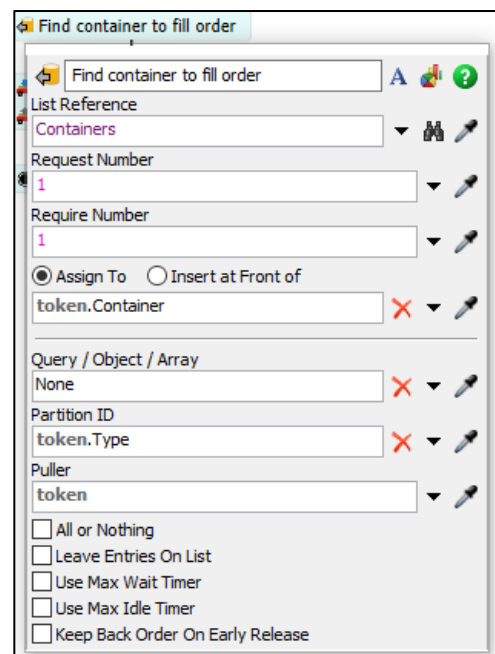
- Name the activity *Find containers to fill order*.
- For **List Reference**, use the dropdown menu to select **Lists** then *Containers*. This points to the **List** activity created above.

The reference to the item that pushed the information to the **List** that is currently in a **Rack** is assigned to a token label through the **Assign To** property.

- For **Assign To**, type in *token.Container* or use the dropdown menu and select **Token Label** then *Container*.

The **List** is partitioned, which means containers of like types are grouped together in a partition. Thus, an entry is pulled from one of the groups/partitions. In this case, an entry is pulled from the partition with the same value as the label *Type*.

- For **Partition ID**, type in *token.Type* or use the dropdown menu and select **Token Label**, then *Type*.



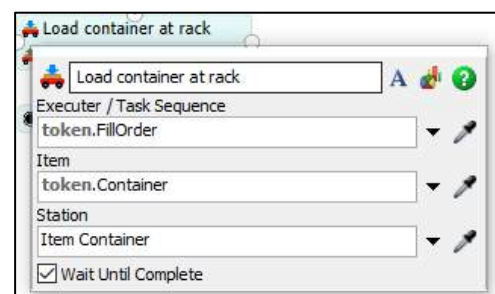
Once a container is available in a rack to fill an order, i.e., after it has been pulled from the list, the Order Picker travels to the container's rack, loads it, travels to the order on the worktable, and unloads it.

Load

After traveling to the container's rack, the Picking Operator loads a container, which takes 0.05 minutes (3 seconds). The load time was defined previously as a property of the **Task Executer** object *OrderPicker_1*.

As noted earlier, the **Load** activity includes traveling from the **Task Executer's** current location to where the item is located. Thus, a separate **Travel** activity is not needed.

The activity is defined below and shown in the figure to the right.



- Select the **Load** activity from the **Task Sequences** section of the Process Flow Library.
- Name the activity *Load container at rack*.
- Set the **Executer / Task Sequence** value to *token.FillOrder* by using the dropdown menu and selecting **Token Label** then *FillOrder*.
- Set the **Item** value to *token.Container* by using the dropdown menu and selecting **Token Label**, then *Container*.

Unload

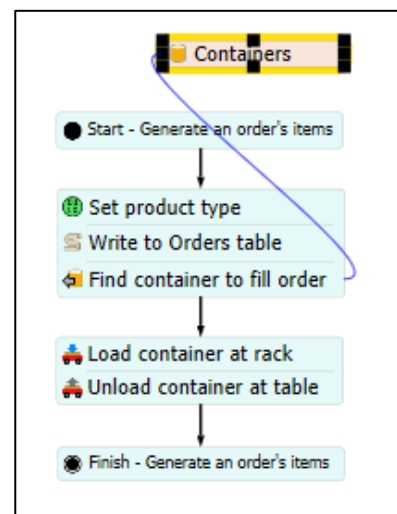
To unload a container, the Picking Operator must travel to the worktable. Once at the worktable, the order is unloaded, which takes 0.05 minutes (3 seconds). Like the load time, the unload time was defined as a property of the **Task Executer** object *OrderPicker_1*.

Similar to the **Load** activity, **Unload** includes traveling from the **Task Executer's** current location, in the case where the container was loaded, to the unload location. Thus, a separate **Travel** activity is not needed.

The activity is defined below and shown in the figure to the right.

- Select the **Unload** activity from the **Task Sequences** section of the Process Flow Library.
- Name the activity *Unload container at table*.
- Set the **Executer / Task Sequence** value to *token.FillOrder* by using the dropdown menu and selecting **Token Label** then *FillOrder*.
- Set the **Item** value to *token.OrderBase* by using the dropdown menu and selecting **Token Label** then *OrderBase*. This is what is unloaded.
- Set the **Station** value to *token,OrderBase* by using the dropdown menu and selecting **Token Label**, then *OrderBase*. This is where the item is unloaded.

The completed sub flow is shown in the figure to the right. Again, the breaking up of the activities is a matter of choice for clarity. All of the activities are executed sequentially, so they could be snapped together to form one block, or each could be separated using a connector or some combination.



If you haven't already done so, save the model. Recall that it is good practice to save often.

25.3.3 Order-Fulfillment Process Part 3 - Fulfill Orders

The third part of the Process Flow logic used to define the order-fulfillment process is shown in the figure to the right and discussed in detail below.

This part begins after the sub flow activities are completed and all required containers have been gathered and loaded onto the order item.

This part's activities include the Order Picker packing the containers and transporting them to the order-fulfillment area and updating the order information in the company's information system. Once complete, the operator is available to work on the next order in the queue or wait until the next order arrives. Also at this time, the **Orders Table** is updated with the time it took to complete the order.



Delay

Delay the token and the operator since this is part of a task sequence. The includes the length of time it takes the Picking Operator to pack the containers and complete the order. The time to do this is triangularly distributed with minimum, maximum, and most likely times of 0.75 minutes, 1.5 minutes, and 1.25 minutes, respectively.

The activity is defined as described below and shown in the figure to the right.

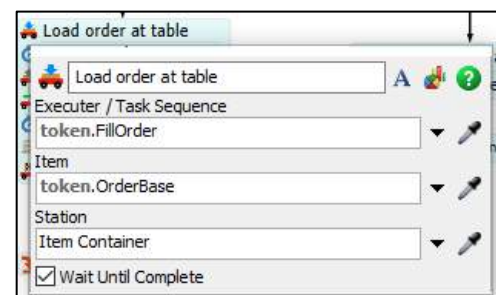
- Select the **Delay** activity from the **Visual** section of the Process Flow Library.
- Name the activity *Process order*.
- For the **Delay Time** property, use the dropdown menu to select **Statistical Distribution**, then *Triangular*.
- In the user interface for the triangular distribution, enter the property values for **Minimum**, **Maximum**, and **Mode** as 0.75, 1.5, and 1.25, respectively. The resultant **Delay Time** property value should be that shown in the figure.



Load

After the containers are packed and the order is complete, the Picking Operator loads the order, which takes 0.05. The activity is defined below and shown in the figure to the right.

- Select the **Load** activity from the **Task Sequences** section of the Process Flow Library.
- Name the activity *Load order at table*.
- Set the **Executer / Task Sequence** value to *token.FillOrder* by using the dropdown menu and selecting **Token Label** then *FillOrder*.
- Set the **Item** value to *token.Container* by using the dropdown menu and selecting **Token Label**, then *Container*.



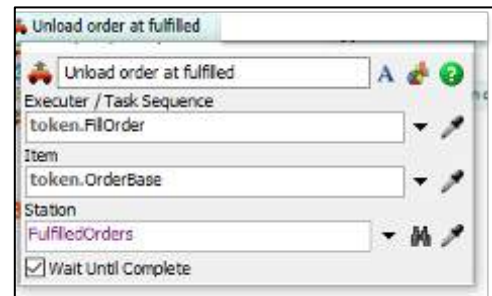
Unload

To dispose of the order, i.e., unload it, the Picking Operator must travel to the **Sink** named *FulfilledOrders*. Once at the **Sink**, the order is unloaded, which takes 0.05 minutes.

Recall the **Unload** activity includes traveling to the destination.

The activity is defined below and shown in the figure to the right.

- Select the **Unload** activity from the **Task Sequences** section of the Process Flow Library.
- Name the activity *Unload order at fulfilled*.
- Set the **Executor / Task Sequence** value to *token.FillOrder* by using the dropdown menu and selecting **Token Label** then *FillOrder*.
- Set the **Item** value to *token.OrderBase* by using the dropdown menu and selecting **Token Label** then *OrderBase*.
- Set the **Station** value by using the sampler (eyedropper icon) to select the **Sink** *FulfilledOrders* in the 3D view. The resulting property value should show *FulfilledOrders*.



Travel

The **Travel to Object** activity sends the Picking Operator to the **Dispatcher** object after unloading the fulfilled order where the operator will enter information about the completed order.

Add the activity as described below and shown in the figure to the right.

- Select the **Travel To Object** activity from the **Task Sequences Assets** section of the Process Flow Library.
- Name the activity *Travel to dispatcher*.
- Set the **Executor / Task Sequence** value to *token.FillOrder* by using the dropdown menu and selecting **Token Label** then *FillOrder*.
- Set the **Destination** value by using the sampler (eyedropper icon) to select the **Dispatcher** *OrderPickers* in the 3D view. The resulting property value should show *OrderPickers*.

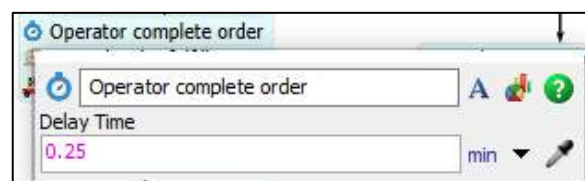


Delay

Delay the token and the operator since this is part of a task sequence, the length of time it takes the Picking Operator to enter order information into the company's system. The time to do this is assumed to be a constant 0.25 minutes (15 seconds).

The activity is defined as described below and shown in the figure to the right.

- Select the **Delay** activity from the **Visual** section of the Process Flow Library.
- Name the activity *Operator completes order*.
- For the **Delay Time** property, type in the value 0.25.



Custom Code

This activity records the time it took to process the completed order, which is from when it arrived until when it was fulfilled. This is another case where the **Custom Code** activity is used to model a miscellaneous operation, in this case, writing to a table.

- From the **Basic** pane, drag an **Custom Code** activity onto the workspace.
- Name the activity *Record order fulfill time*.
- Use the  to the right of the **Custom Code** textbox to access the dropdown menu, then select **Data** and then *Write to Global Table*.

In the resulting interface, change the properties as shown in the figure to the right and described below.

- For **Table**, use the dropdown menu to choose **Global Table** and then *Orders*.
- For **Row** value, use the dropdown menu to select **Labels**, then *token.OrderNum*. The row number is the value of the Order Number.
- For **Column** value, type in “TimeToFulfill,” which is the name of the column that is to be updated.
- For **Value**, type in the following expression
 $\text{time}() - \text{token.TimeIn}$

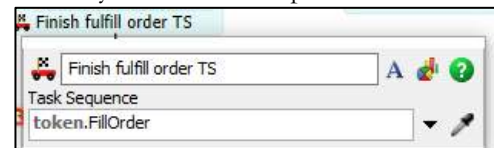


The time to complete an order is the current simulation time, obtained by the *FlexSim* command named *time()*, minus the time the order arrived.

Finish Task Sequence

This activity defines the end of the task sequence for the Picking Operator to fulfill an order. After this activity, the Picking Operator is available to do other tasks or be set to idle and await the next task.

- From the **Task Sequences** pane, drag a **Finish Task Sequence** activity onto the workspace.
- As shown in the figure to the right, for the property **Task Sequence**, use the dropdown menu to select **Token Label**, then *FillOrder*.

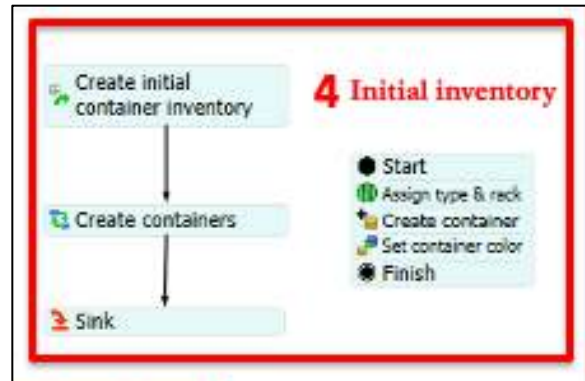


If you haven't already done so, save the model. Recall that it is good practice to save often.

25.3.4 Order-Fulfillment Process Part 4 – Initial Inventory Orders

The final part of the Process Flow logic creates an initial inventory of containers in the Warehouse Area. The activities and flow are shown in the figure to the right and discussed in detail below.

The amount of inventory could be a random variable or a fixed value. In this case, a fixed value is used and is specified as a **Model Parameter** in the *GeneralParameters* table.



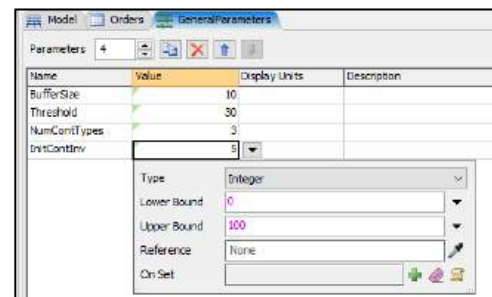
The logic is quite simple with a sub-flow activity creating and placing the appropriate number of containers into the model at time 0. As each initial container is created, its type is randomly assigned, its color is designated based on its type, and it is placed in the appropriate **Rack**.

Each initial container is empty; i.e., it contains no components. The initial containers are created only to assess their impact on the Warehouse Area; therefore, the added detail of the container contents is not needed. Of course, the contents could be generated when a container is created, but that added complexity is not required.

Model Parameter

The number of containers in the initial inventory is defined in a **Model Parameters Table**, as shown in the figure to the right and explained below.

- Add a fourth parameter in the *GeneralParameters* **Model Parameters Table** and name it *InitContInv*.
- Use the dropdown menu to set:
 - **Type** to *Integer*.
 - **Lower Bound** to *0*.
 - **Upper Bound** to *100*.
- Set the current value to *5*.

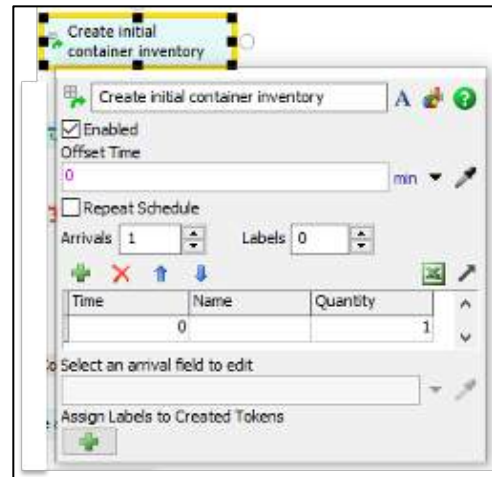


Thus, the initial inventory could be as many as 100 or as few as 0.

Schedule Source

The **Schedule Source** activity initiates the initial inventory process just before the start of the simulation. The activity is shown in the figure to the right and defined below.

- Select the **Schedule Source** activity from the **Token Creation** section of the Process Flow Library.
- Name the activity *Create initial container inventory*.



Run Sub Flow

As discussed before, the **Run Sub Flow** activity creates “children” tokens that loop through a set of activities that are between the **Start** and **Finish** activities that define the sub flow.

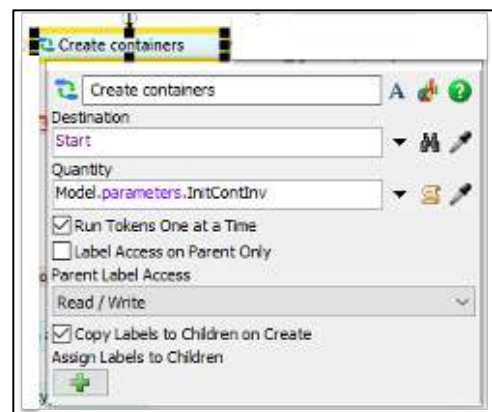
The **Run Sub Flow** activity defines where the looping starts, how many times the loop is executed, and instructions on how the loop operates. In this case, the sub flow generates initial containers in racks.

- Select the **Run Sub Flow** activity from the **Sub Flow** section of the Process Flow Library.
- Select the **Start** activity from the **Sub Flow** section of the Process Flow Library and place it near and to the right of the Run Sub Flow activity, as shown in the figure below.

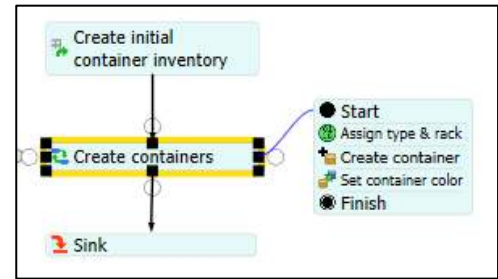


Define the **Run Sub Flow** activity properties as described below and shown in the figure to the right.

- Name the activity *Create containers*.
- For **Destination**, use the sampler tool to select the **Start** activity added above. A blue line should now connect the two activities.
- For **Quantity**, use the dropdown menu to select **Parameter**, then **GeneralParameters** then *InitContInv*. The result value is `Model.parameters.InitContInv`, which is the value of the **Model Parameter** named *InitContInv* that was defined earlier in this section.
- Check the box **Run Tokens One at a Time**
- Uncheck the box **Label Access to Parent Only**
- Check the box **Copy Labels to Children on Create**



As shown in the figure to the right, the main logic flow concludes with a **Sink** activity. The remainder of this section defines the sub-flow activities as shown in the figure to the right and defined below.



Assign Labels

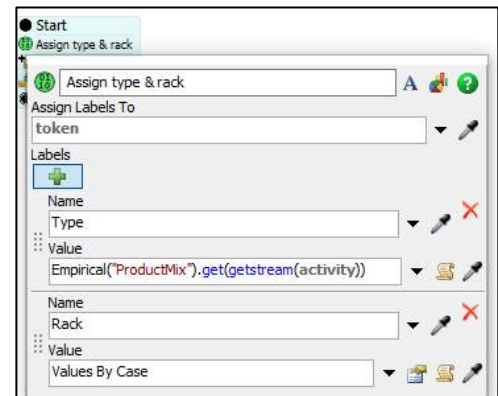
As shown in the figure to the right and as described below, this **Assign Labels** activity is used to set the product type for the container and where the container should be placed in the model, i.e., which **Rack**.

- Name the activity *Assign type & rack*.

Add each label by pressing the button.

Type's **Value** is obtained from a random sample from an **Empirical Distribution**.

- For **Value**, use the dropdown menu and select **Statistical Distribution**, then *Empirical*.
- In the resulting interface, for **Distribution**, use the dropdown menu to select **Empirical Distributions**, then *ProductMix*.



Rack's **Value** uses the **Values By Case** construct since the choice of which **Rack** the container is created in depends on its **Type** value.

- For **Value**, use the dropdown menu and select **Values By Case**.

In the resulting interface,

- For **Case Function**, use the dropdown menu to select **Labels** then *token.Type*.

Use the button to add each of the following cases.

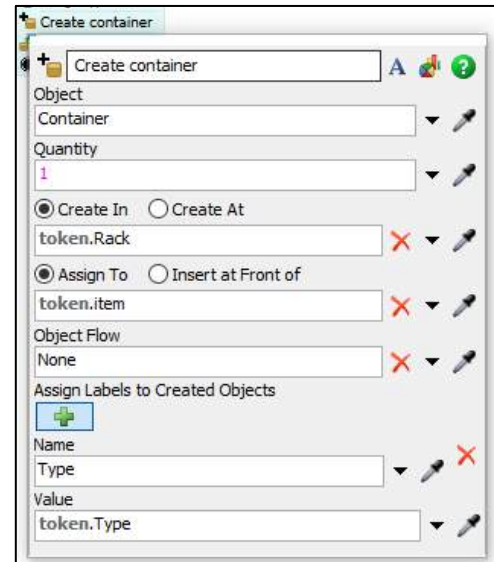
For each case's **Value**, use either (1) the dropdown menu to select **Object** then *Model.find("name")* where name is the name of the appropriate rack queue, e.g. *RackQueue_1* or (2) use the sampler (eyedropper) to select the appropriate rack queue in the 3D view.

- Case 1's **Value** should be *Model.find("RackQueue_1")*
- Case 2's **Value** should be *Model.find("RackQueue_2")*
- Case 3's **Value** should be *Model.find("RackQueue_3")*

Create Object

The **Create Object** activity creates a container and places it in the appropriate **Rack**; actually, it is placed in the **Queue** before each **Rack**, but the item will move immediately to the **Rack** unless the initial number of containers exceeds its rack's capacity. The activity is shown in the figure to the right and defined below.

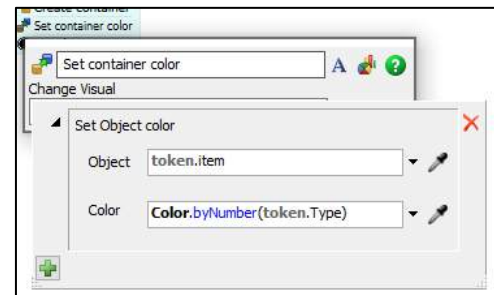
- Select the **Create Object** activity from the **Objects** section of the Process Flow Library.
- Name the activity *Create container*.
- For the **Object** property, use the dropdown menu to select **Flowitems**, then *Container*.
- Leave the default value of 1 for **Quantity**.
- For the **Create In** property, use the dropdown menu to select **Token Label**, then *Rack*.
- Use the  button to add one label to the container item (Container). The label's Name is *Type* and its label **Value** is selected from the dropdown menu by selecting **Token Label**, then *Type*.



Change Visual

A **Change Visual** activity is used to change the color of the container based on its **Type** value. Add the activity as shown in the figure to the right and described below.

- Select the **Change Visual** activity from the **Visual** section of the Process Flow Library.
- Name the activity *Set container color*.
- In the **Change Visual** textbox, using the dropdown menu, select *Set Object color*, then set the following values
 - **Object** to the *token.item* either by typing it or using the dropdown menu and selecting **Labels**, then *token.item*.
 - For the **Color** property, use the dropdown menu and select **Color**, then *Color.byNumber(1)*.
 - Edit the expression by changing the 1 to *token.Type*.



The container's color is based on its Type.

Sink

The definition of the logic flow is complete.

- Select the **Sink** activity from the **Basic** section of the Process Flow Library.

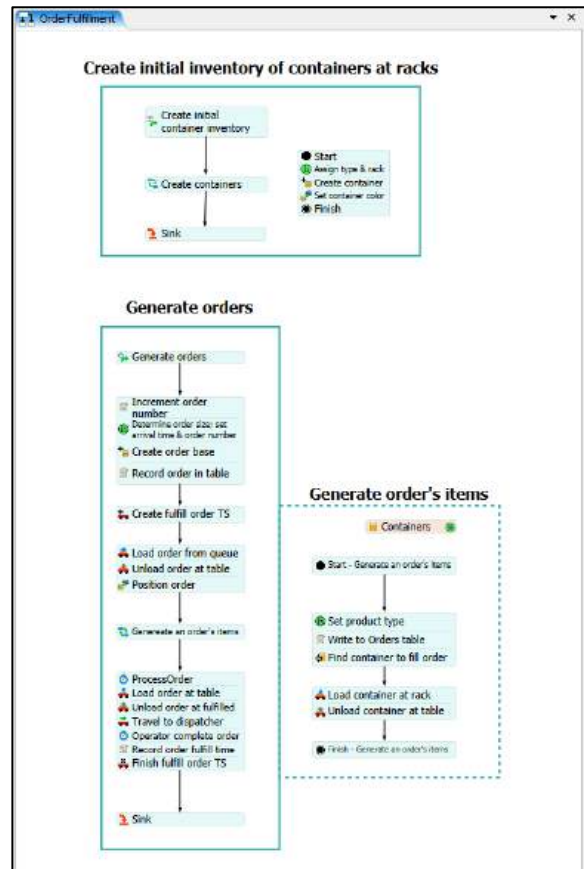


Text

Use the **Container** activity in the *Display* section to label the parts of the logic as shown in the figure to the right.

- Drag out three **Container**, then label and size each as shown in the figure.

This does not affect the model logic or processing but makes the flow more readable.



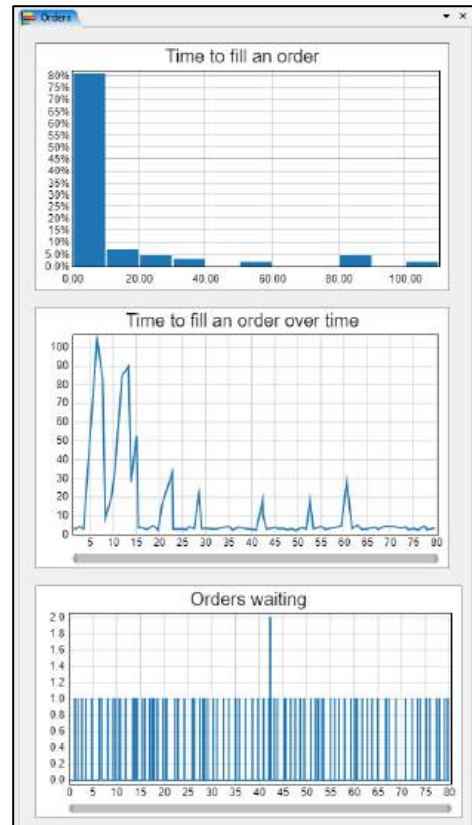
25.3.5 Define and chart performance indicators for the order-fulfillment process

The system is being designed to be responsive to customer demands. That is, the orders should be fulfilled as soon as possible, of course, within reason. Charts and graphs are usually a beneficial means to assess performance, especially over time. The ones used in this example are shown in the figure to the right.

For orders, a good measure is how long it takes to fulfill an order (from the time an order arrives to the time it is completed) and how many orders are waiting at any time. Therefore, a **Dashboard** of charts is created that shows this information. The time to fulfill an order is displayed both as a histogram, to show the distribution of times, and over time to see if there are times when orders back up.. The charts in the figure to the right are from a 4800-minute (80-hour) run of the simulation model.

Since the time to fulfill an order is a key performance indicator, it is defined in a **Performance Measure Table** so that it can be used to evaluate alternative scenarios in the **Experimenter**.

The following provides step-by-step instructions for creating the plots.



- Use the **Dashboards** button on the **Main Menu** or use the  button in the **Toolbox** and then select **Dashboard** to create a new **Dashboard**.
- Name the **Dashboard** *Orders*.

Histogram

This chart provides a histogram of how long it takes to fill orders.

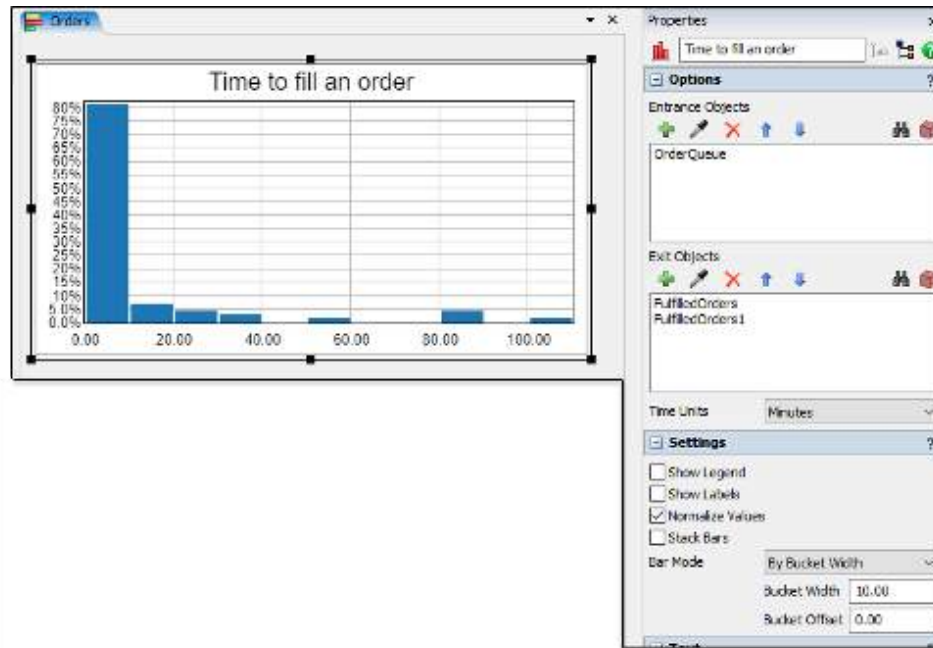
- From the **Dashboard Library** and **Staytime** pane, select the **Staytime** chart type, then *Histogram*.

Create the chart shown in the figure below based on the following description.

- Name the chart *Time to fill an order*.
- In the **Options** pane, use the  button for the **Entrance Objects** property to select **Select Object**, then *Queue*, then *OrderQueue*.
- In the **Options** pane, use the  button for the **Exit Objects** property to select **Select Object**, then *Sink*, then *FulfilledOrders*.
Thus, *Flexsim* measures the time each item spends in these objects and all objects in between.
- In the **Settings** pane, check **Normalize Values**. This converts the y-axis value from the count of occurrences to the percentage of occurrences.
- Also, in the **Settings** pane, for **Bar Mode**

- Select *By Bucket Width*
- Set the **Bucket Width** value to *10.00*

This property defines how many bars the chart has and their width. In this case, each bar is 10 units wide, and there will be as many bars as needed to cover the data.



For the 4800-minute simulation, 80% of the orders are filled within 10 minutes. However, a few took over an hour. Most likely the longer durations are due to a temporary shortage of a specific type of container and/or the absence of a Picking Operator when they were on break or at lunch.

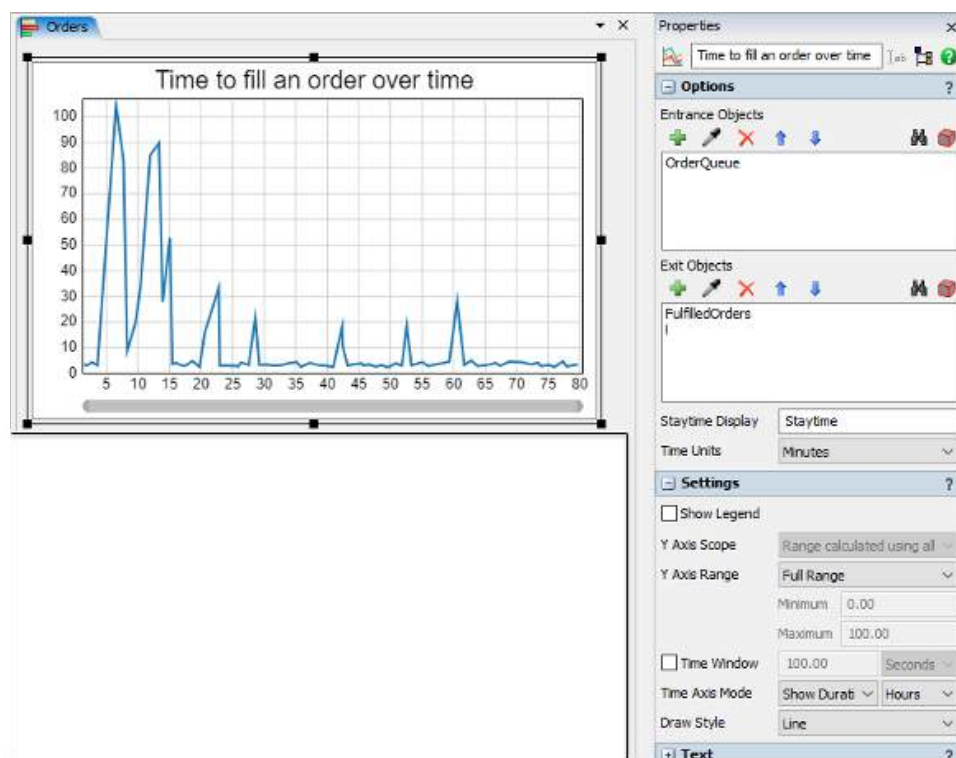
Line chart

This chart also considers how long it takes to fill orders, but it looks at the process's order fill time over time, which helps identify when the long durations occur.

- From the **Dashboard Library** and **Staytime** pane, select the *Staytime* chart type, then *Line Chart*.

Create the chart shown in the figure below and the following description.

- Name the chart *Time to fill an order over time*.
- In the **Options** pane, use the  button for the **Entrance Objects** property to select *Select Object*, then *Queue*, then *OrderQueue*.
- In the **Options** pane, use the  button for the **Exit Objects** property to select *Select Object*, then *Sink*, then *FulfilledOrders*.
- In the **Settings** pane,
 - For **Time Axis Mode**, change to *Show Duration* and *Hours*.
 - For **Draw Style Width**, set to *Line*.



This chart shows when the longer durations occurred. Most were at the beginning, likely due to a temporary shortage of a specific type of container.

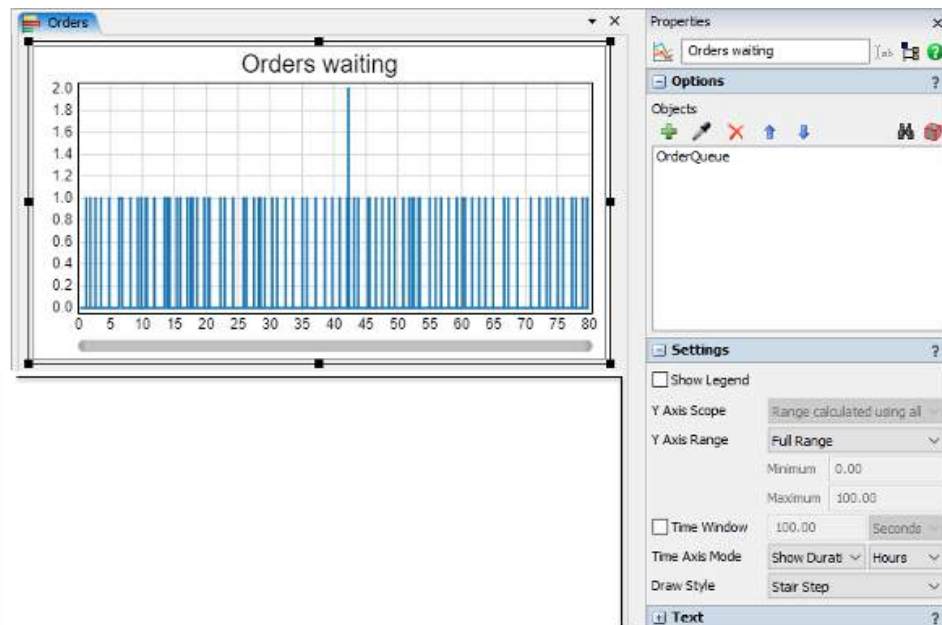
Content chart

This chart shows the number of orders waiting over time.

- From the **Dashboard Library** and **Content** pane, select the **Content** chart type, then *Line Chart*.

Create the chart shown in the figure below and the following description.

- Name the chart *Orders waiting*.
- In the **Options** pane, use the  button for the **Objects** property to select **Select Object**, then *Queue*, then *OrderQueue*.
- In the **Settings** pane,
 - For **Time Axis Mode**, change to *Show Duration* and *Hours*.
 - For **Draw Style Width**, set it to *Stair Step*.



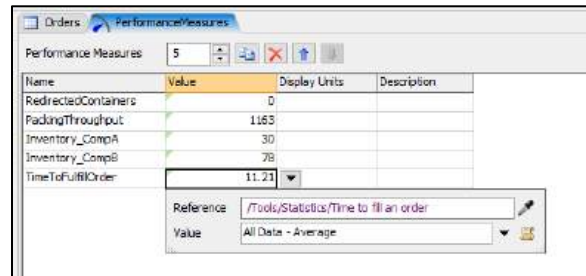
This chart shows that during this 4800-minute simulation, there is no problem with the number of orders waiting to be processed.

Performance Measure

A key performance indicator for this portion of the model is the time to fulfill an order. A summary measure of the indicator is the average or mean time to fulfill an order. Therefore, that value is tracked in the **Performance Measure Table**. As stated earlier, this is likely to be an indicator to consider when evaluating alternative designs in the **Experimenter**.

As shown in the figure to the right and described below, add a performance measure for the time an order takes to complete.

- From the **Toolbox**, open the **Performance Measure Table** named *Performance Measures*.
- Increase the number of **Performance Measures** to 5.
- Name the measure *TimeToFulfillOrder*.
- Use the dropdown menu to define the **Value**.



Name	Value	Display Units	Description
RedirectedContainers	0		
PackingThroughput	1165		
Inventory_CompA	30		
Inventory_CompB	78		
TimeToFulfillOrder	11.21		

Reference: /Tools/Statistics/Time to fill an order
Value: All Data - Average

In the **Reference** textbox, use the sampler (eyedropper) to click on the histogram of *Time to fill an order* on the **Dashboard** named *Orders*, then select **All Data**, then **All Data – Average**. This results in what is shown in the **Reference** and **Value** properties.

Note that the value for *TimeToFulfillOrder* in the figure is 11.21. This means the average time to fulfill an order in the 4800-minute simulation is 11.21 minutes. While averages are helpful to summarize data, they can also be a bit deceiving. While the average is 11.21 minutes, we saw earlier that 80% of the order fulfillment times were below 10 minutes, and several were about an hour or more. Therefore, it is oftentimes helpful to consider data in several ways.



If you haven't already done so, save the model. Recall that it is good practice to save often.



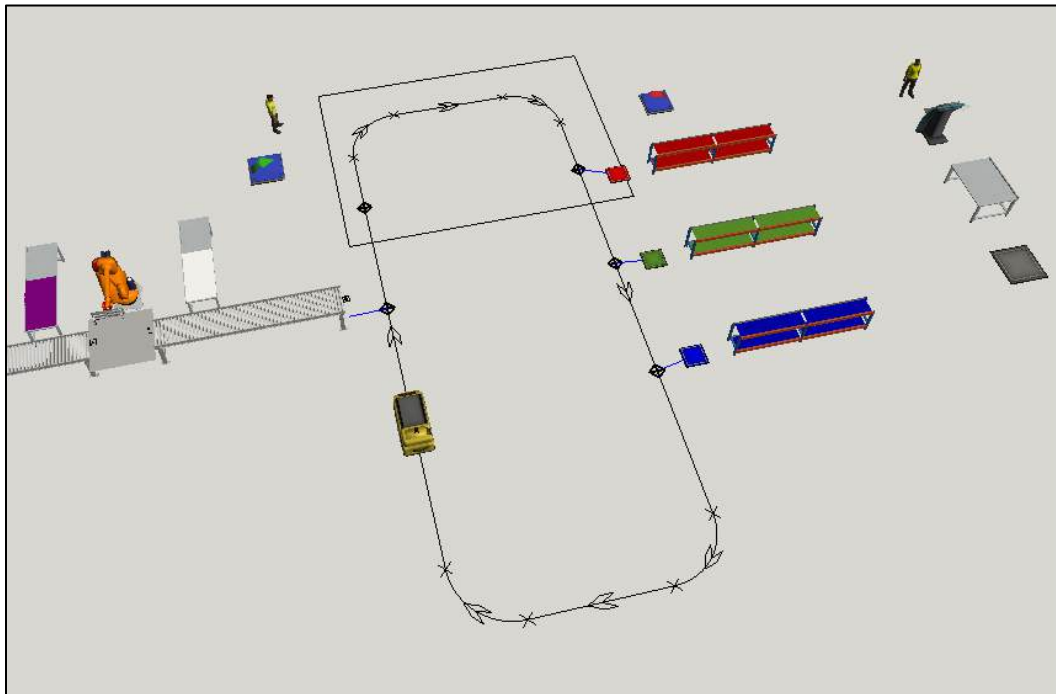
Use the **Save Model As** option in the **File** menu to make a copy of the existing model to be further customized in the next section. Again, you can use any file name, but the following model is referred to as *Primer_20* in the primer.

26 AGV TRANSPORT BETWEEN PACKING AND WAREHOUSE

Chapter 26 implements an AGV system to transport containers from the Packing Area to the Warehousing Area. A Control Area is added to the AGV network to restrict traffic in the area to one task executor. A chart is added to track the AGV's utilization.

This final chapter of the primer model adds a simple AGV (Automated Guided Vehicle) system to transport containers from the Conveyor after the Packing Area to each Rack in the Warehouse Area.

The AGV capabilities in *FlexSim* are extensive; however, in keeping with the theme of this primer, only the basics are considered in the example. An overview of the example AGV system is shown in the figure below. The chapter explains the addition of a transportation network in the model.



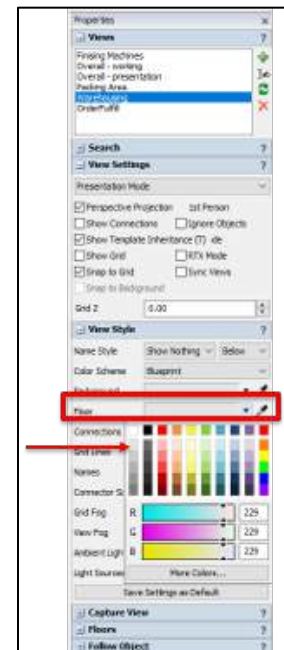
The **AGV** object, a **Task Executor**, picks up a container when it reaches the end of the Packing Area **Conveyor** and transports it to one of the **Queues** in front of the **Racks**, depending on the container type. If there are no containers to transport, the **AGV** returns to a “home” location near the conveyor pick-up point. The AGV travels in a single direction - clockwise - around the network.

Since the Picking Operator must traverse near the AGV path for breaks and lunches, a **Control Area** is used to avoid collisions. No more than one **Task Executor** object (AGV and Operator) can be in the area at any time. Since the Picking Operator traverses the area infrequently, a simple test case is used to verify the zone is working correctly.

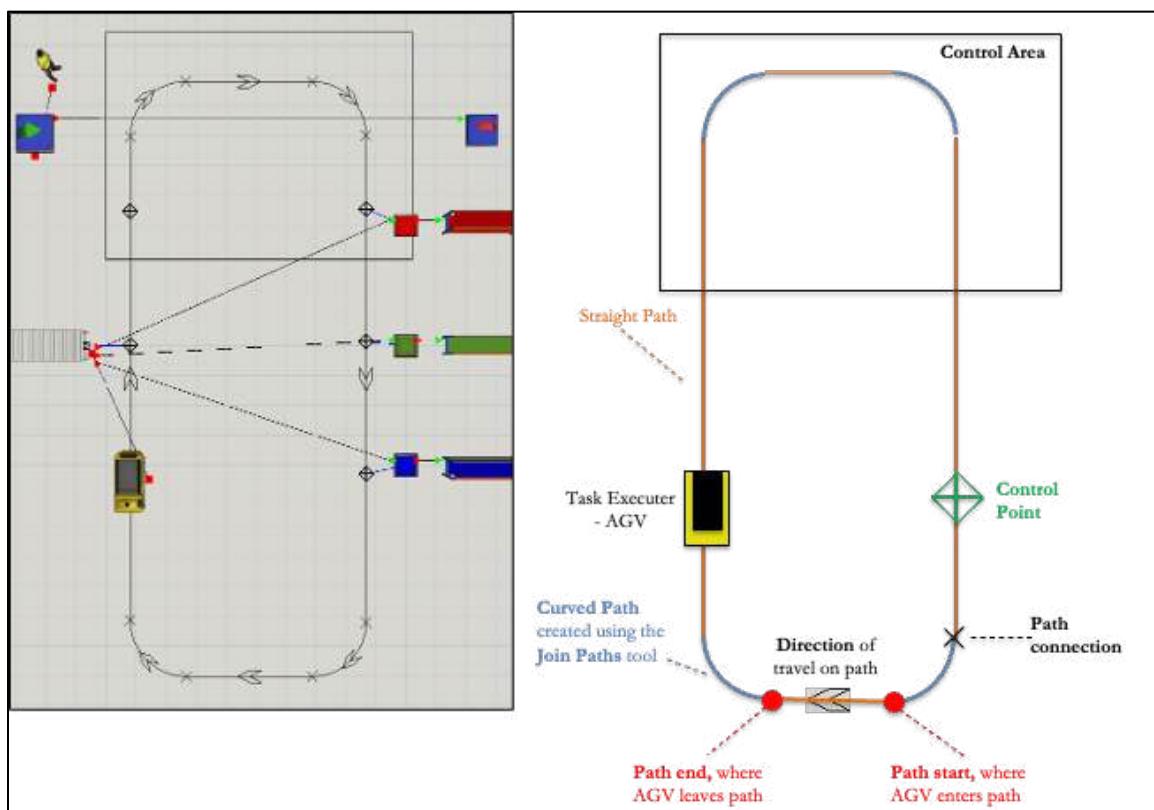
The base model for the additions described in this chapter is **Primer_19** that was saved at the end of Chapter 25. However, a copy of that file was saved as **Primer_20**; thus, we begin with that file.

The background or floor on the previous model was a bit dark, so it is lightened, as shown in the screenshot to the right. To change the floor color:

- Click anywhere on the modeling surface, then open the **View Settings** pane in the **Properties** window.
- Use the dropdown menu for **Floor**, highlighted by the red box in the figure to the right, and select a light gray color, as indicated by the red arrow in the figure to the right.



The following figure identifies the AGV system's basic components.



An AGV system is a network of interconnected paths containing various decision-making capabilities.

- A network is composed of both **Straight Path** and **Curved Path** objects. In this example, considering the simple overall rectangular shape of the network, only **Straight Paths** are used.
The curved paths in the model are automatically created by the **Join Path** tool in the **AGV** section of the **Object Library**. (Technically, the **Join Path** is an object, but since it acts like a tool, that's how I refer to it.)

- AGV paths can be either uni- or bi-directional. In this example, they are uni-directional.
- Paths can force a single AGV orientation regardless of the AGV's direction of travel; i.e., paths can control which way an AGV must face. This is often used when an AGV uses a single path for backing into or out of an area; however, this feature is not considered in this example.
- Paths have a starting point, where an AGV enters the path, and an endpoint, where an AGV exits a path.

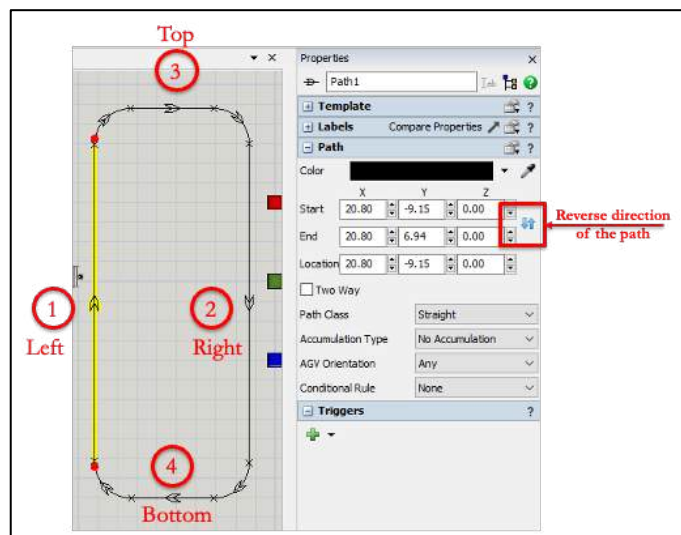
The **AGV** object is a type of **Task Executer** and thus has all of those properties.

Decision making in an AGV network is carried out through **Control Points**. These points control an AGV's entrance to a part of the network, where items are picked up or dropped off, where an AGV looks for work, etc. **Control Points** act as allocation/deallocation points where AGVs look ahead to the next control point and allocate that control point before moving to that point.

A **Control Area** object enforces mutual exclusion on one or more paths in a network. It can be integrated with an A* network to restrict the number of travelers in an area.

26.1 AGV path

The following describes how to lay out the AGV network, as shown in the figure to the right. The numbers in the red circles on the screenshot are the suggested order for adding the paths to the model; the names provide a reference to the paths based on orientation. Also shown is the button in the **Properties** window used to change the direction of a path. It is highlighted by the red box in the **Properties** window to the right..



Create the first path on the left side of the network:

- From the **AGV** section of the Object Library, select a **Straight Path**, then click on the 3D View near where the AGV will enter the path; in this case, it is toward the bottom of the layout. Then, drag the top part and click near its location; this is where an AGV will exit the path. This is an approximate size and location since the values can be fine-tuned on its **Properties** window, as shown in the figure above.
- Name the **Path** *AGV_Path_Left*.

Create the second or right side of the network.

- Use Cntl-C and Cntl-V to copy and paste the first **Straight Path** and locate it near the Rack **Queues**. Align the start end with the left-side path so the two paths are parallel.
- Press the reverse direction button so the path's flow is from top to bottom.
- Name the **Path** *AGV_Path_Right*.

Create the third or top portion of the network.

- Drag out another **Straight Path** from the Object Library. Place the start end near the end of the left side of the network, but not touching, then extend to near the queues in front of the racks.
- Name the **Path** *AGV_Path_Top*.

Create the final or bottom side of the network.

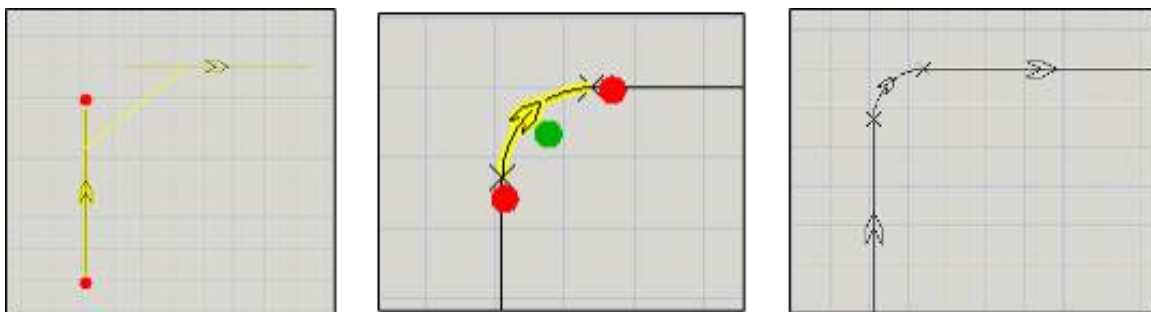
- Use Cntl-C and Cntl-V to copy and paste the third **Straight Path** and locate it near the start of the first path. Align the start end with the top-side path so the two paths are aligned and parallel.
- Press the reverse direction button so the path's flow is from top to bottom.
- Name the **Path** *AGV_Path_Bottom*.

Join the path segments together to form a closed network.

- As shown in the figure to the left below, select the **Join Paths** object in the Object Library (the cursor's shape changes to a green arc), then click on **Path** *AGV_Path_Left* so that it is highlighted (yellow) and then click on **Path** *AGV_Path_Top* so that it is also highlighted (yellow). The two segments will now be joined, as shown in the center figure and the one to the right.

The center figure below shows the curved segment created by the Join Paths object when selected. Note the start and end of the path segment (red circles), the direction of travel, and the connections to the other straight paths.

The figure to the right shows part of the closed-loop network. Note that an **X**-like symbol denotes where the path segments are connected.

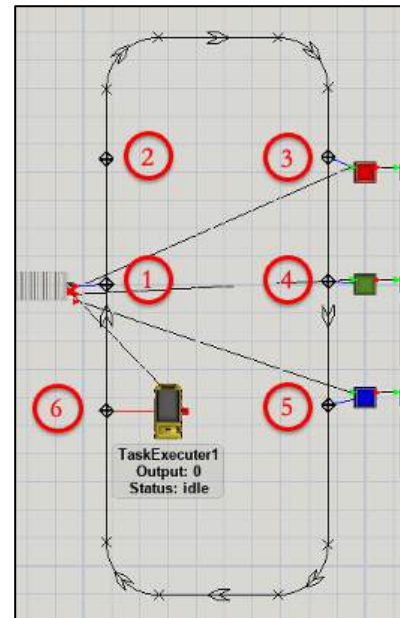


- Continue to click on the network paths, **Path AGV_Path_Right**, then **Path AGV_Path_Bottom** to complete the loop.
This is all done while the cursor's shape is the green arc. If the cursor reverts to the basic shape, reselect the **Join Paths** object as in the above step.
To get out of the Join-Paths mode (green arch shape) and return to the basic arrowhead cursor, press the ESC key.

26.2 Control Points

Control Points enable the AGV to perform tasks and interact with the other objects in the model. As shown in the figure to the right, six **Control Points** are used in this model.

- Drag out six of the diamond-shaped **Control Points** onto the modeling surface.
Note that the shape is a plain diamond when it is not connected to the network.
- Select and position each **Control Point** on the network and in the locations shown in the figure.
Note that when a **Control Point** is a part of the network, the diamond shape contains a plus-sign-like cross.



The following describes what needs to be done at each point.

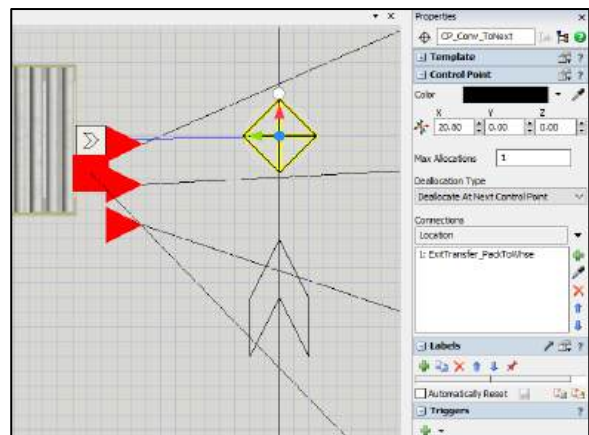
Control Point 1

The figure to the right shows the general properties of a **Control Point** and the specifics for the one that interacts with the conveyor from the Packing Area.

- Name the object *CP_Conv_ToNext*.
- Make an A-Connection from the **Control Point** to the **Exit Transfer** at the end of the **Conveyor** from the Packing Area named *Transfer_PackToWhse*.

Note the connection shows up in the **Connections** section of the interface.

- Do not be concerned if the *x* and *y* locations do not match the ones shown in the figure. The *x*-location is determined by the path's location, and your layout may not be exactly the same as the example. In this case, having the **Control Point** in front of the **Conveyor** is best.



Control Point 2

This **Control Point** is used for the **Control Area**, which will be discussed in a subsequent section.

- Name the object *CP_ControlArea_1*.

The next three **Control Points** are associated with the **Queues** in front of the **Racks** in the Warehousing Area.

Control Point 3

- Name the object *CP_Type_1*.
- Make an A-Connection from the **Control Point** to the **Queue** named *RackQueue_1*. Again, the connection shows up in the **Connections** section of the interface.

Control Point 4

- Name the object *CP_Type_2*.
- Make an A-Connection from the **Control Point** to the **Queue** named *RackQueue_2*.

Control Point 5

- Name the object *CP_Type_3*.
- Make an A-Connection from the **Control Point** to the **Queue** named *RackQueue_3*.

Control Point 6

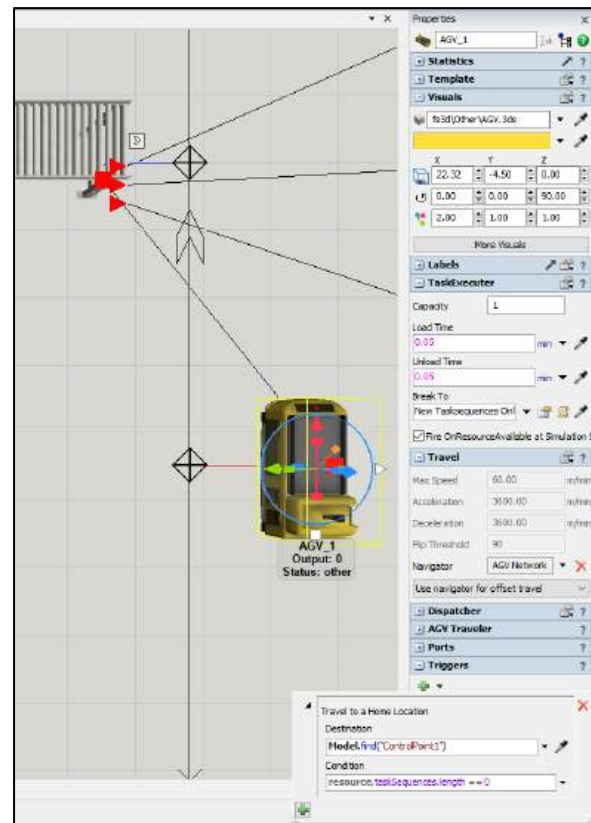
The last Control Point is the home location for the AGV, i.e., where it goes when it has no tasks to perform.

- Name the object *CP_AGV_Home*.
- Since the **Control Point** is renamed, update its name in the **AGV's On Resource Available** trigger. Change the value of the **Home Location** or **Destination** to *Model.find("CP_AGV_Home")*

26.3 AGV

The **AGV** object transports containers, one at a time, between the Packing and Warehousing Areas. The AGV is integrated into the model as described below and as shown in the figure to the right.

- Drag out an **AGV** object, which is in a dropdown section of the **Task Executer** object in the **Task Executors** section of the Object Library. Place the **AGV** near *Control Point 6*.
- Since the AGV will transport containers from the **Conveyor** to the **Rack Queues**, make an S- or Center Port connection from the AGV to the *PackToNext Conveyor's Exit Transfer* named *Transfer_PackToWhse*.
- Be sure the **Use Transport** box is checked on the **Exit Transfer's Output** section.
- Name the object *AGV_1*.
- In the **Visuals** section of the **Properties** window, set the **Z-rotation** to 90 degrees.
- In the **Task Executer** section of the **Properties** window, set the **Load** and **Unload** times to 0.05 minutes.
- In the **Travel** section of the **Properties** window, set the **Max Speed** to 60 meters/minute.



The AGV returns to one of the control points whenever it is idle. This is incorporated as follows.

- Use the  button in the **Triggers** section to create an *On Resource Available* trigger.
- Use the  button to the right of the **On Resource Available** textbox to select *Travel to a Home Location*.
- In the resulting interface, use the Sampler tool for the **Destination** property to select the **Control Point** named *ControlPoint1*.

26.4 Control Area

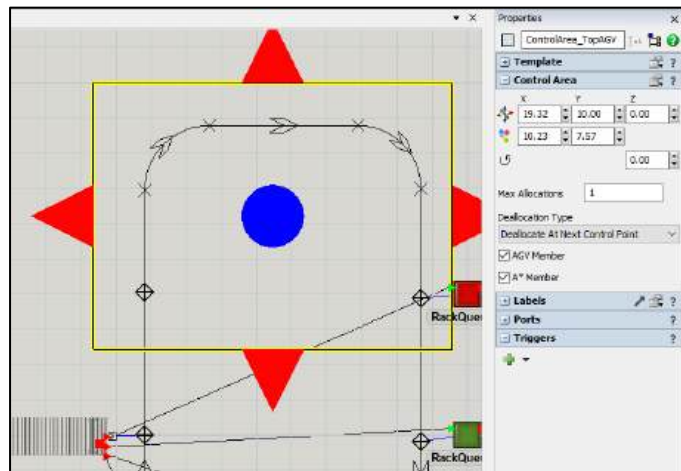
A **Control Area** object enforces mutual exclusion on one or more paths in a network, and it can be integrated with an A* network to restrict the number of travelers in an area. This is its role in this model – to avoid collisions between operators (Order Pickers) traveling through the AGV network when they go on breaks and lunches. The Order Picker(s) use the A* tool to guide it along the shortest path to its destination while avoiding barriers and other objects.

Since the Picking Operator infrequently traverses near the AGV paths, a small model segment is added to test that the **Control Area** is functioning correctly.

Control Area

The figure to the right shows the general properties of a Control Area object.

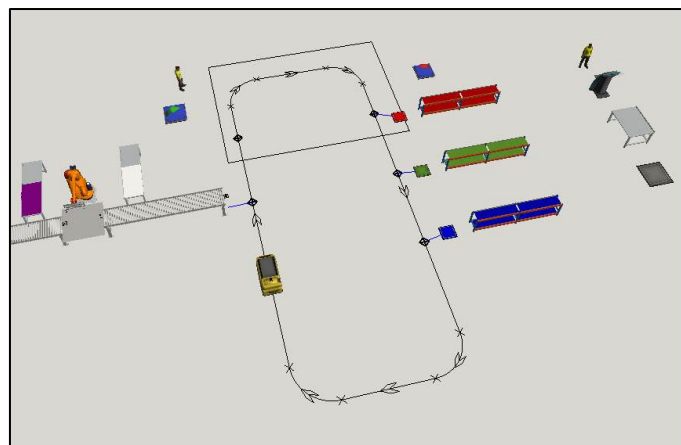
- Drag out a **Control Area** object from the **AGV** section of the Object Library and place it near the AGV network on the modeling surface.
- Using the object's handles (red triangles), size and position it as shown in the figure to the right.
- Name the object *ControlArea_TopAGV*.
- Be sure the following properties are set as follows.
 - **Max Allocation** is 1. This is the maximum number of objects (**Task Executors** and **AGVs**) that can be in the **Control Area** at any time.
 - **AGV Member** box is checked.
 - **A* Member** box is checked.



Test objects

The simple sub-model that tests the **Control Area** uses a **Source**, **Sink**, and **Operator**, as shown in the figure to the right.

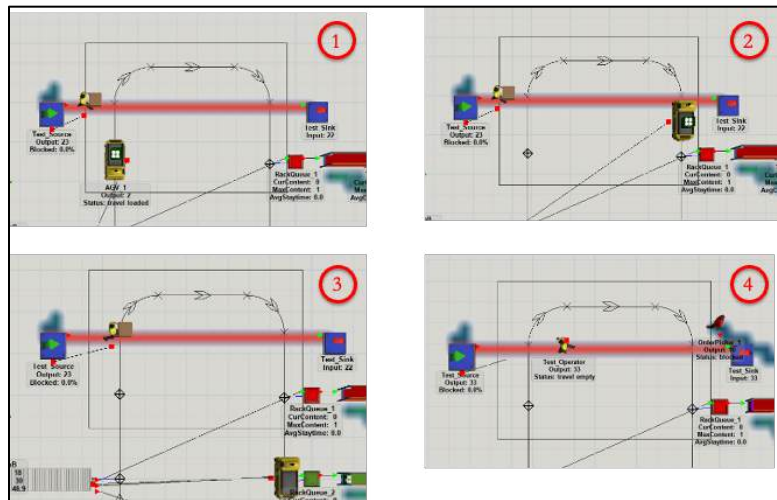
- Drag out the following objects and place them near the AGV network on the modeling surface: **Source**, **Sink**, and **Operator**.
- Arrange the objects as shown in the figure to the right.



- For the **Source**,
 - Name the **Source** *Test_Source*.
 - Make an S- or Center-Port connection from the **Source** to the **Operator**.
 - In the **Source** pane, set the **Inter-Arrival Time** to 3, which generates an arrival (Box) every three minutes.
 - In the **Output** pane, check the **Use Transport** box.
- For the **Sink**, name it *Test_Sink*.
- Make an A-connection from the **Source** to the **Sink**.
- For the **Operator**,
 - Name it *Test_Operator*.
 - In the **Travel** pane, set the **Max Speed** to 25, which slows down the **Operator** so it spends more time in the **Control Area** and is easier to observe its behaviors.
- Open the **A* Navigator** tool in the **Toolbox**. In the **Members** section, select *Traveler Members* and use the  button to obtain a list of objects in the model. In the **Operator** section, select *Test_Operator*.
- While in the **A* Navigator** tool on the **Visual** tab, check the **Show Heat Map** box. This is helpful in tracking Operators' travel.
- **Reset** and **Run** the model and observe the **Operator** and **AGV** behaviors.

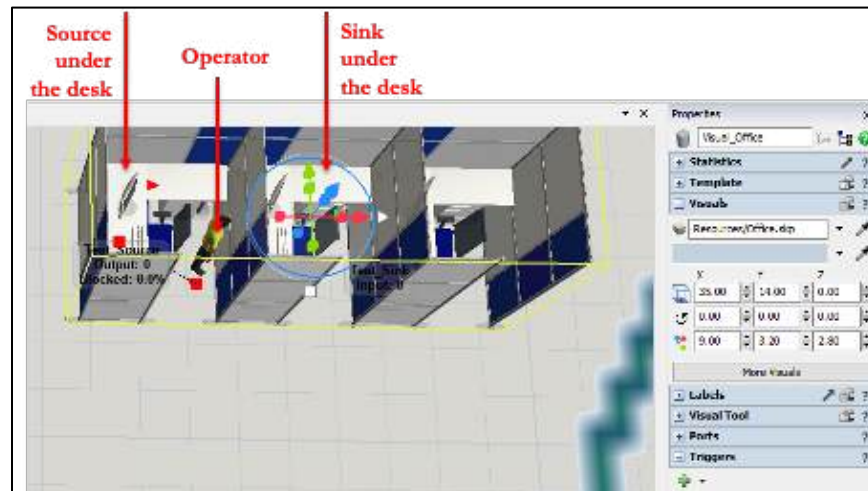
Notice in each of the screenshots in the figure to the right:

1. The **AGV** is in the **Control Area**, and the *Test Operator* is stopped at the left edge of the area.
2. The *Test Operator* is still waiting while the **AGV** continues through the **Control Area**.
3. The *Test Operator* crosses the **Control Area** after the **AGV** leaves the area and is at the **Queue** for the **Rack** named *Type_2*.
4. The *Test Operator* is crossing the **Control Area**, and *OrderPicker_1* is stopped at the edge of the area on the way to a break.



The test segment objects can be deleted once the model is verified, or they can be retained and moved out of the way. In this case, it is retained and hidden by a visual – a set of offices.

- Drag out a **Shape** object from the **Visual** section of the Object Library.
- Name the object *Visual_Office*.
- As shown in the figure below, size and locate the office near the Order Fulfillment Area.



- Move the **Source**, **Sink**, and **Operator** test objects into the office object so they are hidden. The **Operator** does not need to be hidden; it can be visible as if someone is in the office.
- Add the Visual to the **A* Navigator**.
 - Open the **A* Navigator** tool in the **Toolbox**. In the **Members** section, select *FR Members* and use the  button to obtain a list of objects in the model. In the **VisualTool** section, select *Visual_Office*.
 - While in the **A* Navigator** tool on the **Visual** tab, you may want to uncheck the **Show Heat Map** box. This removes tracking the travel of **Operators**.



If you haven't already done so, save the model. Recall that it is good practice to save often.

26.5 Dashboard Charts for AGV and Other Resources

The two key performance indicators for the **AGV** are (1) the contents of the conveyor from which the **AGV** picks up packed containers and (2) the utilization of the **AGV**. They are added to the model in this section. In addition, utilization charts on the Packing Robot and the Picking Operator are added.

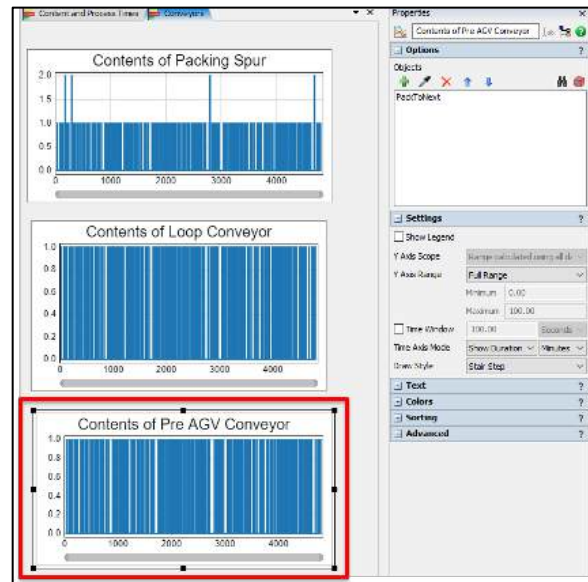
Contents of the Pre-AGV Conveyor

A content graph named Conveyors is added to the **Dashboard**. It is described below and shown in the figure to the right.

- Drag out a *Line Chart* type of **Content** graph and place it below the chart named *Contents of the Loop Conveyor*.

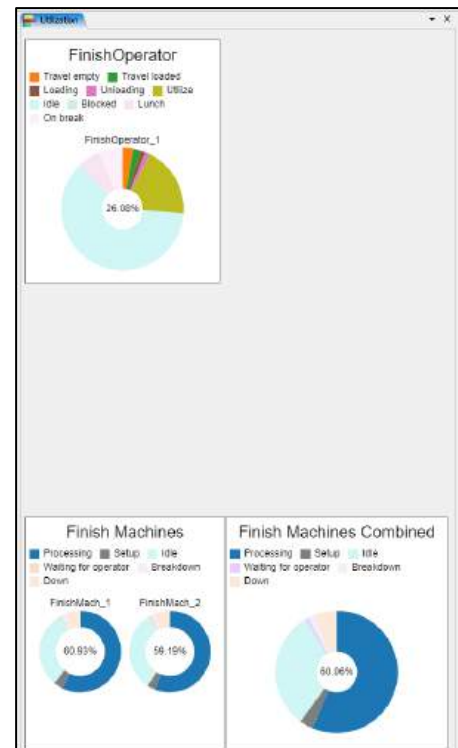
Update the properties as defined below and shown in the figure to the right.

- Name the chart *Contents of Pre AGV Conveyor*.
- Select the conveyor object from where the AGV picks up containers by using the  button in the **Objects** section of the **Properties** window to select **Select Objects**, then **Straight Conveyors**, then **Pack to Next**.
- In the **Settings** pane, uncheck the **Show Legend** box.
- In the **Settings** pane, change *Time Axis Mode* to *Show Duration* and then change *Seconds* to *Minutes*.



Before adding the **AGV** to the **Utilization Dashboard**, resize and rearrange the existing charts so that a few new ones can be added. Resize the charts so they are completely visible, and no vertical or horizontal scroll bars are in the figure. The suggested rearrangement is shown in the figure to the right.

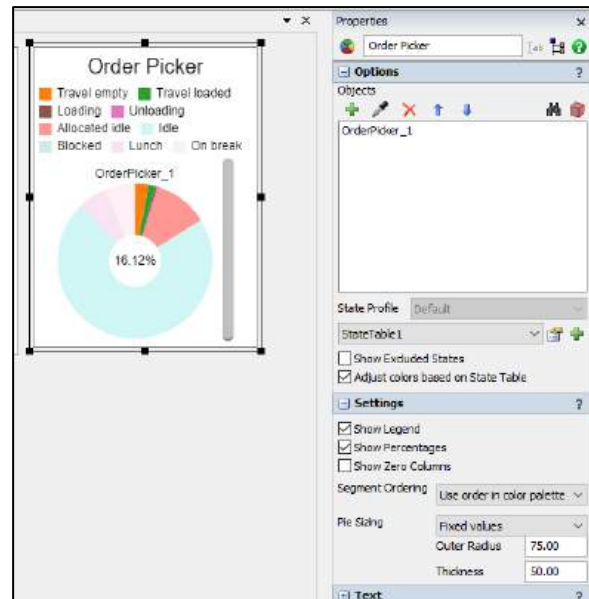
- To help fit the charts, change the **Pie Sizing Outside Radius** as follows.
 - For the *Finish Operator* chart, set the **Outside Radius** to 75.0.
 - For the *Finish Machines* chart, set the **Outside Radius** to 50.0.
 - For the *Finish Machines Combined* chart, set the **Outside Radius** to 75.0.



Add the utilization charts for the Picking Operator, Packing Robot, and AGV. The final *Utilization Dashboard* is shown in the figure at the end of this section.

Picking Operator

- In the **State** pane of the **Dashboard Library**, select a **State** chart, then *Pie Chart*. Position it on the **Dashboard** next to the Finish Operator chart.
- Update its properties as described below and shown in the figure to the right.
- Name the chart *Order Picker*.
- Using the Sampler tool (eyedropper) in the **Objects** section of the **Properties** window, select the **Operator** *OrderPicker_1* in the 3D view.
- In the **Settings** pane, change **Pie Sizing - Outer Radius** to 75.
- In the **Settings** pane, change **Pie Sizing - Thickness** to 50.



Similarly, create the utilization charts for the Robot and AGV as described below.

Robot

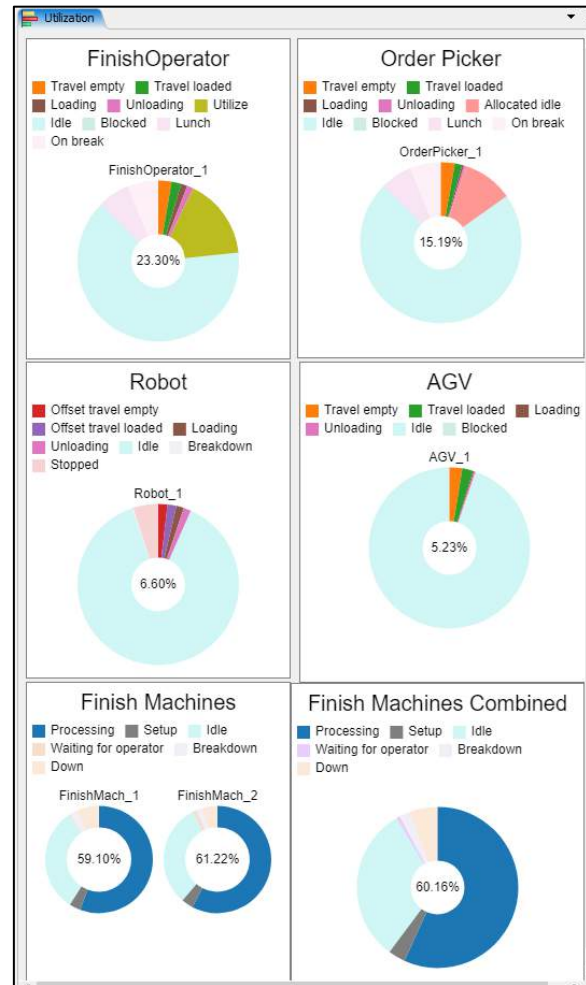
- In the **State** pane of the **Dashboard Library**, select a **State** chart, then *Pie Chart*. Position it on the **Dashboard** below the Finish Operator chart and update its properties as described below.
- Name the chart *Robot*.
- Using the Sampler tool (eyedropper) in the **Objects** section of the **Properties** window, select the **Robot** *Robot_1* in the 3D view.
- In the **Settings** pane, change **Pie Sizing - Outer Radius** to 75.
- In the **Settings** pane, change **Pie Sizing - Thickness** to 50.

AGV

- In the **State** pane of the **Dashboard Library**, select a **State** chart, then *Pie Chart*. Position it on the **Dashboard** below the Order Picker chart and to the right of the Robot chart. Update its properties as described below.
- Name the chart *AGV*.
- Using the Sampler tool (eyedropper) in the **Objects** section of the **Properties** window, select the **AGV** *AGV_1* in the 3D view.
- In the **Settings** pane, change **Pie Sizing - Outer Radius** to 75.
- In the **Settings** pane, change **Pie Sizing - Thickness** to 50.

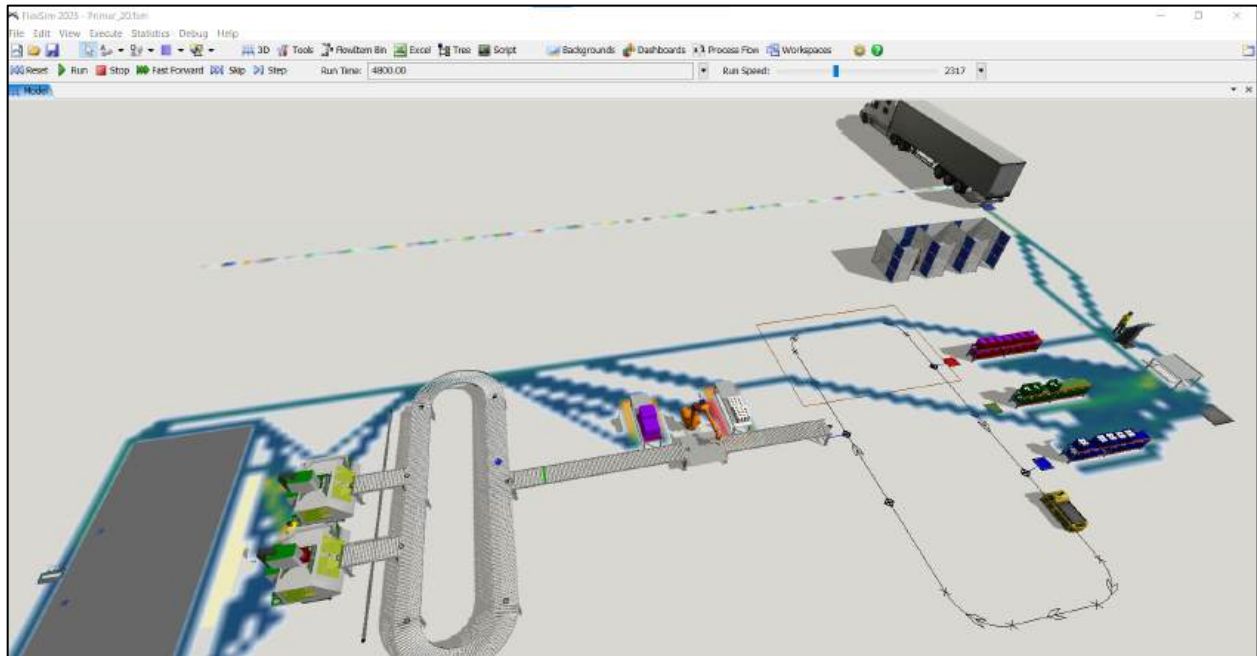
The completed Utilization chart is shown in the figure to the right.

Based on the one 80-hour run of the simulation model that generated the charts to the right, there is plenty of extra capacity in most of the resources. Of course, the resources included in this model are the basic components of the system being designed. As the model is expanded to consider the size of the new facility, these measures will be beneficial in the design of the proposed system.



If you haven't already done so, save the model. Recall that it is good practice to save often.

This concludes the primer model. An overview of the model is shown in the figure below.

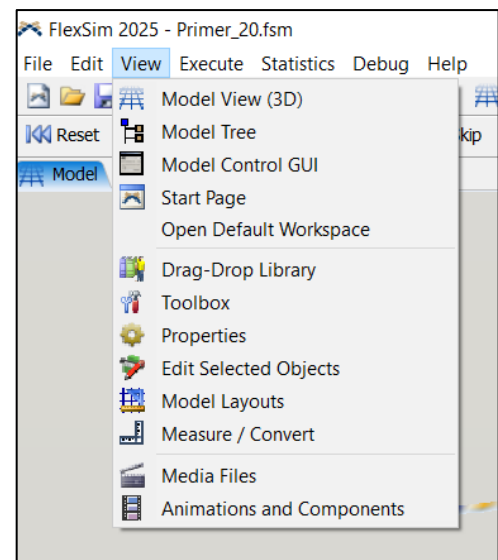


Use the ***Save Model As*** option in the **File** menu to make a copy of the existing model to be further customized in the next section. Again, you can use any file name, but the next model is referred to as **Primer_Final** in the primer.

Note there are no windows – **Library** , **Toolbox**, or **Properties** – visible; they have all been closed.

To open any closed window, use the **View** option on the **Main Menu**.

As shown in the figure to the right, all of the window types can be opened from here. Obviously, there are a number of other windows that have not been addressed in the primer. The three we have dealt with are towards the middle of the list – **Drag-Drop Library**, **Toolbox**, and **Properties**. Of course, we have used the **Model View (3D)**; as mentioned, multiple 3D views can be opened, but it is best to have one model window open and define multiple views (in the Properties window).



PART VII – SUMMARY AND APPENDICES

This section provides a summary of the primer model, three appendixes, and a brief bio of the author.

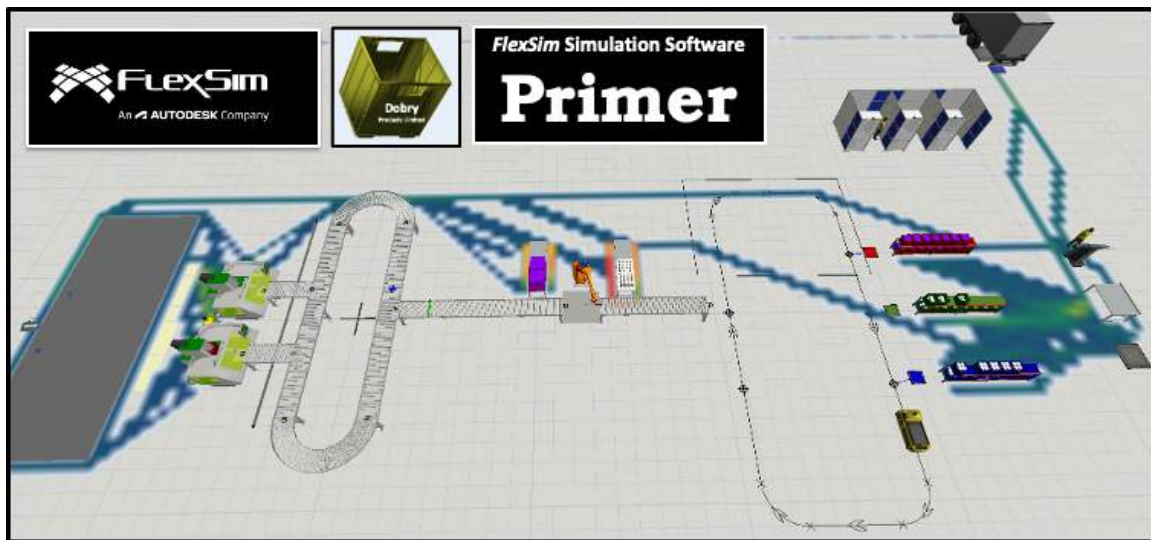
- Chapter 27 summarizes each major area of the model – Finishing Area, Conveyor Transport, Packing Area, AGV Transport, Warehousing, and Order Fulfillment – and summarizes key properties of the model.
- Appendix A is a glossary of key terms used in the primer.
- Appendix B describes and explains elements of *FlexScript* (a subset of *C++*) that are encountered in the primer when discussing setting some property values.
- Appendix C includes a brief description of the concepts and software features added in each primer model and a reference to the section where the model is discussed.
- About the Author provides a brief bio of the primer's author.

27 SUMMARY OF THE PRIMER MODEL

Chapter 27 summarizes each major area of the model – Finishing Area, Conveyor Transport, Packing Area, AGV Transport, Warehousing, and Order Fulfillment – and summarizes key properties of the model.

Congratulations! You have successfully completed the *FlexSim Simulation Software Primer*. Hopefully, you now better understand how to build *FlexSim* simulation models to solve problems and make better decisions through the power of discrete-event simulation and *FlexSim's* robust capabilities.

This final section provides a textual summary of the model that evolved throughout the primer. The final model, as shown in the figure below, results from a development process that progressed from a very simplified version of the system to one that more closely represents the actual operations. Note the heat map that shows operator travel paths in the model. The operators use the A* algorithm to move about the area; i.e., they take the shortest distance between their starting point and destination while avoiding barriers and objects.



Recall that Dobry Products Limited (DPL) is planning to reuse an area in one of its production facilities to finish, pack, and store containers and fill orders. The new production area will finish various types of containers and then the containers are packed by a robot, where the contents depend on the type of container. The packed containers are then moved using an AGV to a warehouse area where they are used to fulfill customer demand.

In this initial model, which would likely continue to evolve and expand as the system design process continues, it is assumed that there will only be three types of containers packed with two types of components. Of course, based on the manner in which the model is developed, it is quite easy to expand the model to consider a larger number of products and components as well as increased production resources, people, and equipment.

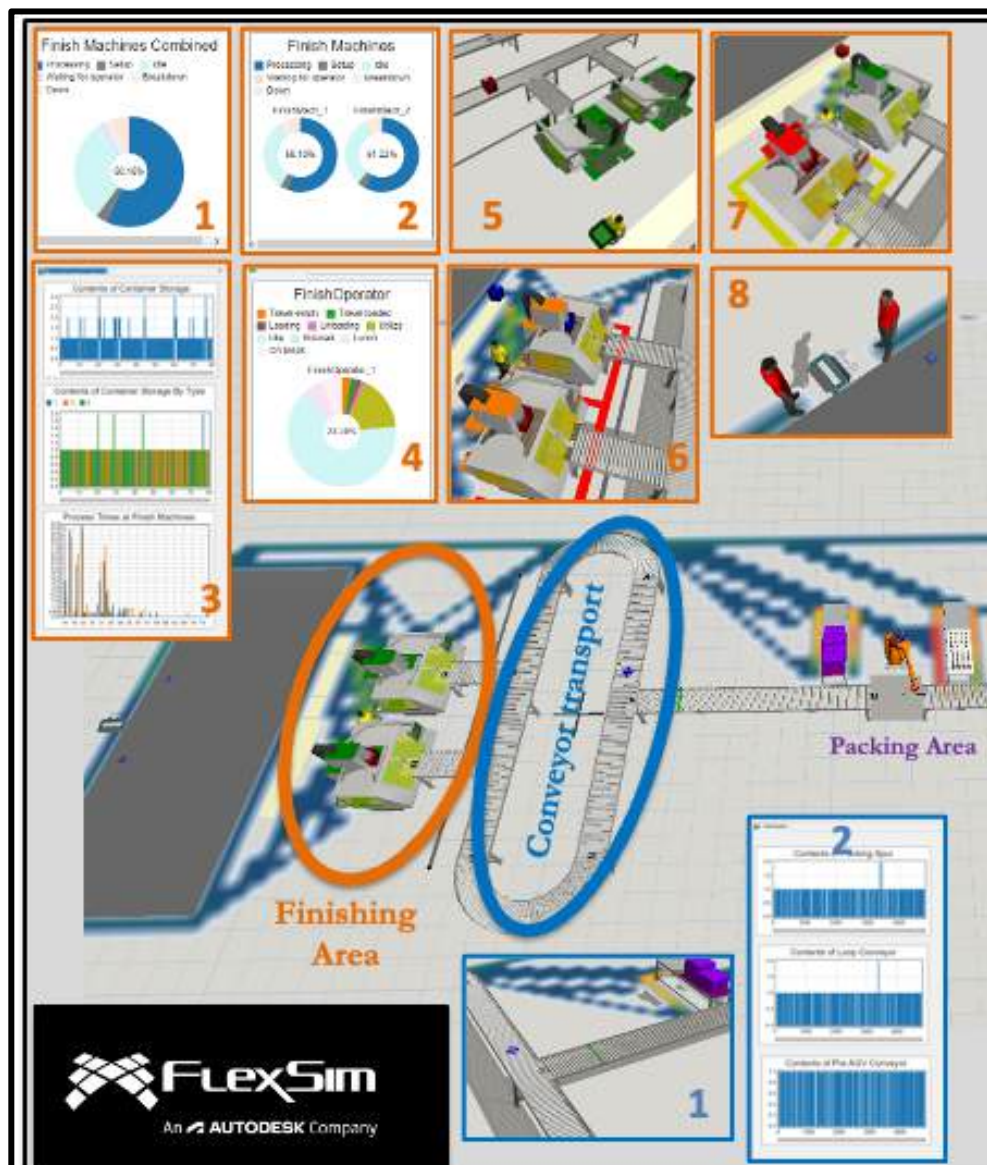
The basic units of measure are meters and minutes.

27.1 Description of modeling each major area of the facility

The following provides a brief description of the model for each major area of the facility – Finishing, Conveyor Transport, Packing, AGV Transport, Warehousing, and Order Fulfilment.

27.1.1 Finishing Area and Conveyor Transport

The following figure shows the Finishing Area and the conveyor used to transport containers between the Finishing and Packing Areas. It provides some close-up views of the objects in the area and charts of key performance indicators, each of which is briefly described below.



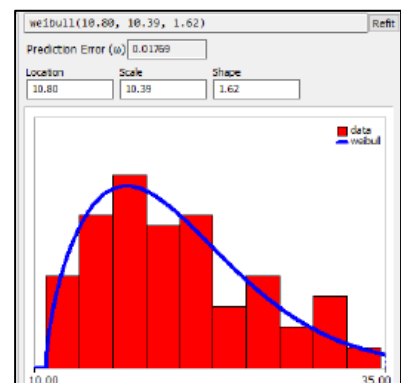
1. Combined utilization pie-chart for the Finishing Machines.
2. Separate utilization pie-charts for each Finishing Machine.
3. Contents of container storage area, both total and by type. Process times at Finishing Machines.
4. Utilization of Finishing Operator.
5. Finishing Operator moving an incoming container to a Finishing Machine. Type 1 container is seen exiting a Finishing Machine on the conveyor to the Packing Area.
6. Finishing Machines down for a quality check that occurs every 10 minutes and has a duration of 15 seconds. Machines are in the “down” state and are colored orange during the downtime.
7. Finishing Machine broken down, as indicated by its red color, which occurs randomly on average every two hours, and has a random time to repair that varies between 5 and 15 minutes. A Finishing Operator performs the repair. The other Finishing Machine, colored green, is up and available to process.
8. Finishing and Order Picker Operators on break (personal breaks and lunch). They travel to the location of the Dispatcher object and remain there for the duration of the break. The gray area is a barrier where the Operators cannot travel.

The figure above also provides a close-up view of one part of the conveyor system and its performance indicators for the conveyors.

1. The screenshot shows where the loop and straight conveyor to the Packing Area meet, a Decision Point (blue triangle object), and a Photo Eye (green line object) that controls the container flow into the Packing Area. If the photo eye is covered, approaching containers do not enter the conveyor to the Packing Area; instead, they make a loop on the conveyor and try again.
2. The charts are time-series of the contents of the conveyor system – straight conveyor to Packing, loop conveyor, and straight conveyor from Packing to AGV pickup.

Container arrivals

The containers arrive at the finishing area (upstream boundary of the model) with an average time between arrivals of about 20 minutes. (Alternatively, the average arrival rate of about three containers is per hour.) *FlexSim's* Distribution Fitter, a part of the Empirical Distribution tool, was used to fit a random sample of 100 interarrival times. The best fit was to a Weibull distribution, which is shown in the figure to the right overlayed on a histogram of the sample data. The distribution was further constrained to generate no interarrival times less than 10 or more than 35 minutes.



It is assumed there are no breaks in the arrival pattern, i.e., there is no downtime in the upstream operation, and that the type of arriving container is random and is based on the percentage of product mix; i.e., there is no batching of containers in upstream operation.

Arriving containers wait to be placed on a Finishing Machine in an area that can store five containers; the storage area size is 1.1 meters by 8 meters. If when a container arrives, and there is no available space, it is diverted to another area in the facility and is no longer considered in the model other than it is counted as a “diverted item.”

Finishing process

Containers are loaded onto one of two Finishing Machines by a single Finishing Operator; the load time is 3 seconds. The operator uses a variation of the Shortest Processing Time (SPT) rule to select a container to load on an available machine and not the common First-In, First-Out (FIFO) rule. SPT means the quickest container to process on a Finishing Machine is loaded first. This is, if any Type 1 (red) containers are waiting, they are loaded first since they have the shortest processing time. If there are no Type 1s, then Type 2s (green) would be loaded first. Finally, Type 3s (blue) are only loaded if there are no Type 1s and 2s. However, Dobry does not want items to wait in the buffer too long; therefore, they follow the SPT rule unless a container has waited more than 30 minutes or some other specified threshold value.

Currently, two identical finishing machines can process any type of container. Each machine is 3 meters square and 2.5 meters high.

The processing time depends on the type of container – Type 1 (red), Type 2 (green), and Type 3 (blue) take 10, 20, and 30 minutes per container, respectively. The Finishing Machine automatically processes each container; i.e., the Operator is not involved in the finishing process on the machine.

However, a setup is required before processing if the loaded container differs from the one that just finished processing on that machine. The Finishing Operator performs the setup, and the time needed to perform the setup is 2 minutes. Once loaded, the machine processes the container without the operator.

After processing, the machine itself loads the finished container onto a conveyor that transports it to the packing area. The conveyor travels at a speed of 60 meters per minute.

Finishing machines are subject to two types of downtime.

- The first type of downtime is for a quality check where each machine pauses for 15 seconds every ten minutes to upload data. This occurs regardless of the machine's state and does not require the operator.
- The second type of downtime is when there is some type of failure within the machine. The time between failures is an exponentially distributed random variable with a mean of two hours. The time to repair the problem is also a random variable that is uniformly distributed between 5 and 15 minutes. For the failure downtime, the finishing operator performs the repair. Failures can only occur when a machine is running.

The Finishing Operator travels at an average speed of 60 meters per minute and takes three seconds to load or unload any item. The operator takes two 15-minute breaks per shift, one after two hours and one after six hours into the shift. The operator also takes a 30-minute meal break in the middle of the eight-hour shift. For all breaks, the operator travels to a break area outside the Finishing and Packing Areas.

Conveyance to Packing

Once a finished container completes processing, it is transported to the Packing Area via a conveyor. If the straight conveyor that leads to packing becomes full, containers will travel around a conveyor loop until they can access the packing line.

27.1.2 Packing Area and AGV Transport

The following figure shows the Packing Area and the AGV system that is used to transport packed containers between the Packing and Warehousing Areas. It provides some close-up views of the objects in the area and charts of key performance indicators, each of which is briefly described below.



1. Time series plot of the level of component inventory by component type.
2. Close-up view of the Packing Robot loading a Type 2 component (white cylinder) from its storage area into a container at the packing station.
3. Close-up view of the Finishing Operator unloading a batch of Type 1 components. The packing robot is stopped while the Finishing Operator is in the area.
4. Pie chart of the utilization of the Packing Robot.

The figure above also provides the AGV's key performance indicator and close-up views of its operation.

1. Pie chart of the utilization of the AGV.
2. Close-up view of the AGV loading a container from the Packing Area.
3. Close-up view of the AGV unloading a container at the Warehousing Area.

Packing Area

A robot performs all operations in the Packing Area.

Once a container arrives at the end of the conveyor that enters the Packing Area and the packing station is idle, the robot moves the container onto the single packing station/table. The container is then packed with components, one by one, using the specified number and type of components. The contents for the three types of containers used in this model are defined below. Only two types of components are considered in this model: Component A (purple box) and Component B (white cylinder). The model has been designed to easily scale up to more container types, component types, Finishing Machines, Packing Robots, AGVs, etc.

- Type 1 container is packed with 2 Component As and no Component Bs.
- Type 2 container is packed with no Component As and 4 Component Bs.
- Type 3 container is packed with 1 Component A and 4 Component Bs.

Once packed, the robot moves the container onto a conveyor that transports it to an AGV pickup point, where an AGV transports it to the Warehouse Area.

The resupply of components to be packed into the containers is managed according to a Reorder-Point type of inventory system. That is, when the on-hand quantity of a component in the Packing Area drops to a specified reorder point level, a batch of that component is ordered. After an order time delay, the batch is delivered to the Packing Area. When a batch of components arrives, the Finishing Operator unloads the batch into the component's storage area. The time to load a component is three seconds, as is the time to unload. The reordering process is modeled using Process Flow in *FlexSim*.

The robot is stopped when an operator is in the area processing a batch of components.

Each robot is subject to downtime. The time between failures is exponentially distributed with a mean of four hours. The time to repair is uniformly distributed between five and ten minutes. The repair is performed by the Finishing Operator.

The Packing Area is pre-loaded with a specified number of components at the start of a simulation; i.e., there is an initial inventory of components at the start of a simulation.

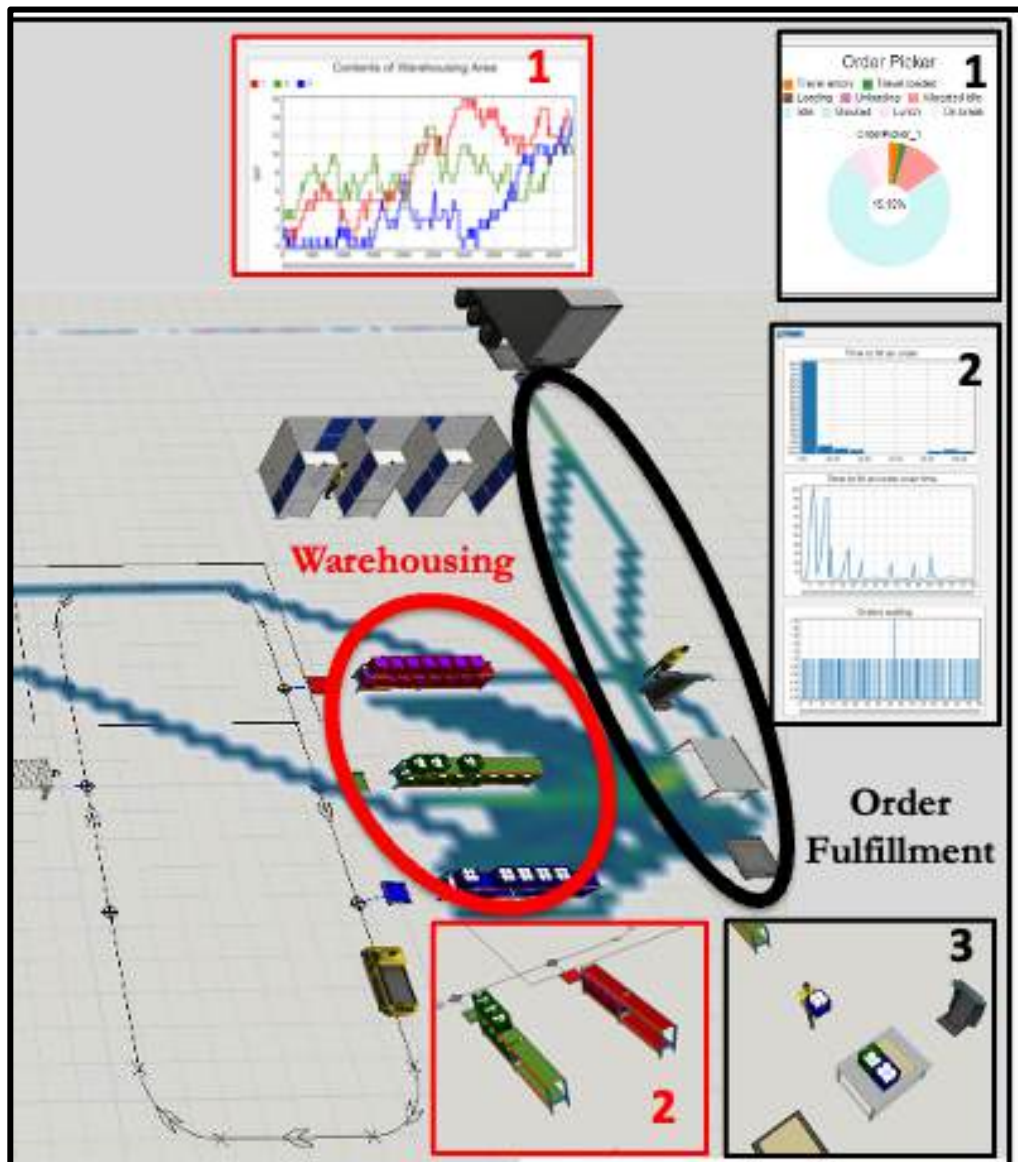
AGV transport

Containers are transported from the Packing Area to the Warehousing Area via a single AGV. The AGV travels on a simple uni-directional loop network at a speed of 60 meters per minute; the network is about 52.5 meters long. The time to load and unload each container is three seconds.

The Order Pickers who work in the Order-Fulfillment Area may cross the AGV network when traveling to and from the facility's break area. Therefore, for safety reasons, part of the AGV network operates within a Control Area, restricting travel to only one Task Executer object (Operator or AGV) at a time.

27.1.3 Warehousing Area and Order Fulfillment

The following figure shows the Warehousing and Order-Fulfillment Areas that store packed containers and prepare them to fulfill customer orders. It provides some close-up views of the objects in the area and charts of key performance indicators, each of which is briefly described below.



1. Time series plot of the level of finished and packed container inventory by container type.
2. Close-up view of the racks used to store containers before order fulfillment.

The figure above also provides the Order Fulfillment Area's key performance indicators and a close-up view of its operation.

1. Pie chart of the utilization of the Order Picker.
2. Histogram and time series chart of the time to fulfill an order. It also shows a time-series chart of the number of orders awaiting processing.
3. Close-up view of the Order Picker fulfilling an order. In this case, the operator is traveling from the Warehousing Area with a container that is a part of the order being processed. Two containers have already been picked and are on the order receptacle on the worktable.

Warehousing

In the Warehousing Area, there is one rack for each type of container, and all racks can hold up to 16 containers on two levels.

Once a container is dropped off at the appropriate pre-rack queue, it is automatically loaded into the rack if there is space. If a rack is full, the containers are automatically loaded into the rack when space becomes available. The model includes the pre-rack queue to handle an overflow when a rack's capacity is reached. This helps to size the system during the design process. If the queues are not included before the racks, containers would back up the conveyor from the Packing Area since there would be no place to route them. In the real system, it may be necessary for an operator, possibly on a schedule, to move containers from the pre-rack queues to their racks. If so, this would be an easy task to model.

Logic in Process Flow creates an initial inventory of containers on the racks when a simulation starts. The total number of containers available on the racks at the start of a simulation is specified as a Model Parameter. The type of container is randomly assigned based on the Product Mix distribution, which is specified as an Empirical Distribution.

Order Fulfillment

The Order Fulfillment process is modeled using Process Flow logic.

The arrival of containers in orders is assumed to be similar to that of the arrival of containers to the Finishing Area from the upstream process, which was expressed earlier as a Weibull distribution. Of course, this is the time between the arrival of containers, not orders. However, marketing has estimated the size of each order to be one container 10% of the time, two containers 15% of the time, three containers 50% of the time, four containers 15% of the time, and five containers 10% of the time. This is an Empirical Distribution referred to as the order-size distribution. Based on this distribution, an order consists of one to five containers; the average order size is 3.0.

It follows that the time between orders (not containers) is the product of the order-size distribution and the interarrival time distribution of containers. Therefore, if the average time between arrivals of containers is 20 minutes and there are an average of 3.0 containers per order, then the average time between orders is about 60 minutes. Thus, the distribution of the time between order arrivals is assumed to be the product of the two empirical distributions. Of course, once the products are being produced and sold, a more representative

distribution of the order-arrival process can be determined. Still, for designing the production system, this should suffice.

In the *FlexSim* model, an order is represented as a flat receptacle that holds an order's containers. The order items arrive at an order queue and wait until an Order Picker operator is ready to process the order. At that time, an Order Picker moves the order from the queue to a worktable.

Orders are processed in the order they arrive, and an Order Picker completes an order before moving to the next order; i.e., if a container is unavailable, the operator waits until it is available before processing the next item. An operator could process multiple orders at a time, but more space and definitions would be needed, such as how long an operator waits to start the next order, the maximum number of partial orders that can be underway, etc.

The Order Picker gathers the requisite number and type of containers from their racks in the Warehousing Area and places them on the order receptacle. When all of the order's containers are collected, the operator takes a little over a minute to complete processing; the time to complete an order is assumed to be triangularly distributed. Once an order is complete, it is transported by the Order Picker to a fulfilled orders area, which is represented as a semi-truck. This is the end of the model; i.e., what happens to an order beyond this point is not considered in this model. After disposing of the order, an Order Picker enters information into a computer before processing the next order. The information entry time is assumed to be 15 seconds (0.25 minutes).

The Order Pickers follow the same break schedule as the Finishing Operators.

27.2 Key properties of each aspect of the model.

The key properties of the model are summarized in the following tables.

- Table 1 contains information on the system's products, i.e., the containers.
- Table 2 contains information on the components, i.e., the items placed in the containers.
- Table 3 provides general system and model information and is composed of three parts
 - Part 1 – Process times, downtimes, and routing rules.
 - Part 2 – Object capacities, speeds, and dimensions,
 - Part 3 – Key Performance Indicators & Charts

It is recommended that each simulation run is for 80 hours (4,800 minutes), and when running experiments, each scenario is replicated 20 times. Discussion of how these values are set is beyond the scope of the primer.

Table 1 - Information on the system's product, i.e., the containers.

					
Product / Container					
<u>Property</u>		1	2	3	<u>Reference/Source</u>
Name		Container			Flowitem Bin
Type		1	2	3	Label
<u>Visual</u>					
Size (l x w x h), meters		0.5 x 0.5 x 0.5	0.5 x 0.5 x 0.5	0.5 x 0.5 x 0.5	Flowitem Bin, Container
Shape		Tote	Tote	Tote	Flowitem Bin, Container
Color		Red	Green	Blue	Source, ContainerArrivals
<u>Contents</u>		2, 0	0, 4	1, 4	Global Table, Packing
(Quantity of Components A and B per container)					
<u>Process Times, minutes</u>					
Setup at Finising [1]		2	2	2	Finishing Machine object
Finishing		15	20	30	Model Parameter, Finish Times
Packing		0.60	0.50	0.75	Global Table, Process Times
Order Fulfillment		Triangular Distribution with min. 0.75, max. 1.50, and most likley 1.25			Process Flow, OrderFulfillment, Process Order
<u>Product Mix</u>		30%	35%	35%	Empirical Distribution, Product Mix
<u>Interarrival Time</u>		Weibull Distribution with min. 10 minutes and max. 35 minutes; mean ~ 20 minutes			Empirical/Fitted Distribution, InterArrivalTimes
<u>Initial Inventory [2]</u>		5			Model Parameter, InitContInv
<u>Notes [-]</u>					
1 Setup is incurred when a container is different from the one					
2 Type is based on Product Mix.					

Table 2 - Information on components that are placed in containers.

				
		Component		
Property		A	B	Reference/Source
Name		CompA	CompB	Flowitem Bin
Visual				
Size (l x w x h), meters		0.4 x 0.4 x 0.2	0.2 x 0.2 x 0.2	Flowitem Bin
Shape		Box	Cylinder	Flowitem Bin
Color		Purple	White	Flowitem Bin
Resupply				
Resupply Method				Fixed schedule, then Reorder Point
Reorder Point, units		6	20	Model Parameter, ReOrderPoint
Batch (Reorder) Size, units		24	60	Model Parameter, ComponentBatchSize
Reorder Time, minutes		60.0	60.0	Model Parameter, ReOrderTime
Time between deliveries, minutes		60.0	60.0	Model Parameter, ComponentFrequency
Store Capacity		1000	1000	Queue object

Table 3 – General system and model information – Part 1.

<u>Load Times, minutes</u>		0.05	all Task Executors (Operators, AGVs, Robots)
<u>Unload Times, minutes</u>		0.05	all Task Executors (Operators, AGVs, Robots)
<u>Operator Downtime (Finishing, Order Picker)</u>			<u>Where set</u>
Breaks (2 per shift), minutes		15	Time Table, OperatorBreaks
Lunch (1 per shift), minutes		30	Time Table, OperatorBreaks
<u>Finishing Machine Downtime</u>			<u>Where set, type</u>
Quality Check		see Note [1]	MTBF/MTTR QualityCheck, Clock-based
Breakdown		see Note [2]	MTBF/MTTR FM_Failures, State-based
<u>Robot Downtime</u>			<u>Where set, type</u>
Breakdown		see Note [3]	MTBF/MTTR PackingRobots, State-based
<u>Routing Rules</u>			
Buffer to Finish Machines		SPT (Shortest Process Time) unless wait is more than specified threshold	
To Packing		FIFO	on conveyor
To Warehouse by AGV		FIFO	on conveyor
<u>Notes</u>			
1 Up Time = 10 minutes, Down Time = 0.25 minutes			
2 Up Time = exponential with mean of 120 minutes, Down Time = uniform between 5 and 15 minutes			
3 Up Time = exponential with mean of 240 minutes, Down Time = uniform between 5 and 10 minutes			

Table 3 – General system and model information – Part 2.

Property/Characteristic	Value	Reference/Source
Capacities	Quantity	Quantity item, where set
Pre Finishing Buffer	10	unfinished containers, Model Parameter
Pre-Rack Buffer	1000	packed container, Queue
Order Queue	1000	order, Queue
Finishing Machines	1	container, Object
Finishing Operator	1	container, Object
Finishing to Packing Conveyor	100	containers, multiple Objects, approximate
Packing Robot	1	container and components, Object
Packing table	1	container but multiple components in container, Object
Packing to AGV Conveyor	12	containers, multiple Objects, approximate
AGV	1	container, Object
AGV Network	25	AGVs, multiple Objects, approximate
Warehouse Racks	16	containers, Object
Order Fulfillment Table	1	order, Process Flow
Order Picker	1	order, Object
Speeds	meters per minute	Where set
Finishing Operator	60.0	FinishingOperator object(s)
Order Picker	60.0	OrderPicker object(s)
Conveyors	60.0	Conveyor objects
AGV	60.0	AGV object(s)
Dimensions, (length, width, height), meters		Where set
Container Storage	8.0, 1.1, 0.0	Queue object
Finishing Machines	3.0, 3.0, 2.5	Processor object
Packing table	2.0, 2.0, 1.0	Plane object
Component's table	1.0, 3.5, 1.0	Shape Object
Component's store	1.0, 2.0, 0.0	Queue object
Robot	0.9, 1.3, 1.2	Robot object
Pre-Rack queue	0.6, 0.6, 0.05	Queue object
Warehouse Racks	4.5, 0.6, 1.75	Rack object
AGV	2.0, 1.0, 1.0	AGV object
Order-processing table	1.5, 2.0, 1.0	Shape object
Control Area	10.2, 7.6, n/a	Control Area object
Truck at Order Fulfillment	22.0, 2.6, 4.2	Shape Object
Conveyor Lengths, meters (all are 1 meter wide)		Where set
Finishing to Loop	2.5	Conveyor objects
Loop	36.5	Conveyor objects
Loop to Packing	9.0	Conveyor objects
Packing to AGV	6.0	Conveyor objects
AGV Path Length, meters	52.5	AGV Path object

Table 3 – General system and model information – Part 3.

Performance Measures	Performance Measure Table	Measure
Redirected Containers	RedirectedContainers	Number of container redirected due to no space.
Packing Throughput	PackingThroughput	Number of containers through Packing.
Inventory of Component A	Inventory_CompA	Current inventory of Component A.
Inventory of Component B	Inventory_CompB	Current inventory of Component B.
Time to Fulfill Order	TimeToFulfillOrder	Average time to fulfill order in minutes.
Charts		<u>Dashboard name. Chart type.</u>
Contents of Container Storage		Content and Process Times. Times series
Contents of Container StorageBy Type		Content and Process Times. Times series.
Process Times at Finish Machines		Content and Process Times. Histogram.
Finish Operator Utilization		Utilization. Pie Chart.
Order Picker Utilization		Utilization. Pie Chart.
Robot Utilization		Utilization. Pie Chart.
AGV Utilization		Utilization. Pie Chart.
Finish Machines Utilization		Utilization. Pie Chart.
Finish Machines Combined Utilization		Utilization. Pie Chart.
Contents of Packing Spur		Conveyors. Time series.
Contents of Loop Conveyor		Conveyors. Time series.
Contents of Pre AGV Conveyor		Conveyors. Time series.
Contents of Component Storage Over Time		Inventory. Time series.
Contents of Warehousing Area		Inventory. Time series.
Time to fill an order		Orders. Histogram.
Time to fill an order over time		Order. Time series.
Orders waiting		Order. Time series.

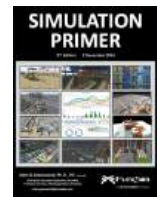
27.3 Epilogue

This primer focused on the modeling and analysis of a small but not simple operations system to aid in the design of a new facility. Simulation is invaluable for understanding and assessing dynamic behaviors and performance, i.e., operations dynamics. The primer leads the reader through detailed, step-by-step instructions for building a simulation model. A screenshot of the final model is provided on the next page. The primer uses a sequential model-development process so that the model evolves and builds upon simpler models. In addition to describing the mechanics of building a model, the primer demonstrates the power of using discrete-event simulation in general and *FlexSim* in particular to support the decision-making and problem-solving processes. As the model evolves, insight and rationale are provided, and good modeling and analysis practices are introduced to demonstrate that model building is not just carrying out rote commands.

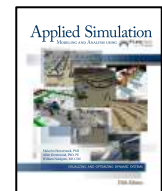
While all of *FlexSim*'s capabilities are not presented, the primer raises awareness of many of the more advanced features available in *FlexSim* without covering the details at that time. Completing the primer should provide the reader with a solid foundation to explore topics in more detail in the *FlexSim User Manual*, tutorials, videos, blogs, *FlexSim Answers*, etc.

It was also beyond the scope of the primer to discuss or even introduce, all of the topics and methods that are associated with simulation modeling and analysis. Therefore, it is suggested that the interested reader consult the following or other more general references on simulation.

Greenwood, A. *Simulation Primer*, FlexSim Software Products, Inc., 2024.



Beaverstock, M., Greenwood, A., and Nordgren, W. *Applied Simulation Modeling and Analysis Using FlexSim*, 5th Edition, FlexSim Software Products, Inc., 2017.



Your feedback is welcome!

Since improvements cannot occur in a vacuum, the author welcomes comments and suggestions; please send them via email to allen.greenwood@autodesk.com



APPENDIX A – GLOSSARY OF KEY TERMS

Appendix A is a glossary of key terms used in the primer.

3D shape	A visual representation of an <i>object</i> defined through an imported graphics file. The object's user interface defines its size, location, color, etc.
A* Algorithm	A heuristic search method used to find the shortest path between objects considering barriers or obstacles and no-travel zones.
A* Navigator	A <i>FlexSim</i> tool for creating barriers and identifying which fixed objects are barriers in a model. The barriers influence and restrict Task Executer (TE) travel paths between objects. In contrast to <i>Path Networks</i> , which also controls Task Executer movement, A* tells the TE where <u>not</u> to travel, compared to <i>Path Networks</i> , which tells a TE where to travel.
Activity (Process Flow)	Operations or steps in a logical process that are the building blocks of <i>FlexSim</i> 's Process Flow. They are analogous to objects in 3D in that they (1) are dragged from their library onto a Process Flow modeling surface or workspace and (2) have properties that define their behavior.
Activity Set (Process Flow)	Preconfigured activities bundled together that model a basic People Module task, such as Walk then Process, Escort then Process, Wait then Process, etc.
Address (Rack)	Identifier of where an item is located/stored in a storage system. Typically, a combination of letters, numbers, and separators that uniquely defines a <i>slot</i> in a storage system.
AGV	Automated Guided Vehicle. A means to transport items using an item that follows a defined path.
Analysis	See <i>Simulation Analysis</i> .
Animation	A sequence of object movements that are triggered as a simulation runs.
Array	A data structure that consists of an ordered series of elements that are indexed.
Bay (Rack)	A section of a Rack object representing storage areas along the horizontal axis.
Continuous simulation	A type of simulation where the states of a system change continuously over time, e.g., the filling of a tank with water or other fluid. This is compared to <i>discrete-event simulations</i> where states change at discrete points in time, e.g., a part arriving at a machine to be processed.

Conveyor	A mechanical means to move items between points in space. There are different types of conveyors, all of which can easily be modeled in <i>FlexSim</i>, including: • Belt conveyor, sometimes referred to as a non-accumulating conveyor, behaves as follows. An item travels down the conveyor until the end and stops if it cannot be removed (downstream object or transport is unavailable). When one item stops, the belt stops, and all other items on the conveyor stop at their current location. • Merge conveyor combines items from multiple conveyor lines into a single line for further transportation. • Power and free conveyor uses a free rail and multiple drive rails to move items loaded on trolleys through a system. • Roller conveyor, sometimes referred to as an accumulating conveyor, behaves as follows. An item travels along the conveyor until the end and stops if it cannot be removed (downstream object or transport is unavailable). Subsequent items on the conveyor continue to flow on the conveyor and stop behind the one in front. • Slug conveyor is a type of merge conveyor where a group of accumulated items are released for further transport; thus, items are released in “slugs.” • Sorting conveyor transports items to different destinations based on stated criteria and item properties. Sorting typically involves conveyor logic objects such as decision points, stations, and photo eyes.
Dashboard	An area for displaying values of some variable as a model runs, typically the current state of a system or summary measures (e.g., mean). The display may be in text form but is normally a graph or chart that updates as a model runs. Dashboards provide a means to view the system dynamics.
Discrete-Event Simulation	The means to consider the dynamics of a system by creating and managing events at discrete points in time. Each event changes one or more states of a system. The resulting state change and modeling logic causes an action or activity to occur.
Distribution Fitter	A statistical tool used to best fit system data to theoretical probability distributions. It is typically used to define input distributions to a simulation model from operational data but also to characterize output data obtained from a simulation as probability distributions.
Domain expert	A stakeholder in a simulation project with specialized knowledge or skills in an area of interest.
Dot Notation	A convention in object-oriented programming where the elements of an object (variables and methods) are separated by dots, e.g., Processor.Type where Type is a variable attribute (label) of the object Processor.

Downtime	The time period when a resource is not available to perform its basic functions. Downtimes are either planned or unplanned. - Planned downtimes are expected to occur at known times and typically recur on a regular basis. Examples include shift schedules, operator breaks, and periodic quality checks. Planned downtimes are implemented in <i>FlexSim</i> via the Time Table tool. - Unplanned downtimes occur unexpectedly. Examples include machine breakdowns and operator absenteeism. Unplanned downtimes are implemented in <i>FlexSim</i> via the MTBF/MTTR tool.
Empirical Distribution	A probability distribution derived directly from system data; i.e., the system data are not fit to a <i>theoretical</i> probability distribution.
Event	An occurrence at an instance in time. When an event occurs, it causes one or more system states to change and possibly one or more actions.
Experiment	A set of <i>scenarios</i> and <i>performance measures</i> considered for analysis. The simulation model is run for a prescribed number of replications and duration and may involve excluding a <i>warm-up</i> period.
Experimenter	A tool for designing, executing, analyzing, and possibly optimizing simulation models. It provides a means for defining <i>scenarios</i> , <i>performance measures</i> , number of <i>replications</i> , simulated time or duration, and whether a <i>warm-up</i> period is considered. In <i>FlexSim</i> , the Experimenter can directly connect with <i>OptQuest</i> to perform optimizations.
Fixed Resource	Objects in a model that are fixed or stationary, e.g., Processor or Combiner. Typically, fixed resources process flowitems.
FlexScript	A scripting language that is a subset of C++ within <i>FlexSim</i> that can be used to define logic and behaviors via custom coding. FlexScript includes many pre-built functions, or commands, for performing common programming operations.
Flowitem (Item)	As the name indicates, something that flows through a simulation model. Movement is triggered by events and controlled by model logic. Similar to entities or transactions in other simulation software.
Flowitem Bin	A tool in the Toolbox for customizing and creating flowitem classes, including size, shape, graphic, labels, packing method, etc.
Global Table	A tool that provides a systematic means to store information in terms of rows and columns.

Global Variable	User-defined information that can be accessed and updated from anywhere in a <i>FlexSim</i> model at any time during a simulation. Each variable is typed as Integer, Real, String, Array, etc.
Item	See <i>Flowitem</i> .
Job	A defined strategy for generating scenarios in the <i>Experimenter</i> . There are three type of jobs in <i>FlexSim</i> – experiment, range-based, and optimization.
Label	A user-defined property on an object or item that can be defined, used, and/or updated at any time during a simulation.
Level	Storage spaces along the vertical axis of a <i>Bay</i> in a Rack object.
Library	Set of modeling objects, tools, Process Flow activities, dashboard, or user-defined.
List	A tool for creating complex flows in a model. Basically, under a specified condition, an object pushes information onto a list (referred to as a list entry), and under a different condition, an object, using selection criteria, pulls an entry from the list, resulting in an action in a model.
Mobile Resource	see <i>Task Executer</i> .
Model	Representation or abstraction of a system and its behavior. A model represents a system's components and the interaction among the components, the variability inherent in the system, and the system's dynamic behavior.
Model Fidelity	The degree to which a model represents the actual system. While it might appear that more is better, in this case, it is not. A model should only be as detailed as needed to answer the posed question(s). Of course, this is easier said than done; determining the proper fidelity comes with practice and experience.
Modeling	See <i>Simulation Modeling</i> .
Modeling Surface	The interface where models are built in 3D space by dragging and dropping objects onto the gridded surface and locating them in 3D space in terms of their x, y, and z coordinates. In <i>FlexSim</i> , the surface grid's unit of measure is specified as Length Units when a model is created; e.g., if the Length Unit is feet, then each grid unit represents one foot in the x and y directions. Similar modeling environments are available when defining model logic in Process Flow and creating charts and graphs in Dashboards; however, these are just dimensionless work surfaces for organizing work.
Modeling Workspace	See <i>Modeling Surface</i> .

Monte Carlo simulation	A means to obtain numerical results based on repeated random sampling.
MTBF/MTTR	Mean Time Between Failures / Mean Time To Repair. In <i>FlexSim</i> , this tool provides the ability to model the reliability of an object in terms of a set of parameters and properties such as the time between failures (operating time), downtime duration, resources needed, states affecting the time between failures (e.g., clock time versus processing time), etc. Typically, the time between failures and repair times are random variables.
Object	A pre-built, yet customizable, representation of actions commonly found in operations systems, such as planned and unplanned delays (e.g., processing, storage), transportation via fixed (e.g., conveyor) or mobile (e.g., operator) means, combining or separating objects, etc. Its behavior is defined through properties that are specified via its user interface, the <i>Process Flow</i> logic builder, or custom computer code using <i>FlexScript</i> .
Object Library	A list of available objects that can be selected, dragged, and dropped onto the modeling or work surface. The primary <i>FlexSim</i> library contains Fixed Resources, Task Executors, Conveyors, Fluids, Modules, etc. User libraries of objects can be developed that contain special user-developed objects. Other libraries are available for defining logic in Process Flow and creating charts and graphs.
Object Flow Diagram	A diagramming methodology used during the conceptual design phase of model development to identify and represent the various system elements and relationships that must be considered by the simulation.
Operational dynamics	The changing behavior and performance of an operations system over time.
Optimization	The search for the “best” solution, i.e., the best means for running an operations system. It involves changing system parameters or characteristics and evaluating the resulting performance. Optimization algorithms provide means for effectively carrying out the search.
Operations system	A collection of elements that transform input into output through a set of related activities and processes that require a variety of resources, such as equipment, material, people, and information. Operations systems are characterized by complex interactions among resources with numerous sources of variability, which drives its dynamic behavior, i.e., changing behaviors and performance over time.
<i>OptQuest</i>	A simulation optimization software product developed and maintained by OptTek Systems, Inc. that provides algorithms and analysis techniques for determining the best input values to obtain the best outcomes.

Parameter Table	A tool that organizes and stores global variables that can easily be accessed from anywhere in a <i>FlexSim</i> model. The values in the table can be updated in the table rather than within an object in a model. The values are also easily accessed by the <i>Experimenter</i> to facilitate evaluating alternative <i>scenarios</i> and drive an <i>optimization</i> .
Path Network	A set of connected nodes that defines a path on which Task Executors (TE) can travel. In <i>FlexSim</i> , straight-line or curved paths restrict the direction of travel, speed, and whether passing on the path is permitted. In contrast to <i>A-Star</i> , which also controls Task Executor movement, a Path Network explicitly restricts or limits where a TE can travel, whereas A* identifies what a TE needs to avoid or where it cannot travel.
Performance Measure	An output of a simulation model that is of interest for analyzing and understanding the estimated behavior of the system being considered.
Performance Measure Table	A tool that defines, organizes, and stores key performance indicators. The values are also easily accessed by the <i>Experimenter</i> to facilitate evaluating alternative <i>scenarios</i> and drive an <i>optimization</i> .
Port	A means for items to move in and out of objects (input and output ports). Also, it provides a means for objects to communicate (center ports). Whenever two items are connected, a port is created on each object. Theoretically, there is no limit on the number of ports an object may have.
Process Flow	Drag-and-drop, flowchart-like logic builder within <i>FlexSim</i> that uses various types of activities and their properties to define inter-object or intra-object logic.
Pull logic	A means of flow between objects where the decision as to which object an item is obtained from is determined by the receiving object, not the sending object. See also <i>Push logic</i> .
Push logic	A means of flow between objects where the decision as to which object receives the item is determined by the sending object, not the receiving object. See also <i>Pull logic</i> .
Queueing System	An operation composed of three main elements - customers, servers, and queues (also referred to as buffers) where customers require service from one or more servers. If a server is not available when the customer demands the service, the customer waits in a queue until a server is available. The interactions among the three elements result in the behavior of the system.
Reliability	The probability that an object performs its desired functions under certain conditions for a period of time. Another way to state this is that reliability is the probability that an object does not fail over a period of time. See <i>MTBF/MTTR</i> for the means to specify reliability in <i>FlexSim</i> .

Replication	Repeating the execution of a simulation model using a different random number stream. Replications are necessary because simulations involve random variables or stochastic parameters. As a result, system performance varies from simulation run to simulation run, just as in the actual system.
Routing	The logic that is used to move items between objects. Commonly, items are “pushed” from one object to another but may also be “pulled” from one object to another. Routing is typically defined on an object’s Output pane in the Properties window. • Push logic involves deciding which output port will be used. The decision might consider sending an item based on the first downstream object available, the one with the shortest queue, a condition based on the value of the item’s attribute (label), the state of the system, etc. • Pull logic options are similar to those used for pushing, but the decision involves which input port on the object should be used to obtain an item.
Run Time	The duration of a simulation run in simulated time, not real-time. Time in <i>FlexSim</i> is unitless; it is given context through a user’s specification of time units (e.g., seconds, minutes, days). If a user specifies the model units as seconds, then a Run Time of 10,000 is 10,000 seconds of simulated time (about 2.8 hours).
Run Speed	How fast a model runs relative to real time. For example, if a model’s units are specified as seconds, then a Run Speed of 1000 means the model runs 1000 times faster than in real time. Therefore, if a model’s Run Time is 10,000 seconds (about 2.8 hours) and Run Speed is set to 1000, the 2.8-hour simulation will take 10 seconds to run.
Scenario	A combination of a model’s decision variable values. The values of the variables are changed during experimentation to determine their effect on estimated system performance.
Simulation	A process that involves the modeling and analysis of an operations system to improve organizational performance.
Simulation Analysis	Simulation models are built for analysis. As such, analyses are the ways and means of using a simulation model to experiment with and test ideas and alternatives before deciding actions and committing resources. Analyses may be performed within the software or exported to other software. In addition to output analyses, input analyses are needed to define system properties and represent them in a model. Oftentimes, input analyses involve selecting the most appropriate probability distributions to represent aspects of the system, e.g., process times, inter-arrival times, and quality.

Simulation Modeling	The ways and means for representing a system physically (size, distance, speed, etc.) and logically (what, who, when, and where things are done, as well as how much and how long) in order to understand its behavior over space and time and to assess possible consequences of actions, virtually.
Slot (Rack)	A storage space in a Rack object that is a subdivision of a <i>Bay</i> in the horizontal direction.
State	A condition of a system or the value of a system variable, such as whether a resource is busy or idle or how many customers are currently in the system.
Staytime	How long an item stays in an object.
Study Model	A small model for testing and validating a method or concept in isolation before incorporating it into the primary model. This good modeling practice is commonly used to develop logic for a single object or set of objects.
Table	A systematic means to store information. In <i>FlexSim</i> , tables can be imported from or exported to Microsoft Excel. See either <i>Global Table</i> , <i>Time Table</i> , or <i>Parameter Table</i> .
Task Executer	A mobile resource that moves about in a model, typically to move items between objects. They include operators, fork trucks, AGVs, cranes, etc. As the name indicates, the object executes a sequence of tasks that are sent from another object. Tasks include traveling between objects, loading an item, processing an item, etc. Tasks can be prioritized and can preempt other tasks.
Task Sequence	A set of tasks that a Task Executer (operator, fork truck, crane, etc.) carries out. Typically, task sequences are requested and sent by fixed resources or Process Flow activities and involve tasks such as traveling between objects, loading an item, processing an item, etc.
Theoretical Distribution	A probability distribution such as the Beta, Exponential, Lognormal, Normal, Triangular, Uniform, Weibull, etc. that represents a discrete or continuous property. For example, an arrival process may be exponentially distributed or a processing time may be considered to be triangularly distributed.
Time Table	A means to define and specify a resource's deterministic availability, such as shift schedule, operator break times, periodic inspections, etc.
Token	The basic component of <i>FlexSim</i> 's Process Flow logic. Tokens are analogous to flowitems in 3D in that they flow through activities as a simulation runs. However, tokens are typically more abstract than flowitems since they usually represent logical flow rather than physical flow. Each token has a set of labels or characteristics.

Toolbox	A type of library that contains modeling tools or aides, such as data tables, timetables, storage systems, etc.
Transfer	Objects that are automatically created in a model whenever a conveyor segment is connected to another conveyor segment or another type of fixed resource. There are Entry and Exit Transfers.
Tree	The hierarchical, object-oriented data structure that stores a model's data. In <i>FlexSim</i> , all of this data is readily accessible.
Trigger	A place in a model, typically in an object, where optional actions can be defined to occur when certain conditions are met as a model runs, such as when an item enters/exits an object, a process is finished, a message is received, the model is reset. There are many types of actions that can occur when a trigger is "fired," such as changing an item's appearance (shape, color, etc.), reading/writing a table or label value, opening/closing ports, sending a message, etc.
Units	Units of measure - time, length, and volume (for fluid-based models only) - must be specified at the beginning of a simulation.
Validation	The process of determining if a model is accurate enough for its intended use.
Verification	The process of determining if a model has been implemented correctly.
Warm up	A period of time at the beginning of a simulation where statistics are not collected. This allows a simulation to get to a point where the conditions are more representative of those that would occur in the actual system. For example, a simulation may start with all resources empty and idle, which may not be typical in the real system. Thus, the statistics would be biased low if this early time is included in an analysis.

APPENDIX B – PROGRAMMING-BASED PROPERTY VALUES

Appendix B describes and explains elements of *FlexScript* (a subset of C++) that are encountered in the primer when discussing setting some property values.

item.Type

The value of the label named Type on the flow item being processed.

current.Type

The value of the label named Type on the current object (e.g., Source, Processor, Task Executer), the object currently being processed.

exponential (0, 10, getstream(current))

A function that returns a sample value from an exponentially distributed random variable with a mean of 10. The random number stream is based on the object making the function call. Random sampling and random number streams are general simulation topics, but discussion is beyond the scope of this *FlexSim* primer.

getstream(current)

A function that returns a random number stream value based on the object making the function call. Random sampling and random number streams are general simulation topics and are not discussed in this *FlexSim* primer.

duniform(1,3, getstream(current))

A function that returns a sample value from a uniformly distributed random variable with discrete values between 1 and 3; basically, it returns a value of 1, 2, or 3 with equal likelihood. The random number stream is based on the object making the function call. Random sampling and random number streams are general simulation topics, but they are beyond the scope of this *FlexSim* primer.

triangular (10, 35, 15, getstream(current))

A function that returns a sample value from a triangularly distributed random variable with a minimum value of 10, a maximum value of 35, and a most-likely value of 15. The random number stream is based on the object making the function call. Random sampling and random number streams are general simulation topics, but are beyond the scope of this *FlexSim* primer.

Empirical(EmpiricalDistribution1).get(getstream(current))

A *FlexScript* class that returns a sample value from an empirically distributed random variable. The empirical distribution name EmpiricalDistributiou1 is user defined, and the distribution may be discrete, continuous, or fitted using system data to a theoretical distribution. The random number stream is based on the object making the call. Random sampling and random number streams are general simulation topics, but are beyond the scope of this *FlexSim* primer.

Model.parameters.FinishTime_

Provides the value of the model parameter named FinishTime_

`Color.random()`

Sets the color of an item or object to a randomly-selected color.

`Color.yellow`

Sets the color of an item or object to yellow.

`Color.byNumber(item.Type)`

Sets the color of an item to the current value of an item's label named `Type`. *FlexSim's* default color palette is 1 = red, 2 = green, 3 = blue, 4=yellow, etc.

`current.centerObjects[1]`

References the object connected to the current object's first center port.

`current.setProperty("MaxContent", Model.parameters.BufferSize);`

A code snippet that sets the value of the current object's property named `MaxContent` to the value stored in the Model Parameter named `BufferSize`.

`current.outObjects[1].Cover == 0`

A conditional test to see if the value of a label named `Cover` on the object that is connected to the current object's first output port is equal to 0.

`Model.parameters["InitInv-Comp_A"].value`

Obtains the current value of the Model Parameter named `InitInv-Comp_A`

`token.CompNum == token.NumComp`

A conditional test to see if the value of a token label in Process Flow named `CompNum` is equal to the value of another token label in Process Flow named `NumComp`.

`token.Component.CompType`

References the value of an object's label named `CompType`; the object's reference is stored in a token's label named `Component`.

`Table("ComponentReference").executeCell(token.CompType, "ROP")`

Executes a *FlexScript* code statement located in a cell in the Global Table named `ComponentReference`. The cell is located in the row given by the value of the token label `CompType` and the column named `"ROP"`.

`token.Storage.subnodes.length`

The number of items currently in the object that is stored in the token label named `Storage`.

More information on coding in *Flexscript* is available in the *FlexSim User Manual* and the following: another primer by this author entitled *A Primer on Coding in FlexSim 2017*. Section 3 of Chapter 12 in the *Applied Simulation Modeling and analysis Using FlexSim* textbook.

APPENDIX C – MODEL SUMMARIES

Appendix C includes a brief description of the concepts and software features added in each primer model and a reference to the section where the model is discussed.

Finish Filename Base Filename	Primer Reference	Brief Description
MyFirstModel Base: none	Chapter 5	Simple, single-server queueing system using all default values. Basic 3D objects and connections. On-object output statistics.
Primer_1 Base: MyFirstModel	Chapter 7	Initial model of Finishing Area. Customization of the Source (triangularly-distributed inter-arrival times, new flowitem class for containers, On Creation trigger to define the label Type and set the item's color), Queue (maximum content and item placement), and Processor (process time based on Type, set-up time). Create new item, container, in Flowitem Bin.
	Chapter 8	Setting Run Time and Run Speed. On-object statistics. Dashboards, add time-series plots of Queue contents over time, both total and by container type, histogram of time waiting in Queue.
Primer_2 Base: Primer_1	Chapter 9	Incorporate Finishing Operator to move containers from container Queue to Finishing Machine Processor. Use Dispatcher object to control Operators. Operator performs setup operation on Processor. Customize Operator properties (speed, load/unload time, do not travel offset).
Primer_3 Base: Primer_2	Chapter 11	Change Finishing Machine and Container Queue graphics. Add a second Finishing Machine. Employ new routing rules for (1) moving containers between the container Queue and Finishing Machines and (2) if no space is available when a container arrives. Create Model Views.
Primer_4 Base: Primer_3	Chapter 12	Define product mix as an Empirical Distribution. Fit arrival process data using FlexSim's curve fitter and use it to specify the probability distribution that best represents the interarrival time distribution for containers.
Primer_4A Base: Primer_4	Chapter 13	Introduce Model Parameters and Global Tables. Use Parameters Table to store finishing times. Use Global Table to store the time and type of each container as they arrive to the system.
Primer_5 Base: Primer_4A	Chapter 14 (Sect 14.1-14.2)	Add downtime for break and lunch times for the Finishing Operator and a chart to summarize utilization and percentage of time the Operator is in different
Primer_6 Base: Primer_5	Chapter 14 (Sect 14.3-14.5)	Add a quality check type of downtime on the Finishing Machines, which occurs on a fixed time interval and requires no resources. Add a composite state chart for the Finishing Machines.
Primer_7 Base: Primer_6	Chapter 14 (Sect 14.6-14.7)	Add random breakdowns on the Finishing Machines that has random repair times and needs the Finishing Operator to perform the repair. Add two state charts that summarize utilization and percentage of time in various states - one for the Finishing Operator, and one that shows the states for each Finishing

Finish Filename Base Filename	Primer Reference	Brief Description
Primer_8 Base: Primer_7	Chapter 15 (Sect 15.2-15.5)	Create flow items for each type of component and a Source to generate batches of each type of component at a specified frequency and quantity. Create a data table to store information on operations. Use Model Parameters Tables to store each type of component's batch frequency (time between batches) and batch size. Provide a queue for batches to await processing. Provide storage areas for components, i.e., each type of component has its place to be stored in the
Primer_9 Base: Primer_8	Chapter 16 (Sect 16.1-16.6)	Use a Separator object to unpack components. For each batch, the Finishing Operator moves it from the queue to the Packing Area, unpacks it, and loads the components into its storage location. Connect Finishing and Packing Area by a simple conveyor. Using the Combiner and Robot objects, containers are packed with components based on their type. The Robot incurs downtimes that the Finishing Operators addresses. Additional model views are created. Charts are used view component inventory levels and assess batch policies.
Primer_10 Base: Primer_9	Chapter 17 (Sect 17.1)	The first of two alternative means to control the travel paths of Task Executors when moving between objects is applied. This alternative uses Path Networks, a set of connected node objects, which defines restricted paths for the TE to
A-StarStudy Base: None	Chapter 17 (Sect 17.2.1)	A simple study model is used to introduce the basics of using the A* Navigator to control Task Executor travel.
Primer_10A Base: Primer_10	Chapter 17 (Sect 17.2)	The second of two alternative means to control the travel paths of Task Executors when moving between objects is applied. This alternative uses the A* Navigator, which employs the A* algorithm to define the shortest path between objects while avoiding specified barriers.
Primer_11 Base: Primer_10A	Chapter 18 (Sect 18.1-18.2)	<i>FlexSim's</i> Experimenter is introduced, including setting up experiments, running multiple replicated scenarios, and analyzing the results. Two examples are included, which assess the effect on performance of: (1) the size of the buffer of containers prior to Finishing and (2) component replenishment plans.
Primer_12 Base: Primer_11	Chapter 19 (Sect 19.2)	Connect the Finishing and Packing Areas via a loop conveyor. Each Finishing Machine feeds containers to the loop conveyor. A spur line connects the loop conveyor to the Packing Area. Controls (Decision Point and Photo Eye) are added so that if the spur line becomes full, containers travel on the loop conveyor until the spur has capacity.
Primer_13 Base: Primer_12	Chapter 20 (Sect 20.1-20.2)	A few Parameter updates are made: higher arrival rate of containers, change in product mix, and change in component delivery frequency and batch size. Also, two new charts are added to track the contents of the conveyors. Change the routing logic between the container queue and the Finishing Machines to a simple Shortest Processing Time rule - Type 1 containers are always processed first, the Type 2, then Type 3.

Finish Filename Base Filename	Primer Reference	Brief Description
Primer_14 Base: Primer_13	Chapter 20 (Sect 20.3)	Change routing logic between container queue and Finishing Machines to the Shortest Processing Time rule unless a container has waited more than a specified threshold value, then it has priority. This is implemented using the List
Primer_15 Base: Primer_14	Chapter 21 (Sect 21.2)	Introduction to Process Flow. Add three Model Parameters Tables to support modeling in Process Flow - for each component, add its reorder point, reorder time, and initial inventory
	Chapter 22 (Sect 22.1-22.2)	The first Process Flow example provides the capability to have a model start with an initial inventory of components rather than all stores being empty at the start; this provides a more realistic starting condition. This addition includes such Process Flow activities as scheduled source, assign labels, create objects, decision, and sink. Also, a Global Table is created that contains all of the information on the components.
Primer_16 Base: Primer_15	Chapter 23 (Sect 23.1)	Process Flow is used to model a reorder-point inventory system, which is used to control the inventory of components. Prior to creating the logic, several changes are made to existing 3D objects for aesthetics, to facilitate using Process Flow, and to create a generalized solution. The changes include: (1) incorporating a non-functional Visual object (a table) beneath the component storage objects in the Packing Area, (2) resetting the colors of the component storage objects at the start of a simulation, (3) creating a component order item, (4) removing objects for scheduled orders, (5) creating a storage Group, and (6) adding and updating data in the components' Global Table.
Primer_17 Base: Primer_16	Chapter 23 (Sect 23.2-23.3)	Use of Process Flow to implement a reorder-point inventory management system for the components that are packed into containers. A wide variety of Process Flow activities are used, including custom task sequences for the Finishing Operator, event-driven source, assign labels, decisions, breathe, sub-flows, create and delete objects, visual changes, decisions, and custom code. In addition, to help verification and validation, the example introduces breakpoints and token coloring
Primer_18 Base: Primer_17	Chapter 24	Introduce the Standard Rack, a 3D object, and use it to store containers in a Warehousing Area. Customize the size and configuration of the Racks. Each container type is stored in its own Rack. Add a Model View of the Warehousing Area. Create a time series chart that shows the the number of containers in the Warehousing Area by type.

Finish Filename Base Filename	Primer Reference	Brief Description
Primer_19 Base: Primer_18	Chapter 25 (Sect 25.2-25.3)	Define the order-fulfillment activities in Process Flow, which includes generating orders (random times and content) for containers and having an Order Picker gather the appropriate containers, fulfill the order, and update an information system. Order mix is an Empirical Distribution read from <i>Excel</i> . An order is a new type of flow item; it contains the containers in an order. Filling orders from containers on Racks are managed through a List. The Racks may contain initial inventory at the start of a simulation. An output table is generated that captures information about each order. Charts are added to track the time to fulfill orders and how many orders wait to be processed. Average time to fulfill an order added as a Performance Measure.
Primer_20 Base: Primer_19	Chapter 26 (Sect 26.1-26.5)	Add an AGV system to transport containers from the Packing Area to the Warehousing Area. The system includes a Control Area to restrict traffic in the area to one Task Executer. A small test segment is temporarily added to verify that the Control Area works properly. Base floor color is changed. Chart is added to track the AGV's utilization. Utilization charts are also added for the Robot and Order Picker.
Primer_Final Base: Primer_20	Chapter 5-26	Full model with all windows closed.

ABOUT THE AUTHOR

Allen G. Greenwood, Ph.D., P.E.(retired)

allen.greenwood@autodesk.com



Currently, Allen is a Simulation Education Specialist at Autodesk and Professor Emeritus of Industrial and Systems Engineering at Mississippi State University. One of his long-term, foundational professional goals, both in practice and in academe, has been to enhance and increase the application of simulation to support the problem-solving and decision-making processes.

In addition to his faculty positions at Mississippi State, he was Professor of Engineering Management at Poznan University of Technology in Poland, Professor and Chair of the Department of Engineering Management at Prince Sultan University in the Kingdom of Saudi Arabia, Professor of Engineering at the American University of Armenia, and Assistant Professor of Management Sciences at Northeastern University and Virginia Tech.

At all of these institutions, he developed and taught courses in systems simulation at the undergraduate, graduate, and professional levels. In addition, he taught courses in operations research / management science, logistics systems design, enterprise systems engineering, project management, statistics, decision analysis, information systems, etc.

Allen's research interests/expertise include the design and analysis of production and project systems; simulation modeling, analysis, and optimization; and the design and application of decision-support systems.

Before joining academia, he held engineering and supervisory positions at American Enka Company and General Dynamics Corporation.

During his academic career, he has led or was a principal contributor to numerous industry projects, mainly in systems simulation. As a result, his professional experience spans a wide variety of domains -- engineering design and development (military aircraft and aerospace), manufacturing and production systems (military aircraft, shipbuilding, automotive, textile fibers, healthcare, electrical systems, material handling systems, consumer products, etc.), and project management. His work has been funded by such organizations as the US Air Force Research Laboratory, Office of Naval Research, Naval Sea Systems Command, NASA, Northrop Grumman Ship Systems, Nissan North America, General Electric Aviation, the Center for Advanced Vehicular Systems (MSU), and Poznan University of Technology.

He has authored or co-authored over 150 creative works, including journal and conference papers, technical reports, software programs, etc. In addition, he is co-author of *Applied Simulation: Modeling and Analysis Using FlexSim*, currently in its fifth edition, and recently authored two primers, one focused on simulation in general and one focused on using *FlexSim* software.

Allen received his B.S.I.E, M.S.I.E, and Ph.D. (Management Science) degrees from North Carolina State University, University of Tennessee, and Virginia Tech, respectively. He is a registered Professional Engineer (retired) in Texas.